Acceleration of Biomedical Image Processing and Reconstruction with FPGAs

Dissertation
for attaining the Ph.D. degree
of Natural Sciences

submitted to the Faculty of Computer Science and Mathematics
of the Johann Wolfgang Goethe University
in Frankfurt am Main, Germany

by Frederik R. Grüll
born in Heidelberg, Germany

Frankfurt am Main 2014

accepted by the Faculty of Computer Science and Mathematics of the Johann Wolfgang Goethe University as a dissertation.

Dean:	Prof. Dr. Thorsten Theobald
Expert assessors:	Prof. Dr. Udo Kebschull
	Prof. Dr. Ivan Kisel
Date of disputation:	February 6, 2015

Abstract

Acceleration of Biomedical Image Processing and Reconstruction with FPGAs

Field Programmable Gate Arrays (FPGAs) are microchips containing a large number of logic blocks that can be freely configured and connected to calculate any digital logic. Increasing chip sizes and better programming tools have made it possible to push the boundaries of application development with FPGAs. Originally used as "glue logic", the compute-intense core of an application can now be computed by an FPGA and accelerate its execution.

In this thesis the potential of FPGA acceleration is examined for applications that perform biomedical image processing and reconstruction. The dataflow paradigm was used to port the analysis of image data for localization microscopy and for 3D electron tomography on an FPGA. The implementation of both applications consists of long pipelines of logic operators that all compute on the image data in parallel on the FPGA. An acceleration of 185 compared to an Intel i5 450 CPU was achieved for localization microscopy, and electron tomography could be sped up by a factor of 5 over an Nvidia Tesla C1060 while maintaining full accuracy in both cases. For the porting to be efficient, most of the original imperative source code had to be dissected and re-written in a form suitable for dataflow computing.

Dataflow computing describes programs in terms of the flow of data in a directed graph, where the nodes form operators such as adders, multiplexers, accumulators or counters. Source code for existing high-performance applications, however, is usually written in C-like imperative languages to be run on CPUs or graphics cards. The graphs in dataflow computing closely resembles the data dependencies between operations, while imperative code in its basic form describes the control flow of a program in terms of sequences of statements, if-else branchings and loops.

Imperative code can be compiled to machine code, and since the invention of the von-Neumann architecture multiple auxiliary units (i. e. caches, branch prediction, out-of-order execution) have been added to speed up the execution of arbitrary sequences of machine instructions. Custom hardware on FPGAs, on the other hand, can run most effectively if the program is synthesized as a system of long pipelines close to the dataflow description that all run in lock-step and ideally process one data item per clock cycle, resulting in massive parallelism. Despite the slow starts in clock frequency and silicon area owed to the reconfiguration overhead, a custom design on FPGAs can more than compensate by omitting the auxiliary units found on CPUs and promises significant acceleration and savings on power and rack space.

The suitability of FPGA accelerators was examined for biomedical imaging application during the creation of this thesis, as well as the process of transformation from an imperative source towards a dataflow description. It was known before that FPGAs are well-suited for image processing, were a pipeline can apply the same operations repeatedly until all pixel values of an image or movie frame have been processed [1]. However, the suitability was less clear if FPGAs could accelerate bigger applications from high-performance computing that combine image processing with feature extraction or reconstruction and a more complicated control flow. Two applications were picked as prototypes to examine the process of porting and the accelerator gain. The implementation was carried out with the MaxCompiler library from Maxeler Technologies [2].

Localization microscopy improves the resolution limit of optical light microscopy by an order of magnitude [3, 4]. The Abbe diffraction limit states that two point-like objects with a distance d smaller than about half the wavelength of light cannot be distinguished any more. For a single object, however, the localization can be determined more precisely than d by fitting its diffraction disk with a Gaussian profile. Stochastic localization microscopy uses fluorophores that switch between two different spectral states and appear blinking when illuminated with laser light. The bright state is much shorter than the dark one and allows the signals to be separated after the sample was recorded for several minutes. The processing consists of removing the background glow for

every frame, finding the spots of the bright fluorophores, and fitting them in a region of interest (ROI) to extract the precise location and its confidence.

For acceleration every processing pipeline had to be re-designed. The background measurement was changed to exponential smoothing for every pixel over time. The spot finder was modified to operate after the background was subtracted. The former least-square fit could be simplified to a Gaussian estimator for feature extraction. The resulting pipeline system then consists of two statically scheduled pipelines connected by a FIFO. The first pipeline operates on entire frames, and the second extracts the features of every detected spot and operates on the ROIs only.

The second application reconstructs the density distribution in a 3D volume from 2D images obtained with an electron microscope from multiple angles. The method belongs to the class of computed tomography, which is widely used in medicine and biology. The Simultaneous Algebraic Reconstruction Technique (SART) [5] reconstructs the volume reliably with about 50 images as an input. It starts with an empty volume in memory and calculates a projection by casting rays on a virtual detector (forward projection). The projection is then compared with the corresponding image from the electron microscope. The difference is spread back equally on the rays into the volume in memory (back projection). The process is repeated for the next image until all images have been taken into account. Before this thesis, only parts of the back projection were implemented on an FPGA [6], but not full SART.

The existing implementation of SART for graphics cards [7] followed each ray through memory for forward and back projection, causing a random memory access pattern in the volume storage. For the FPGA application, the voxels in the volume in DRAM are addressed linearly in burst mode instead, and the corresponding rays are calculated thereafter. The final design encompasses two statically scheduled pipelines: the first pipeline accumulates the virtual projection for the current angle, and the second one projects back the difference between virtual projection and the microscopic image. Due to beam declination, the problem could not be reduced to a stack of 2D projections.

Both applications were implemented on FPGAs with the tool flow from Maxeler Technologies and Xilinx. The hardware design for localization microscopy occupied less than 1/3 of a Virtex-5 LX330T FPGA. The accuracy of results was the same as the previous software implementation. It was examined independently [8] and proved to be in line with other algorithms. A speed-up of 100 was achieved due to the re-write of the algorithm in software, and a factor of 185 due to hardware acceleration, yielding a total acceleration of 18 500. The design is capable of analyzing 167 Mpx/s, 2.5 times faster than the MaLiang implementation on graphics cards [9]. The hardware took three months to build for one person after extensive Monte-Carlo simulation. [10]

The application for electron tomography was carried out on a Virtex-6 SX475T FPGA. The size of the chip allowed to process three voxels per clock cycle and occupied 3/4 of the Digital Signal Processors (DSPs), which were the limiting resource. Both projection pipelines have close to 100 stages and run at 170 MHz, yielding a throughput of 510 Mvoxel/s. Compared to the previously mentioned graphics cards implementation on a Nvidia Tesla C1060, the FPGA implementation is five times faster. The hardware design required a development time of eight months. [11]

The results show that localization microscopy and electron tomography applications could be successfully accelerated with dataflow computing on FPGAs where they even outperform graphics cards. With tools to automatically map a high-level dataflow description on the FPGA, the main challenge was found in the translation of the algorithm into a form that harnesses the strengths of an FPGA, namely very high pipeline parallelism, resource-efficient fixed-point number operations with custom range and precision, and fine control over all memory access. The translation process from software to hardware that is described in this thesis ensured effective translation to a dataflow design, efficient implementation on reconfigurable hardware, and an acceleration that can most likely be achieved for future high-applications in image processing for high-performance computing.

Beschleunigung biomedizinischer Bildverarbeitung und -rekonstruktion mit FPGAs

Field Programmable Gate Arrays (FPGAs) sind Microchips, die aus einer großen Anzahl von frei konfigurierbaren Logikblöcken bestehen und beliebige digitale Schaltungen abbilden können. Anhaltende Verbesserungen bei Chipgröße und Programmierwerkzeugen ermöglichen es, FPGAs auch für größere Anwendungen einzusetzen. Der rechenintensive Teil einer Anwendung wird auf den FPGA ausgelagert und beschleunigt die Ausführung.

In dieser Dissertation wurde die Eignung von FPGAs für die Ausführungsbeschleunigung für Anwendungen untersucht, die zur Bildverarbeitung und -rekonstruktion in der Biomedizin eingesetzt werden. Dazu wurde die Bildanalyse für die Lokalisationsmikroskopie und die Bildrekonstruktion in der dreidimensionalen Elektronentomografie in eine Datenflussbeschreibung überführt und als Pipeline-System auf dem FPGA implementiert. Für die Lokalisationsmikroskopie konnte eine Beschleunigung von 185 gegenüber einer CPU (Intel i5 450) erreicht werden, und die Elektronentomografie wurde um einen Faktor fünf gegenüber einer Grafikkarte (Nvidia Tesla C1060) beschleunigt. In beiden Fällen konnte die Genauigkeit der Ergebnisse erhalten werden.

Ein Datenflussgraph beschreibt ein Programm als Fluss von Daten in einem gerichteten Graphen, in dem die Knoten aus Operatoren wie Addierer, Multiplexer, Akkumulatoren und Zählern bestehen. Der Quellcode einer Anwendung aus dem wissenschaftlichen Rechnen ist im Gegensatz dazu gewöhnlich in einer imperativen, C-ähnlichen Sprache geschrieben und wird auf CPUs oder Grafikkarten ausgeführt. Der Datenflussgraph beschreibt die Datenabhängigkeiten zwischen den Operationen, während imperativer Quellcode in seiner einfachsten Form den Kontrollfluss eines Programms mit Sequenzen, Verzweigungen und Schleifen definiert.

Imperativer Code lässt sich effizient zu Maschinencode für die von-Neumann Architektur übersetzten, und seit ihrer Erfindung wurden zahlreichen Bestandteile eingeführt, um die Ausführung von Instruktionen mit beliebiger Reihenfolge zu beschleunigen. Hardware, die auf die Anwendung angepasst werden kann, kann auf Einheiten wie die Sprungvorhersage, Caches und Out-of-order execution oftmals vollständig verzichten. Stattdessen wird das Programm in Hardware von einem Pipelinesystem ausgeführt, das dem Datenflussgraphen folgt, und jede Pipeline verarbeitet massiv parallel pro Takt ein Datum. Der nachteilige Bedarf für die Rekonfigurationslogik des FPGAs wird dadurch mehr als ausgeglichen und es kann eine Beschleunigung in der Ausführung bei gleichzeitigen Verringerung des Energiebedarfs erreicht werden.

Für die Beschleunigung von Anwendungen aus der biomedizinische Bildverarbeitung bildete die effektive Übersetzung des imperativen Quellcodes auf eine Datenflussbeschreibung einen Kernbestandteil der Arbeit. Für die allgemeine Bildverarbeitung ist bereits bekannt, dass sich FPGAs gut für die Verarbeitung von Pixeldaten eignen, wenn jedes Pixel die gleiche Verarbeitungskette durchläuft [1]. In dieser Arbeit wurde darauf aufbauend die Eignung von FPGAs für Anwendungen im wissenschaftlichen Hochleistungsrechnen untersucht, die Bildverarbeitung mit Bildanalyse und -rekonstruktion verbinden und sich durch einen verschachtelteren Kontrollfluss auszeichnen. Dazu wurden exemplarisch zwei Anwendungen analysiert, um die Portierung als zielgerichteten Prozess und die erhaltene Beschleunigung zu untersuchen. Beide Anwendungen wurden mit der MaxCompiler-Bibliothek auf FPGAs implementiert [2].

Lokalisationsmikroskopie verbessert die Auflösungsgrenze von optischer Lichtmikroskopie um etwa eine Größenordnung [3, 4]. Nach der Abbeschen Auflösungsgrenze können zwei Objektpunkte mit einem Abstand d kleiner als die halbe Wellenlänge nicht mehr optisch unterschieden werden. Für einen einzigen Punkt kann die Position jedoch mit einem kleineren Fehler als d bestimmt werden, indem man seine Beugungsscheibe mit einer Gaußschen Verteilungskurve annähert. Für die stochastische Lokalisationsmikroskopie werden Fluorophore verwendet, die unter Laserlicht zwischen verschiedenen Spektralzuständen wechseln und blinkend erscheinen. Der helle Zustand besteht viel kürzer als der dunkle und ermöglicht eine optische Trennung der Fluorophore,

nachdem ein Film der Probe einige Minuten aufgezeichnet wurde. Für die anschließende Analyse wird der Hintergund entfernt, die Signale der hellen Fluorophore gesucht und schließlich die Beugungsscheibe vermessen, um den genauen Ort und seinen Fehler zu erhalten.

Zur Beschleunigung wurde jeder algorithmische Schritt neu entworfen. Die Hintergrundabschätzung wurde auf exponentielles Glätten umgestellt, und die Signalsuche erfolgt erst nach Abzug des Hintergrunds. Der gaußsche Fit wurde von der Methode der kleinsten Fehlerquadrate auf Maximum-Likelihood umgestellt. Das danach erhaltene Pipelinesystem besteht aus zwei Pipelines, die auf dem FPGA über einen FIFO-Puffer verbunden sind. Die erste Pipeline verarbeitet die Einzelbilder, die zweite bestimmt die Eigenschaften der gefunden Beugungsscheiben.

Die zweite Anwendung, die Elektronentomografie, rekonstruiert die Dichteverteilung in einem 3D-Volumen aus 2D-Bildern, die von einem Elektronenmikroskop aus verschiedenen Winkeln aufgenommen wurden. SART (Simultaneous Reconstruction Technique) [5] rekonstruiert die Dichteverteilung zuverlässig aus etwa 50 2D-Bildern. Die Methode beginnt mit einem leeren Volumen und berechnet eine Vorwärtsprojektion durch Elektronenstrahlen auf einen virtuellen Detektor. Der Unterschied zum realen Bild aus dem Elektronenmikroskop wird danach gleichmäßig entlang der Strahlen in das Volumen zurückprojeziert. Der Prozess wird fortgesetzt, bis alle Bilder verarbeitet worden sind. In dieser Dissertation wurde nicht wie zuvor nur Teile der Rückprojektion [6], sondern erstmals der vollständige SART-Algorithmus auf einem FPGA implementiert,

Die bereits vorhandene Implementierung für Grafikkarten [9] folgte bei der Berechnung der virtuellen Projektionen dem Elektronenstrahl durch das Volumen und erzeugte ein nicht-lineares Zugriffsmuster im Arbeitsspeicher. Für den FPGA wurden die Voxel stattdessen linear im Burst-Modus adressiert, und die zugehörigen Strahlen wurden anschließend ermittelt. Das Hardware-Design besteht aus zwei Pipelines, in dem die erste die virtuelle Vorwärtsprojektionen berechnet und die zweite die Rückprojektionen durchführt. Wegen Abweichungen im Strahlengang konnte die Anwendung nicht auf 2D-Tomografie reduziert werden.

Beide Anwendungen wurden auf FPGAs mit den Entwicklungswerkzeugen von Maxeler Technologies und Xilinx implementiert. Das Hardware-Design für die Lokalisationsmikroskopie belegte weniger als 1/3 auf einem Virtex-5 LX330T FPGA bei voller Genauigkeit. Die Genauigkeit der Implementierung wurde unabhängig untersucht [8] und bewegte sich im gleichen Rahmen wie die übrigen Softwarelösungen. Der Beschleunigungsfaktor betrug insgesamt 18 500 auf einem Virtex-6 SX475T, davon 100 nach Neukonzeption der ursprüngliche Software und 185 durch den FPGA. Damit können 167 Megapixel/s bei einer Taktfrequenz von 200 MHz verarbeitet werden, 2,5 mal schneller als die MaLiang-Methode für Grafikkarten [9, 10].

Die Portierung für die Elektronentomografie wurde auf einem Virtex-6 SX475T durchgeführt. Die Größe des FPGAs ermöglichte die Verarbeitung von drei Voxeln pro Takt, wobei die digitalen Signalprozessoren (DSPs) die Designgröße beschränkten. Beide Pipelines bestehen aus etwa 100 Stufen, sind mit 170 MHz getaktet und erreichen einen Durchsatz von 510 MVoxel/s. Im Vergleich mit einer Nvidia Tesla C1060 Grafikkarte wurde die Anwendung um einen Faktor fünf beschleunigt.

Lokalisationsmikroskopie und Elektronentomografie konnten erfolgreich mit Datenflussrechnen auf FPGAs beschleunigt werden und sogar die jeweilige Grafikkartenimplementierung überholen. Dank der weitgehend automatisierten Abbildung des Datenflussgraphen auf FPGAs bestand der Kern der Arbeit in der Übersetzung der Algorithmen in eine Form, die die Stärken des FPGAs effizient ausnutzt. Dazu gehört ein hoher Pipeline-Parallelismus, ressourceneffiziente Berechnungen auf Festkommazahlen mit frei wählbarer Kodierung sowie direkte Kontrolle über jeden Speicherzugriff. Der Portierungsprozess von Software zu Hardware wird in dieser Arbeit beschrieben und kann zukünftig verwendet werden, um auch für andere Anwendungen aus dem Bereich der hochperformanten Bildverarbeitung eine beschleunigte Ausführung zu erzielen.

Zusammenfassung / German summary

Seit Beginn des neuen Jahrtausends hat sich die Art und Weise gewandelt, wie die Ausführung von Computerprogrammen beschleunigt wird. Zu Beginn stand noch die mit jeder Computergeneration exponentiell wachsende Taktfrequenz im Vordergrund. Wurde die gewünschte Verarbeitungsgeschwindigkeit einer Anwendung nur um einen konstanten Faktor verfehlt, konnte der Programmierer erwarten, dass zukünftige Rechensysteme die Ausführung ausreichend beschleunigen würden. Mit Erreichen von Taktfrequenzen ab etwa 4 GHz führte diese "mühelose" Beschleunigung jedoch zu Kühlproblemen der Prozessoren, und eine weiter Verbesserung konnte sich nur noch aus der Prozessorarchitektur ergeben. Anstatt weiterhin die serielle Verarbeitungsgeschwindigkeit zu erhöhen verlagerte sich der Fokus auf die parallele Verarbeitung mit mehreren Rechenkernen oder mit Vektorinstruktionen.

Im Jahr 2005 wurden auch zunehmend Grafikkarten als Anwendungsbeschleuniger entdeckt. Ihre Architektur erlaubte es nicht nur, die gleichen Operationen massiv parallel auf einer Vielzahl von Pixeln oder Dreiecken durchzuführen. Auch für andere Rechenprobleme, die sich gut auf diese Parallelität abbilden ließen, konnte der Durchsatz auf einer Grafikkarte gegenüber einer CPU um einen Faktor von bis zu zwei Größenordnungen verbessert werden. Grafikkarten sind heute im wissenschaftlichen Rechnen die am weitesten verbreiteten Anwendungsbeschleuniger.

Eine noch feinfaserige Art der Parallelität kann mit Field Programmable Arrays (FPGAs) erreicht werden. Dabei handelt es sich um rekonfigurierbare Mikrochips, die jede beliebige Digitalschaltung abbilden können, solange ausreichend Platz vorhanden ist. Ein FPGA erreicht diese Fähigkeit, indem er viele konfigurierbare Logikblöcke mit einem schaltbaren Leitungsnetzwerk verbindet. Jeder Block enthält Flip-Flops für Datenregister und implementiert einfache logische Funktionen als Wahrheitstabelle in seinem Speicher. Die flexible Verwendungsmöglichkeit eines FPGAs führt allerdings auch dazu, dass eine Digitalschaltung bis zu 35 mal so viel Siliziumfläche wie auf einem vergleichbarer anwendungsspezifischer Mikrochip benötigt, und dass die Taktfrequenz etwa um einen Faktor zehn darunter liegt. Spezielle Elemente auf dem FPGA für häufig gebrauchte Operatoren wie z. B. Multiplizierer verbessern den Rückstand.

Trotzdem können FPGAs eine Anwendung um ein Vielfaches beschleunigen. Ein Grund liegt darin, dass eine CPU zum Großteil aus Hilfseinheiten besteht, die das Programm nicht direkt ausführen, sonder nötig wurden, um die eigentlichen Recheneinheiten für beliebige Anwendungen möglichst gut auszulasten. Dazu gehören Zwischenspeicher (Caches), die Sprungvorhersage und Schaltungen für die Out-of-order Execution. Ist das Programm schon im Voraus bekannt, kann diese Logik weggelassen oder stark reduziert werden, und es bleibt mehr Fläche für den Algorithmus übrig.

Zu Beginn der Arbeit war bereits bekannt, dass sich FPGAs gut für die Bildverarbeitung eignen [1], wenn einfache Operationen auf allen Pixelwerte eines Bildes durchgeführt werden. Im Gegensatz zur Grafikkarte werden die Operationen nicht gleichzeitig auf einem Teilbild angewendet. Stattdessen wird die Bildoperation als Pipeline implementiert, die mit jedem Takt ein Pixel einliest und ein fertig berechnetes Ausgabepixel produziert. Die einzelnen Verarbeitungsschritte der Bildoperation sind in Hardware umgesetzt und leiten die Zwischenergebnisse wie auf einem Fließband an den nächsten Schritt weiter.

In dieser Arbeit wurde die Eignung von FPGAs als Anwendungsbeschleuniger auf dem Gebiet der biomedizinische Bildverarbeitung und Rekonstruktion untersucht. Dazu wurden zwei Anwendungen beschleunigt: die Bildanalyse für die Lokalisationsmikroskopie und die Bildrekonstruktion für die dreidimensionale Elektronentomografie. Bei beiden Anwendungen handelt es sich um Verfahren, die noch nicht auf FPGAs implementiert wurden, die mit den herkömmlichen Methoden nicht in der gegebenen Zeit hätten beschleunigt werden können und die in der Praxis sehr von einer verkürzten Zeit für die Auswertung profitieren. Das dabei gesammelte Methodenwissen für eine

effiziente Portierung wurde aufgearbeitet und kann für zukünftige Projekte verwendet werden.

Datenflussrechnen

Anwendungen für das wissenschaftliche Rechnen sind in der Regel in einer imperativen Programmiersprache geschrieben. Der Kontrollfluss des Programms wird im Quellcode im einfachsten Fall mit Sequenzen, Verzweigungen und Schleifen dargestellt, die dann vom Compiler in Maschinencode übersetzt werden. Für eine effiziente Portierung eines Programms auf einen FPGA ist jedoch der Datenfluss eines Programms entscheidend. Formalisiert wird er von einem Datenflussgraphen beschrieben, wobei die Knoten Rechenoperationen darstellen und die gerichteten Kanten die Datenabhängigkeiten beschreiben. Der Graph dient dann als Vorlage für die zu implementierende Hardware. Jeder Knoten arbeitet genau dann, wenn an allen Eingängen gültige Daten anliegen und der Ausgang nicht blockiert.

Auf dem FPGA ist eine Implementierung der verwendeten Algorithmen als Pipeline deutlich effizienter, weil im besten Fall alle Pipeline-Stufen zu jedem Zeitpunkt ausgelastet sind. Die massivparallele Verarbeitung erhöht dann den Durchsatz entsprechend. Die Verarbeitung funktioniert besonders effizient, wenn lange Abschnitte einer Pipeline zusammengefasst werden können und im Gleichschritt arbeiten (statische Synchronisation). Für die Flusskontrolle genügt es hier, nur die Eingänge und Ausgänge des Abschnitts zu überwachen und falls nötig den ganzen Abschnitt anzuhalten. Der einzelne Operator benötigt keine eigene Synchronisationslogik mehr.

Sequenzen von Zuweisungsausdrücken ohne Seiteneffekte aus einer Sprache wie C oder Fortran können auf eine Hardware-Pipeline abgebildet werden, indem man sie zuerst in eine Form überführt, in der jeder Variablen nur einmal ein Wert zugewiesen wird (Static Single Assignment Form). Die Syntaxbäume der Ausdrücke können dann zum Datenflussgraph kombiniert werden und geben die Pipeline vor. Für Verzweigungen werden beide Zweige sowie die Bedingung in Hardware implementiert, und ein Multiplexer wählt am Ende das richtige Ergebnis aus.

Bei Schleifen wird zuerst der Rumpf in eine Pipeline überführt. Beim Verbinden des Ausgangs mit einem Eingang muss der Datenfluss jedoch gewöhnlich von Hand geändert werden. Eine zyklische Pipeline mit Latenz l würde sonst die Verarbeitung innerhalb der Schleife in l unabhängige Ausführungsstränge aufspalten. Die verschiedenen Umgehungsstrategien werden gesondert beschrieben.

Im Ergebnis erhält man einen Prozessor, der Listen von Elementen verarbeitet, bei denen es sich zum Beispiel um die Pixel eines Bildes oder die Voxel eines Volumens handeln kann. Anders als in einer CPU, die Listen im Speicher verarbeitet, existieren die Listen auf dem FPGA als zeitliche Abfolge, und eine Verschiebung der Leseposition im Speicher einer CPU entspricht einer Verzögerung mit einer FIFO-Warteschlange in Hardware.

In der vorliegenden Arbeit wurde das Ziel einer möglichst hohen Beschleunigung in einem kombinierten Ansatz verfolgt. Zum einen wurde der FPGA entsprechend den verwendeten Algorithmen und ihrem Datenfluss konfiguriert. Zum anderen wurden alle Algorithmen angepasst oder ausgetauscht, um möglichst wenige FPGA-Ressourcen für Synchronisationslogik aufwenden zu müssen. Zusätzliche FPGA-Ressourcen konnten eingespart werden, indem alle Operationen nur mir der von den Eingabedaten vorgegebenen Genauigkeit berechnet werden. In der Bildverarbeitung liegt die Eingabe für jedes Pixel eines gerasterten Bildes als Ganzahlwert vor, und die Genauigkeit der Zwischenergebnisse ergibt sich aus der Fehlerfortpflanzung. Oftmals kann der Großteil der Zwischenergebsnisse eines Algorithmus mit Festkommazahlen kodiert werden, die deutlich weniger Fläche als die Fließkomma-Operatoren auf dem FPGA belegen und deren Genauigkeit passend gewählt werden kann.

Für die Implementierung wurde die MaxCompiler-Bibliothek von Maxeler Technologies verwendet [2], mit der der Datenflussgraph in einem Java-Dialekt beschrieben wird und dann auto-

matisch für die rekonfigurierbare Hardware übersetzt wird. Die Verwendung einer Hochsprache machte eine erfolgreiche Implementierung zweier Anwendungen in der gegebenen Zeit überhaupt erst möglich, indem sie die statische Synchronisation für Datenflussgraphen mit hunderten Knoten übernimmt und die Verbindung mit der kontrollierenden CPU abstrahiert.

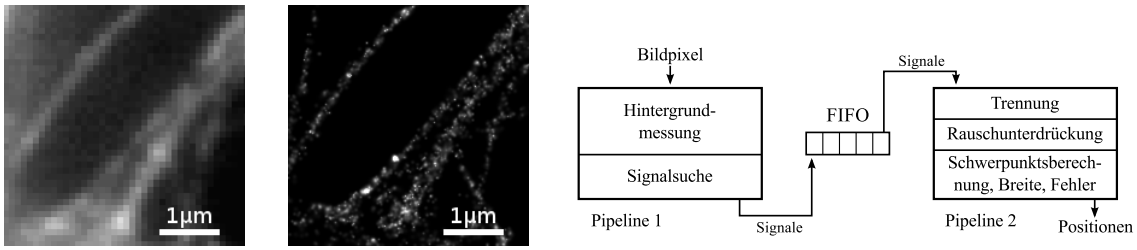
Lokalisationsmikroskopie

Die Auswertung der stochastischen Lokalisationsmikroskopie war die erste Anwendung, die im Rahmen dieser Arbeit beschleunigt wurde. Bei der Lokalisationsmikroskopie handelt es sich um ein Verfahren, das die Auflösungsgrenze von Lichtmikroskopen um etwa einen Faktor zehn verbessert [3, 4]. Nach der Abbeschen Auflösungsgrenze können zwei eng beieinander liegende Punkte nur dann noch unter einem konventionellen Lichtmikroskop optisch unterschieden werden, wenn ihr Abstand d mehr als etwa die halbe Wellenlänge des Lichts beträgt. Bei kleinerem Abstand verschmelzen die Beugungsscheiben beider Punkte und lassen sich nicht mehr trennen. Für eine einzelne Beugungsscheibe kann das Zentrum mit einem Gauss-Fit jedoch sehr genau bestimmt werden.

Die stochastische Lokalistationsmikroskopie verwendet spezielle Fluorophore, mit denen die zu beobachtenden Strukturen in einer biologischen Probe markiert sind. Bei Bestrahlung mit Laserlicht wechseln diese Fluorophore selbstständig zwischen zwei optischen Zuständen und blinken unter dem Mikroskop. Der dunkle Zustand hält länger an als der helle, und es leuchten immer nur wenige Fluorophore zu einem Zeitpunkt, die gut unterschieden werden können. Es wird ein Film der Probe aufgenommen und die Beugungsscheiben der Flourophore werden anschließend gefittet um Position, Breite, Intensität und Fehler bestimmt. Zuletzt werden die Positionen in ein neues Bild mit einer Auflösung von bis zu 20 nm eingetragen (Abb. 1b).

Die Auswerte-Software war in der Programmiersprache Matlab geschrieben, die mehrere Stunden für die Auswertung einer nur fünfminütigen Aufnahme mit einer Auflösung von 256 $\times$ 256 Pixeln benötigte. Für die Portierung auf FPGAs wurden zuerst die einzelnen Schritte des Algorithmus untersucht. Die meiste Rechenzeit wurde für den Fit der Beugungsscheiben mit einer Gausschen Verteilung aufgewendet, der iterativ die Fehlerquadrate minimierte. Für die Hardwareimplementierung wurde der Fit auf Maximum Likelihood umgestellt. Zusätzlich wurde eine Verarbeitungsstufe eingeführt, die das Hintergrunleuchten der Fluorophore misst und abzieht. Ohne Hintergund konnte der Maximum-Likelihood-Fit zu einer Schwerpunktsberechnung vereinfacht werden, die auf einem FPGA sehr effizient berechnet werden kann.

Der neue Algorithmus wurde in Matlab und C implementiert, mit einer Reihe von Monte-Carlo-Simulationen auf Genauigkeit getestet und anschließend auf den FPGA portiert. Das Hardware-Design besteht aus zwei Pipelines, von denen die erste jedes Pixel der Filmaufnahme einliest,



(a) Konventionelle Weitfeldmikroskopie

(b) Lokalisationsmikroskopie

(c) Aufbau der Pipelines im FPGA.

Abbildung 1: Lokalisationsmikroskopie: Auflösungsverbesserung und Hardware-Design für die Filmanalyse. [4, 10]

den Hintergrund unterdrückt und nach Signalen sucht, die sich deutlich vom Hintergrundrauschen abheben. Die gefunden Signale werden mit ihrer Umgebung an die zweite Pipeline übergeben, die von der ersten mit einem FIFO-Puffer entkoppelt ist. In der zweiten Pipeline werden falls nötig eng beieinander liegende Beugungsscheiben getrennt, das Rauschen unterdrückt und zuletzt Position, Breite, Intensität und Fehler bestimmt (Abb. 1c).

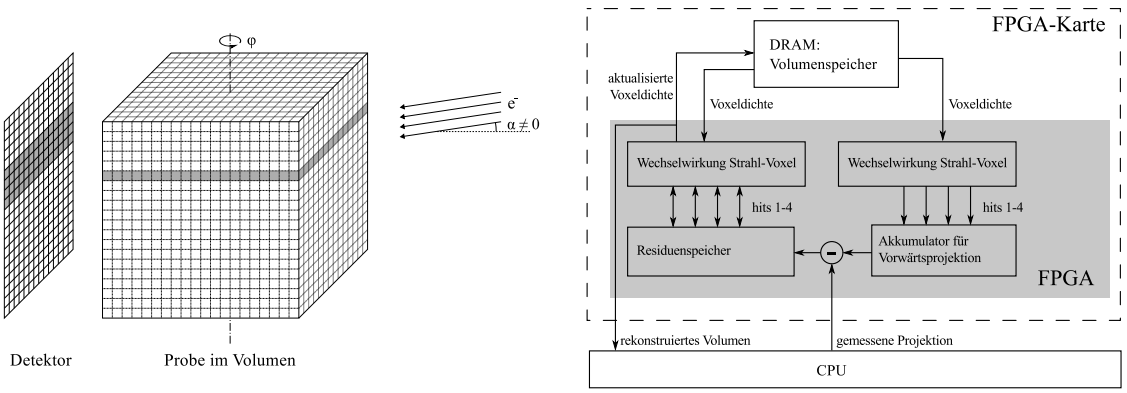
Elektronentomografie

Die zweite Anwendung rekonstruiert die dreidimensionale Dichteverteilung einer Probe, die von verschiedenen Seiten mit einem Elektronenmikroskop aufgenommen wurde (Abb. 2a). Sie gehört der Computertomographie an, die in der Medizin und Biologie weite Verbreitung gefunden hat. Der verwendete Algorithmus (SART) [5] benötigt etwa 50 Einzelbilder für eine zuverlässige Rekonstruktion und beginnt mit einem leeren Volumen im Speicher. In einer Schleife wird dieses entlang der Elektronenstrahlen auf einen virtuellen Detektor projiziert und mit dem gemessenen Bild verglichen. Der Unterschied wird anschließend wieder in das Volumen zurückprojiziert, indem die Differenz für jedes Detektorpixel gleichmäßig im Volumen entlang der Strahlrichtung verteilt wird. Vorwärts- und Rückprojektion werden solange wiederholt, bis alle Bilder aus dem Elektronenmikroskop verarbeitet sind.

Die zu beschleunigende Anwendung lag in der Sprache CUDA für Nvidia-Grafikkarten vor [7]. Diese folgte den Elektronenstrahlen für beide Projektionen und erzeugte ein pseudo-zufälliges Zugriffsmuster auf die Voxel im Speicher. Für die Implementierung auf einem FPGA wurde der Algorithmus auf ein lineares Zugriffsmuster umgestellt, bei dem die Voxel Zeile für Zeile und Ebene für Ebene eingelesen werden. Die Strahlen, die ein Voxel treffen, werden anschließend bestimmt, um die Vorwärtsprojektion aufzubauen, beziehungsweise um die Dichte des Voxels bei der Rückprojektion anzupassen. Die Aufnahmen werden während der Berechnung von der CPU an den FPGA auf der Einsteckkarte gesendet. Nach der letzten Rückprojektion wird das rekonstruierte Volumen ausgelesen.

Der lineare Zugriff ermöglicht eine höhere Speicherbandbreite und eine einfache Adressierung im Burst-Modus. Zu Beginn der Arbeit waren nur Teile der Rückprojektion auf FPGAs für die 2D-Tomographie implementiert worden. Abweichungen in der Strahlführung des Elektronenmikroskops verhinderten zusätzlich die Reduzierung der 3D-Tomographie auf 2D-Tomographie für jede Detektorzeile.

Das gewählte Hardware-Design besteht aus zwei Pipelines, von denen die erste die Vorwärtspro-



(a) Abbildung der Probe

(b) Aufbau der Pipelines im FPGA.

Abbildung 2: Elektronentomografie: Abbildung der Probe auf einen Pixeldetektor und Hardware-Design für die Volumenrekonstruktion. [11]

jektion und die zweite die Rückprojektion berechnet (Abb. 2b). Die Voxel des zu rekonstruierenden Volumens werden im DRAM-Speicher auf der FPGA-Karte gehalten. Für die Vorwärtsprojektion wird das Volumen linear eingelesen und die Schatten der einzelnen Voxel zu einer Projektion akkumuliert. Diese wird dann in der zweiten Pipeline mit der gemessenen Projektion verglichen und die Dichteverteilung für jedes Voxel im DRAM während der Rückprojektion angepasst. Vorwärts- und Rückprojektion laufen parallel. Je ungenauer das Elektronenmikroskop kalibriert ist, um so länger muss die Rückprojektion auf die erste Zeile der Vorwärtsprojektion warten, weil dann eine Ebene im Volumen der Probe auf mehr als eine Detektorzeile abgebildet wird.

Ergebnisse

Die Algorithmen beider Anwendungen wurden auf FPGAs mit den Entwicklungswerkzeugen von Maxeler Technologies und Xilinx implementiert. Die FPGAs befinden sich auf einer Einsteckkarte und sind per PCI express mit der CPU verbunden. Während die Berechnung läuft erfolgt der Datenaustausch parallel, und der maximale Durchsatz ergibt sich aus der Taktfrequenz der Pipelines.

Das Hardware-Design für die Lokalisationsmikroskopie belegte weniger als 1/3 der Ressourcen auf einem Virtex-5 LX330T FPGA bei voller Genauigkeit. Die Genauigkeit der Implementierung wurde unabhängig untersucht [8] und bewegte sich im gleichen Rahmen wie die übrigen Softwarelösungen. Der Beschleunigungsfaktor betrug insgesamt 18 500 auf einem Virtex-6 SX475T, wobei ein Faktor von 100 aus der Neukonzeption der ursprüngliche Software stammte und ein Faktor von 185 auf den FPGA zurückzuführen ist. Damit können 167 Megapixel/s bei einer Taktfrequenz von 200 MHz verarbeitet werden, 2,5 mal schneller als die MaLiang-Methode für Grafikkarten [9, 10].

Die Portierung für die Elektronentomografie wurde auf einem Virtex-6 SX475T durchgeführt. Die Größe des FPGAs ermöglichte die Verarbeitung von drei Voxeln pro Takt, wobei die Signalprozessoren (DSPs) die Designgröße beschränkten. Beide Pipelines bestehen aus etwa 100 Stufen, sind mit 170 MHz getaktet und erreichen einen Durchsatz von 510 MVoxel/s. Im Vergleich mit einer Nvidia Tesla C1060 Grafikkarte wurde die Anwendung um einen Faktor fünf beschleunigt. [11]

Diskussion

Lokalisationsmikroskopie und Elektronentomografie konnten erfolgreich mit Datenflussrechnen auf FPGAs beschleunigt werden und sogar die jeweilige Grafikkartenimplemtierung überholen. Dank der weitgehend automatisierten Abbildung des Datenflussgraphen auf FPGAs bestand der Kern der Arbeit in der Übersetzung der Algorithmen in eine Form, die die Stärken des FPGAs effizient ausnutzt. Dazu gehört ein hoher Pipeline-Parallelismus, ressourceneffiziente Berechnungen auf Festkommazahlen mit frei wählbarer Kodierung sowie direkte Kontrolle über jeden Speicherzugriff.

Für beide Anwendungen begann die Portierung mit einer Analyse der kleinsten Dateneinheit, dem Pixel beziehungsweise dem Voxel, und dem Datenfluss innerhalb des algorithmischen Kerns. Darauf aufbauend konnte der Lösungsraum untersucht und eine Implementierung gefunden werden, die effizient auf die Architektur eines FPGAs abgebildet werden kann. Dieser Prozess wird in der Arbeit beschrieben und kann auch zukünftig verwendet werden, um für andere Anwendungen aus dem Bereich der hochperformanten Bildverarbeitung eine beschleunigte Ausführung zu erzielen.

Die verwendete Bibliothek für die Abbildung des Datenflussgraphs auf Hardware machte diese Arbeit überhaupt erst im gegebenen Zeitraum möglich. Im Gegensatz zu Programmiersprachen, die versuchen, eine imperative Beschreibung direkt auf Hardware abzubilden, bleibt eine Datenflussbeschreibungssprache nah genug an der Hardware, um den zu erwartenden Ressourcenverbrauch für den Programmierer nachvollziehbar zu halten und stellt nach Meinung des Autors einen guten Kompromiss zwischen niedrigeren Sprachen mit geringerer Produktivität und imperativen Hochsprachen mit einer zu großen semantischen Lücke dar.

Acknowledgements

First, I would like to thank my supervisor Prof. Dr. Udo Kebschull for giving me the opportunity to work in his research group. This thesis would not have been possible without his support. I appreciate the academic freedom I received while pursuing my research, the open communication of ideas and visions, and the ever reliable and fair treatment as an employee.

Further, I would like to thank my colleagues Sebastian Manz, Stefan Böttger, Heiko Engel, Thomas Janson, Andrei Oancea, Christian Stüllein and Jano Gebelein for hints and discussions about my research, as well as for proof-reading the thesis. Additional thanks go to Jano Gebelein for taking over most administrative tasks so I could focus on the research. Manfred Kirchgessner wrote the implementation for localization microscope on the SiliconSoftware hardware, I appreciated his contribution very much. Last, but not least I want to thank my family. Without the support of my parents I would likely not be where I am now.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Computing has seen a major shift on all levels since the beginning of the century. Before, software applications were expected to scale, but it was also given as granted that single-core processor speeds would continue to increase exponentially. Software that did miss a performance target by a constant factor would meet it soon with the next generation of hardware. This almost effort-less acceleration came to a halt when clock frequencies of CPUs could not be increased any more due to uncontrollable heat dissipation above about 4 GHz. Since then, acceleration needed to be sustained by a different method, and computer programming increasingly focused on parallel computing. Here, the computer hardware executes multiple instructions at the same time, and the total throughput is increased by the parallel factor in the best case. CPUs have gained the ability for parallel execution with multi-core processors, and vector instructions were re-discovered to speed up execution on the instruction level. Modern CPU architectures can still be programmed in a sequential manner, but the most compute-intense parts are now routinely converted to multi-threaded execution or vector instructions. All major computer languages have shifted to offer support for parallel programming, by either providing library support or sometimes through language extensions.

A massive parallelism suitable for practical use was discovered in about 2005 when graphics cards proved to be capable as application accelerators outside of the realm of computer graphics. On these cards, hundreds of compute cores that resemble an array of simple processors were able to accelerate certain computing problems by more then two magnitudes compared to CPUs. Graphics cards quickly gained popularity in high performance computing, despite their intrinsic execution model that restricts acceleration to massively parallel programs where each thread group performs the same step at a time.

An even finer grained execution model can be found in Field Programmable Gate Arrays (FPGAs), the hardware used in this thesis. FPGAs can be programmed below the level of the processing unit. A large array of very basic hardware elements can be configured to emulate every other digital computer chip very close to its hardware level, given the FPGA does not run out of resources. The increasing size of FPGAs has moved their purpose from small logic applications towards accelerators for fully-featured applications in high-performance computing. The tremendous flexibility of their internal and peripheral architecture promises to accelerate all sorts of computing problems with less restrictions that graphics cards impose on the algorithms of the application.

FPGAs can be found in areas where high computing performance or low latency are such a hard requirement that custom computing becomes feasible, but application specific integrated circuits (ASICs) remain too expensive due to low purchasing volumes. They are used in experiments for

high-energy physics were trigger latencies must be low, in electronic high frequency trading or seismic exploration. The acceleration gained in these areas motivates the extension of the scope of application towards other high-performance applications in science, such as image processing and reconstruction.

In this thesis the usage of FPGAs as application accelerators is researched for the domain of biomedical image processing and reconstruction. Here, many compute-intense problems can be found that have not yet attracted the interest of commercial vendors in particular, and where the increased size of modern FPGAs and new programming tools have only recently made it feasible to port them from CPUs or graphics cards towards FPGAs.

Shortened computing times are crucial for timely diagnostics. Data analysis that is performed in real time with data acquisition moves diagnostics from a work-flow interruption towards an interactive tool. Samples could be examined a second time before they decay, diagnosis would become faster and laboratory examination has the potential to become cheaper and therefore promote its usage.

1.2 Overview

1.2.1 The idea

An algorithm that was written for high-performance computing on a CPU or graphics card reflects many optimizations that were chosen on purpose during performance evaluation, but also implicitly based on past experience of the programmer. FPGAs can emulate CPUs, but only slower by at least one order of magnitude. For efficient execution that achieves an acceleration over the previous hardware architecture, an FPGAs application must be dissected and re-implemented such that the unique architecture is employed not for emulation, but for custom designed computing. The most efficient implementation for FPGA computing is usually the pipeline, a system of hardware operators and wired connections in the FPGA where every operator performs one operation in every time step. Laid out with a length of hundreds of pipeline stages, a result is produced at its output with every cycle in the best case, and since every operator executes all the time, the design is optimal for algorithms that can be molded into such a structure.

To reduce the cognitive load of not only understanding the algorithm of the program to port from the source code, but also the description of the hardware, it was chosen to use a so-called dataflow compiler from Maxeler Technologies for pipeline description. FPGA programming is traditionally performed in languages that describe the behavior on a very low level, and hardware programming is observed to consume much more time for the same functionality than software programming. Because the features aimed for data manipulation at the lowest level of traditional hardware programming languages were not needed, a higher level compiler for pipeline description could be used to bring down development time and target the acceleration of two application within this Ph.D. thesis.

1.2.2 Aim of the thesis

Two applications were suffering from slow execution times and were selected as acceleration opportunities to develop a guide about how to port software written in an imperative language towards a dataflow description. From the dataflow description, they can then be automatically translated to a system of pipelines on an FPGA. The observations and insights gained during the porting process form the center of this thesis. Most of the procedures that help to bridge the gap have been used before, but were not collected and written down in a concise manner. The second aim was to actually accelerate the applications and provide value for its users in terms of run-time

and data throughput. For this, other factors such as maintaining the accuracy of the results and usability became important.

The example applications cover common algorithms which can be found in most related applications in image processing. The first application is used for analyzing raw data from localization microscopy. By marking important molecules in (live) biological structures and switching them stochastically between a bright and a dark state under a conventional light microscope, localization microscopy can separate the blurry optical signals in the time domain in a second step on a computer and eventually produce an image with a resolution improved by one order of magnitude. For this application a CPU implementation was available and needed to be ported in a combination of algorithmic and hardware design.

The second application reconstructs 3D images from 2D projections for electron tomography, a special case of computed tomography. Due to imprecisions in the alignment of the electron microscope, the problem cannot be reduced to a set of 2D reconstructions, and hence the algorithms developed for 2D tomography are unsuitable. Electron tomography provides a much higher resolution, but requires the sample to be dead. It complements optical light microscopy on the cellular level and generates 3D images. The source code was available in the CUDA language for graphics cards.

1.2.3 Defended statements

With this thesis the usage and practicability of FPGAs as application accelerators for biomedical image processing and reconstruction was researched. In its summary, the following statements can be defended.

1. **Effective portability from an imperative program to a dataflow description**
   It is shown that concepts from applications used for image processing and reconstruction can be ported to a dataflow description that is suitable for FPGAs. In a continuation and as an advancement to previous work, compute intense application kernels can now be ported to FPGAs that were excluded before by the limited size of previous FPGAs and programming tools.

2. **Efficient development with high-level languages**
   Custom hardware is still mostly developed with low-level languages, either VHDL or Verilog. It is shown that a description on a higher level with dataflow programming as the fundamental programming paradigm increases development efficiency and enables complex designs that would not be possible with the legacy programming tools within the given time.

3. **Acceleration of both sample applications by a significant factor**
   An acceleration was achieved for both applications that were implemented. For image analysis in localization microscopy, a total acceleration of a factor of 18 500 was achieved, with about equal parts owed to algorithmic redesign and hardware acceleration. The application for electron tomography was accelerated by a factor of five compared to an implementation on graphics cards.

1.3 Outline

In the first chapter, the basic principles of FPGAs are introduced and their usage for application acceleration. Dataflow computing is presented as an efficient design method for high-performance computing with a focus on image processing. The following chapter gives an overview about the state of the art of the still new concept of exploiting reconfigurable computing for data processing,

and compares it with graphics cards and multi-core accelerators. As the hardware continues to get more capable with every generation, the tools to describe problems for application accelerators must keep pace and are covered in the second part of chapter 3.

The translation process from imperative languages, such as C, requires are re-thinking of the algorithms of a program. A general top-down approach for an efficient porting process is presented in chapter 4, "Acceleration of Imperative Code with Dataflow Computing". After the computational bottlenecks of a program are understood, the chapter describes how to pipeline a previously sequential control flow. The chapter concludes with hardware details that determine the efficiency of logic operations and memory access.

This top-down approach is then used to accelerate two example applications in chapter 5. Localization microscopy and electron tomography both require high-performance processing of image data and are suited to benchmark the FPGA hardware as well as the approach of combining algorithmic with hardware design. Both applications were successfully accelerated. Localization microscopy achieved an acceleration factor of 185 compared to a single core on a CPU after the program was improved by a factor of 100 in software alone. Electron tomography was accelerated by a factor of five compared to a Nvidia Tesla C1060 graphics card. These results are presented and discussed in depth in chapter 6.

The thesis is completed with the conclusion in chapter 7. Dataflow computing could be confirmed to be well-suited for algorithms in image processing that were previously too big or too complicated for previous FPGA generations and the constraints of legacy programming tools.

1.4 Publications

1. **Frederik Grüll**, Manfred Kirchgessner, Udo Kebschull, "Analytic Solution for Image Analysis in Localization Microscopy With Full Accuracy", *DPG Dresden 2011: Biological Physics Division*, Dresden, Germany, 2011

2. **Frederik Grüll**, Manfred Kirchgessner, Rainer Kaufmann, Michael Hausmann, Udo Kebschull, "Accelerating Image Analysis For Localization Microscopy With FPGAs", *International Conference on Field Programmable Logic and Applications 2011, Proceedings of*, Chania, Greece, 2011

3. Rainer Kaufmann, Jörg Piontek, **Frederik Grüll**, Manfred Kirchgessner, Jan Rossa, Hartwig Wolburg, Ingolf E. Blasig, Christoph Cremer, "Visualization and Quantitative Analysis of Reconstituted Tight Junctions Using Localization Microscopy", *PLoS ONE*, vol. 7, Public Library of Science, 2012

4. **Frederik Grüll**, Michael Kunz, Michael Hausmann, Udo Kebschull, "An Implementation of 3D Electron Tomography on FPGAs", *International Conference on ReConFigurable Computing and FPGAs, Proceedings of*, Cancun, Mexico, 2012

5. **Frederik Grüll**, Udo Kebschull, "Biomedical Image Processing and Reconstruction with Dataflow Computing on FPGAs", *International Conference on Field Programmable Logic and Applications 2014, Proceedings of*, Munich, Germany, 2014, to be published

Chapter 2

Basic Principles

In this chapter we will cover the basic principles of reconfigurable hardware and how it can be used to accelerate applications. Dataflow computing is introduced as one way to program reconfigurable hardware in an efficient way that is opposed to the von-Neumann model of computation. Further, we will introduce a set of basic operations for image processing and how they relate to dataflow computing. The basic principles of the accelerated applications, namely localization microscopy and electron tomography, are covered in chapter 5.

2.1 Field Programmable Gate Arrays

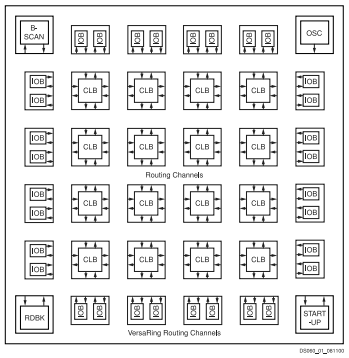
Custom hardware is tailored to a specific problem and used where general purpose hardware would not meet the required computing performance, such as in sound processors or graphics cards. Other use cases are found in domains where general purpose hardware cannot compete in terms of input bandwidth, latency or cooling requirements. Traditionally, these computing solutions were implemented as fully customized hardware known as Application-Specific Integrated Circuits (ASICs).

Since the mid-1980s, Field Programmable Gate Arrays (FPGAs) have been established as a third alternative that is located between general-purpose and application-specific integrated circuits (ICs) in terms of flexibility. FPGAs can be configured to change their functionality on the hardware level in a reversible process. The user can therefore choose from many different ASIC designs to be mapped on an FPGA model, as long as the resources of the FPGA are not depleted. For small series, FPGAs offer a way to avoid the expensive up-front cost required for the photo-lithographic masks of an ASIC production line and still provide the speed advantage of custom hardware. A second advantage over ASICs is given by the ability to update an FPGA with a new design to further improve the performance or to fix bugs later in the product life cycle, without changing the actual IC.

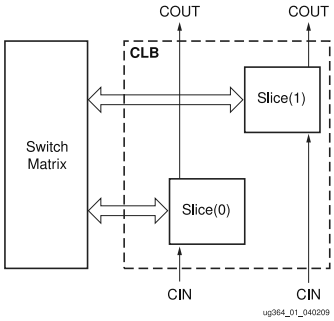
Since their invention FPGAs have started to cover multiple use cases. Starting from ICs routinely used for glue logic, they have become a replacement for or emulator of digital ASICs, and have gained traction as application accelerators in high-performance computing. To better understand FPGAs we will first cover the internal setup and how it can be programmed, and will then describe their usage as application accelerators.

2.1.1 Integrated Circuits

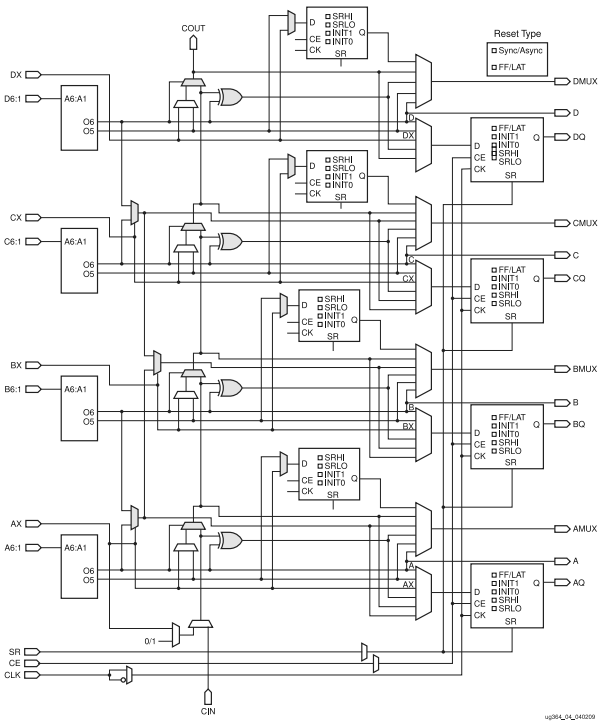
The first FPGA was introduced in 1985 by Xilinx, Inc. [12]. The structure of a modern FPGA still follows the same principle: a large array of configurable logic blocks (CLB) is embedded into a wire network that routes the electrical signals (Fig. 2.1). The binary operations of an FPGA are carried



(a) FPGA (simplified)



(b) Configurable Logic Block (CLB)



(c) logic slice

Figure 2.1: Internal structure of an FPGA. The figures show the top three levels of abstraction of a Xilinx Virtex-6 FPGA as used in this thesis. Images: Xilinx [13, 14]

out inside of its CLBs, and the data consumed and produced by each of them is distributed through the wire network. This way, any logic function can be mapped by configuring the connections and the CLBs. The array of CLBs is surrounded by auxiliary units, such as input/output buffers (IOB), clock management (OSC) and other periphery needed for the configuration process and for communication with the outside world (Fig. 2.1a). For connections to a PCIe bus or a network interface special high-speed transceivers are employed. Additionally, some FPGAs also feature entire CPU cores for heterogeneous computing within the FPGA.

Configurable Logic Blocks

The interconnects between the CLBs are configured in the switch matrix. Both CLBs and the switch matrix contain special Static Random Access Memory (SRAM) storage cells that are written during start-up and determine the function of the FPGA. This information is called the configuration of the FPGA. It is provided through its configuration interface from an external device, or obtained from non-volatile storage that sits next to the chip.

Each CLB is divided into two slices for Xilinx FPGAs (Fig. 2.1b). They are connected to the switch matrices and to two adjacent CLBS for fast carry chains in adders and similar logic. An FPGA may contain several types of slices [13]. The internal structure of a logic slice is depicted in Fig. 2.1c for a Xilinx Virtex-6 FPGA. It contains four lookup-tables (LUTs) close to its inputs on the left, a set of multiplexers and eight one-bit storage elements that may be used as flip-flops (FFs).

The SRAM content of each LUTs defines its truth table and is set during reconfiguration as well. A LUT can then be used to look up the result of any Boolean function with the same number of arguments at run-time. It substitutes the combinatorial logic between registers of an ASIC. The output of a LUT can either be directed to the carry chain by the multiplexers, to a FF or out of the

CLB to the switch matrix. The state of the multiplexers is also defined by the configuration, as well as the initial state of the FFs and their exact mode of operation.

All FFs have a clock-enable (CE) input. If it is set to low, the FF will not store a new value at the next clock cycles and keep the old value instead. This is a convenient feature when building pipelines that need to be paused repeatedly. All mutable state is contained in the FFs and a low CE signal preserves the state until the pipeline will need to advance again.

Combined, LUTs and FFs provide combinatorial logic and storage within a well-defined unit. A modern FPGA contains up to hundreds of thousands of these slices. With just the design explained above, the logic of any digital ASIC could be mapped to an FPGA, given its resources do not get depleted. Because of the reconfiguration overhead, an FPGA needs much more transistors to build a logic primitive than an ASIC. With only LUTs and FFs, about 35 times the silicon area is required than an equivalent ASICs on average. In practice however, the ratio can be brought down to under five times the silicon area with the support of special logic cores for common design units, e. g. multipliers and on-chip RAM. [15]

Block RAM

Without dedicated FPGA resources for RAM, a design would require one FF per bit to be stored, and additional interface logic would be needed for read and write access through an address and data bus. To make better use of the silicon area, some slices contain LUTs that can be configured and combined to form distributed RAM. To store even more information with a low resource footprint, FPGAs embed special Block RAM cores (BRAM). For the Virtex-6 series, an FPGA may have multiple thousands of BRAM cores with a size of 4.5 KB each [16].

Besides of their combined ability to store megabytes of data directly on the FPGA, BRAMs can be accessed through two ports independently. That implies that in the same clock cycle two addresses can be read or written. This is of particular use to build a First In, First Out (FIFO) data buffer, where a write port inserts data at the front and a read ports removes is at the end. This use case is common enough that some FPGAs support it directly with dedicated hardware support and offer signals to indicate when the FIFO is empty, full, almost empty or almost full. A FIFO can then decouple different parts of an FPGA that are in a consumer-producer relationship and buffer unsteady data rates. The almost empty and almost full signals provide a convenient way to notify the producer or the consumer in advance that it will need to block soon. For pipelining, a FIFO provides the means to connect two pipelines that cannot operate in lockstep, and it provides the feedback to pause any of when a stall occurs.

Digital Signal Processors

Adders and subtracters can be implemented with a low resource footprint on FPGAs with a dedicated carry lane. The hardware is then able to sum or subtract two numbers with fixed-point encoding per clock cycle. For a fully-pipelined multiplication with N bits for each input, the resource usage grows with order $\mathcal{O}\left(N^2\right)$. Since multiplication is a very common use case, some FPGAs have special Digital Signal Processing (DSP) elements available for area-efficient and fast multiplications. On the Virtex-6 series used here, a DSP slice consist of a 25×18bit multiplier followed by an adder for fused multiply-add with two's complement integers. Each clock cycle computes the term $x = a \times b + c$ with correct rounding for the whole term. Common operations, such as matrix multiplications or convolutions, benefit from special DSP slices and the silicon footprint of the FPGA configuration is reduced. Pre-adders are included to allow DSP slices to be cascaded for wider operators. Bit pattern recognition and feedback paths for accumulators further extend their use case to Fast Fourier Transforms, floating point operations and filter chains. [16]

In the realm of image processing DSPs are of particular interest for convolutions. A convolution on a rasterized computer image applies a filter kernel with a same-sized region in the image surrounding the current pixel, carried out repeatedly around every pixel of the image. It contains multiplications and additions and can make use of the fused multiply-add support. After the image has been processed with convolutions, an optional feature extraction stage will often require fixed-point or floating-point multiplications when the data is reduced to a few scalar properties, making further use of DSP resources. When the DSP resources are depleted, the remaining logic can still be used as a substitute, but may slow down the maximum achievable clock frequency, and hence reduce the throughput.

FPGAs may contain further hard-wired cores to further increase the package density and customize them for their intended use. These cores can still be configured, but most of the logic implementation is fixed. A common core is a high-speed transceiver that connects an FPGA with a computer network or the PCIe bus of a host computer. Some product lines also feature entire CPUs for heterogeneous computing on the same silicon chip.

2.1.2 Low-level Hardware Description Languages

The first FPGA, the Xilinx XC2064, consisted of an array of only 8×8 CLBs. At this size the function of each CLB and the routing between the slices could still be performed on a sheet of paper. When ASICS and FPGAs grew more complex in the 1980s, it became apparent that hardware should be described with computer languages to increase productivity. At the time, software was already written at a higher level and automatically translated to machine code. An equivalent tool flow for hardware was created for FPGA programming.

VHDL and Verilog

Two of the hardware description languages that were developed are still in wide use today: VHDL and Verilog. VHDL (VHSIC Hardware Description Language) is based on the Ada language and was initially created as a means of documentation for digital ASIC designs. It became standardized in the year 1988 as an IEEE standard [17] and soon started to be used to simulate the behavior of the described ICs, too.

An example of a 16-bit register with two write ports described in VHDL can be seen in Listing 2.1. The code shown uses a subset of VHDL that can be transformed in a process known as logic synthesis into an actual configuration of an FPGA. The port map at the beginning declares which input and output signals the register uses. The process description below defines the functionality. The register may be written from one of the inputs at the rising edge of the clock signal (`clk`) by setting the write-enable (`we`) signal to high. The input is chosen by the select signal (`sel`) from either `val_a` or `val_b`. Otherwise, the value is kept. Entities like this one can instantiated multiple times and connected by code to form a more complex design.

Verilog serves the same purpose, but is based on the syntax of C. Both hardware description languages operate on the register transfer level (RTL), meaning they are used to describe the flow of signals through combinatoric logic between registers that store the values. Both languages contain higher constructs such as loops with a variable number of iterations, but these can only be used for simulation and are not synthesizeable to hardware. Because only a small subset is recognized (that also depends on the synthesis tool chain), a developer must not only know Verilog or VHDL as a language, but also the subtleties of its usage to eventually create working hardware. For efficient bitfile generation, they should also have a good knowledge about the internal structure of an FPGA. Otherwise, the configurable LUTs and FFs may not be used in the most efficient way. A description close to the hardware safes up to 50% of the resources when implementing a register

alone, which is one of the most basic structures in a hardware design for digital logic [18, 19]. Hardware description languages that hide these details from the developer have been created since then, but VHDL and Verilog still remain the backbone of FPGA development.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity reg16 is
    Port ( clk     : in   STD_LOGIC;
           sel     : in   STD_LOGIC;
           we      : in   STD_LOGIC;
           reset   : in   STD_LOGIC;
           val_a   : in   STD_LOGIC_VECTOR(15 downto 0);
           val_b   : in   STD_LOGIC_VECTOR(15 downto 0);
           val_out : out  STD_LOGIC_VECTOR(15 downto 0));
end reg16;

architecture Behavioral of reg16 is

signal val : std_logic_vector(15 downto 0);

begin

process (clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            val <= (others => '0');
        elsif(we = '1') then
            if(sel = '0') then
                val <= val_a;
            else
                val <= val_b;
            end if;
        else
            val <= val;
        end if;
    end if;

end process;

val_out <= val;

end Behavioral;
```

Listing 2.1: A 16-bit register in VHDL that can be written from two inputs. When the clock signal rises and depending on the input signals select (`sel`), write enable (`we`) and `reset`, the value in the register is either taken from one of the two inputs `val_a` or `val_b`, kept or reset. [20]

FPGA design flow

After a design was written in a hardware description language on the register transfer level, the logic must be translated into a configuration bitfile for the FPGA. This is done automatically by the design-flow tools of the FPGA vendor. The process encompasses three steps, each building on top of the previous [19].

1. **Logic synthesis:** The abstract form of the design, typically written in VHDL or Verilog, is parsed and special language pattern for state machines, multiplexers or memory are recognized. Other optimizations check whether resources can be shared and will need to be implemented only once on the FPGA. The result is a description of the design as a set of logic elements and their connections. This is known as "net list" and describes binary and arithmetic operations for the combinatorial logic, registers for state and logic cores that the developer specified directly in the source.

2. **Technology mapping:** The description from the previous step is transformed such that it matches the resources found on the target hardware. The combinatorial logic is mapped onto the LUTs and DSPs with their size taken into account, the registers are implemented as FFs, and the memory is built from the BRAM available or other resources. The result is specific for the targeted FPGA.

3. **Place and route:** Finally, the hardware primitives are placed on the FPGA and connected with routes through switch boxes. This steps will try to keep the wiring fast enough to reach the desired clock frequency by moving the logic on the chip until all critical paths are satisfied. The time needed for this step increases with the size of FPGA, the overall resource usage and the desired clock frequency.

At the end of the tool chain, given all timing constraints could be met, a configuration bitfile will be produced that computes the source description on an FPGA for a certain voltage and temperature range. Contrary to software compilation, the step for placing and routing contains indeterministic searches for an optimum. For modern FPGAs with a high resource usage the process can typically take from hours to multiple days.

2.1.3 FPGAs as Application Accelerators

We have seen that FPGAs require at least five times the silicon area for the same logic non-reconfigurable ICs. They also run at a lower clock frequency, typically with an order of magnitude in difference. Still, FPGAs have been shown to successfully accelerate programs that were previously executed on a CPU despite the initial penalty. The difference can be explained with the design constraints of general purpose hardware. A CPU must execute general programs encoded in machine language, whereas the configuration of an FPGA can be customized to the wanted algorithm.

Fig. 2.2 gives the floor plan of a core in the AMD Jaguar CPU, one of the few recent x86 CPUs were the area of its internal function units is available to the public domain. It depicts the size needed for auxiliary units that do not directly contribute to the computation and can mostly be omitted in custom hardware. About half the chip area of the CPU contains caches, buffers and read-only memory, identified by their regular structure. Caches are usually smaller on FPGAs, because information about the memory access pattern is available in advance. The Ucode ROM is needed to translate the machine code into micro-instructions to be executed by the other parts of the CPU. A machine language for general purpose computing is not needed on an FPGA that computes one algorithm only, as it can be directly executed by the hardware.

The caches and buffers keep data and instructions close to the arithmetic and logic function units in a CPU. These are required to hide the latency of the working memory that compromises many clock cycles, during which the CPU core would otherwise be waiting and idle. The data cache keeps parts of the RAM that are frequently accessed by the program, and the instruction cache makes the instructions of the program itself available for fast execution. The re-order buffer (ROB) modifies the order of parts of the program to avoid access conflict on the same storage location that would slow down execution. Branch prediction finally applies heuristic to guess which path a program will take next to prepare the right commands for execution in the processor pipeline.

The core areas for integer calculations and floating point (FP) logic then occupy only a small small fraction of the floor plan. Most of the silicon area is dedicated to auxiliary units that accelerate data access for general-purpose computing. A significant part of the chip must also be powered off to prevent overheating. About 21% of a CPU at a 22 nm silicon process must stay switched off at a time because of the much higher clock frequency and tighter integration [21, 22]. For custom computing on reconfigurable hardware most auxiliary units can be omitted or at least greatly

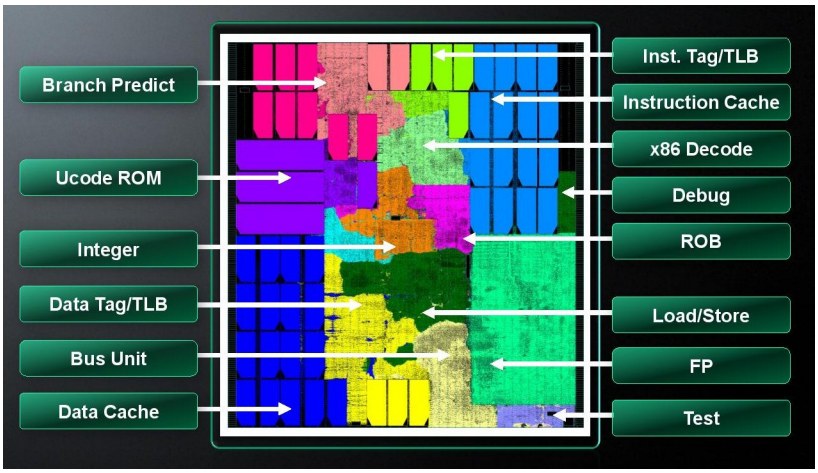


Figure 2.2: Floor plan of an AMD Jaguar core. Caches, branch prediction and the re-order buffer (ROB) are dedicated units that are included to keep the pipeline running. Image: Advanced Micro Devices [23]

reduced, and in combination with the lower clock FPGAs are able to beat CPUs in terms of speed, space and electric energy for thoroughly ported applications.

Pipelining

The instructions of a program are executed at the center of an x86 CPU, surrounded by the caches and buffers. All but the most basic CPUs split execution of an instruction into sequential steps and perform every step in parallel to maximize throughput. The most basic pipeline for computers with a Reduced Instruction Set (RISC) consists of five stages. First, the instruction is fetched from the instruction cache. It is then decoded into micro-operations in the second step and actually executed in the third one. If memory needs to be accessed, it is read or written in the forth step. Finally, the result of the instruction is written into a CPU register in the last step [24]. Modern CPUs have long pipelines with dozens of short stages to achieve a high clock frequency.

If the program has a linear structure and fits into the caches, the pipeline is filled all the time during execution and the clock frequency of the CPU correlates strongly with the throughput of instructions. The latency of an instruction in the pipeline is a multiple of the cycle duration between two consecutive instructions. Pipelining allows a computer system to hide this latency and keep the throughput high. The instructions are not executed one at a time, but in parallel with an overlap as big as possible. For a perfectly running pipeline that produces one result per clock cycle, a pipeline increases the throughput by a factor equal to the number of pipeline stages.

For acceleration it is important to estimate the different latencies of common operations in a computer system. Some latency numbers for common operations can be seen in Fig. 2.3. They span nine orders of magnitude from a level-1 cache reference in a CPU to the (still barely noticeable) round-trip time of 100 ms on the IP network between the USA and Europe. The aim to mitigating latencies can be seen on every level of high-performance computer system: the CPU pipeline allows a frequency of instruction to be faster than the inverse of the latency of a single instruction. Branch predictions aims to keep the pipeline without empty bubbles. On a higher level, optimizations for longer latencies are built into the process scheduler of the operating system for concurrency. A process that waits for a locked mutex, disk storage or the network periphery will be de-scheduled and another process may continue execution on the same CPU.

Custom hardware does not have to support every random sequence of instructions. Instead, the program is known in advance, and the hardware is specially tailored. This allows the implementation

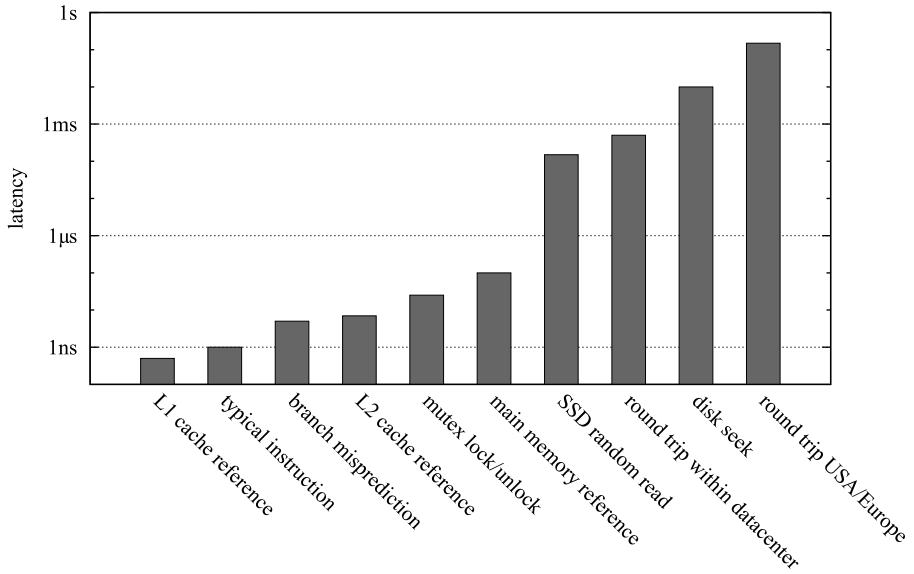


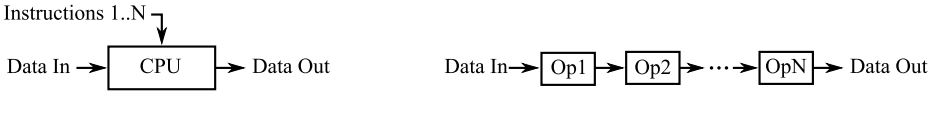
Figure 2.3: Latency numbers for common operations on a CPU system [25]. The logarithmic time axis spans nine orders of magnitude. With pipelining the throughput of operations can be greater than the inverse latency.

of other types of parallelism and pipelining. Instead of pipelining the instruction flow, we can pipeline the data flow of a linear program without branching (Fig. 2.4b). The stages that fetch and decode instructions in a CPU can be removed, since the instructions are fixed. The execution stage in the CPU is reduced to only perform the current instruction, and the next instruction is appended physically on the FPGA to consume the result of the previous one. The execution of the program changes from an execution of instructions in time towards an execution in space, where each instruction in the program occupies its own silicon area. It is therefore also called "dataflow computing" or computing in space. Transforming programs with branchings or complex loops requires extra effort and will be covered on its own in section 4.3 "Pipelining Imperative Control Flows" (p. 47).

Flynn's Taxonomy

Computers can compute in parallel, either on the instruction stream or the data stream. In 1970 Michael Flynn ordered the different kinds of parallelism in a concise taxonomy that has later been named after him [26]. It is still in use today to describe the parallel architecture of hardware.

- **Single instruction, single data (SISD)**: A processor applies one instruction on one data item at a time. This is the mental model a single-core machine operates when it runs through an imperative program. A SISD machine can be accelerated hidden from the programmer



(a) CPU (b) dataflow pipeline

Figure 2.4: CPU versus dataflow pipeline. In the CPU the execution of the instruction flow is pipelined. If the stream of instructions is data-independent and repeatedly applied, the data flow itself can be implemented in hardware as a pipeline on an FPGA.

by pipelining instruction execution and by re-ordering instructions that do not depend on each other. The length of the pipeline and hence the performance of an SISD machine is limited by data dependencies that occur when an instruction accesses data that was written by a previous one.

- **Single instruction, multiple data (SIMD)**: Instead of applying an instruction to only one data item, a SIMD machine applies the instruction to a vector of data items. This capability can now also be found in previously SISD-only architectures. The most widespread example is the x86 processor architecture after the introduction of the Streaming SIMD Extensions (SSE) [27]. The second well-known member of the SIMD family are graphics cards with thousands of compute units that all perform the same instruction in parallel.
- **Multiple instructions, single data (MISD)**: Here, multiple instructions are executed on a single data stream. A pipeline that processes a stream of data through multiple stages of operators can be seen as an MISD machine [28]. The number of (fixed) instructions processed per clock cycle rises linearly with the length of the pipeline.
- **Multiple instruction, multiple data (MIMD)**: An MIMD machine can be obtained by parallelizing either a SIMD or a MISD machine a second time. For the first, MIMD is implemented by running a program in multiple SIMD cores. A pipeline (MISD) on the other hand can be parallelized by instantiating multiple instances, where each pipeline processes its own data stream.

The concepts of SISD, SIMD and MIMD currently outnumber the MISD architecture by a big margin. This is likely due to the fact that the former three can be programmed with imperative, functional or logic computer languages with few architecture-dependent additions. The SIMD architecture requires a transformation into a dataflow description to build the pipeline, and the mapping process from imperative code to it is still not readily available.

Limits of Acceleration

Acceleration through frequency scaling has found its end in the last decade at frequencies of about 4 GHz. Since then the focus has shifted towards parallelization. FPGAs that house thousands of operators running at the same time offer a way to further increase parallelism compared to the hardware in wider use today. The achievable gain through parallelism for a given problem was examined by Gene Amdahl, and the resulting law is known as Amdahl's law [29].

For a program that can be run in parallel except for a strictly serial part with fraction s, Amdahl's law describes the total acceleration a that can be achieved by increasing parallelism to a factor of N. The speed-up will find its limit in the serial part; even an arbitrarily large factor of parallelism will not reduce its time. The total acceleration is calculated as in Eq. 2.1. For a sequential part of $s = 5\%$, the maximum acceleration is limited to a factor of $1/s = 20$. The relation is shown in Fig. 2.5.

$$a_{\text{Amdahl}} = \frac{1}{s + (1-s)/N} \tag{2.1}$$

The resulting insight may seem very restrictive at a first glance. It implies a barrier for the maximum speedup for a given algorithm which cannot be overcome with parallelism. However, Amdahl's law acts on the assumption that the workload is fixed. In high-performance computing, this is usually not the case and instead the user will rather allocate a certain time span for computing. With a time budget of e. g. one weekend or a fixed number of hours on a supercomputer, the

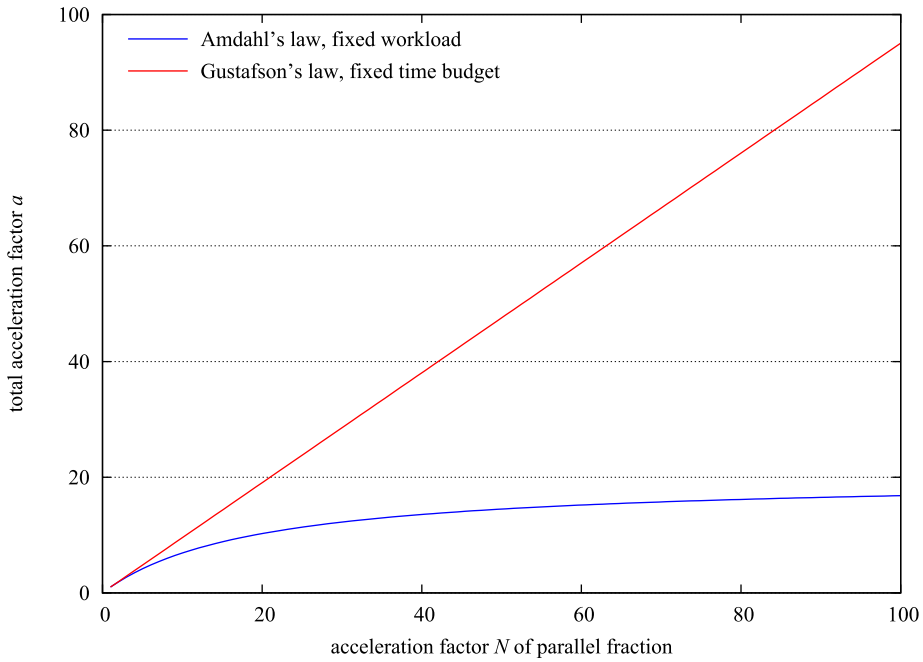


Figure 2.5: Amdahl's law and Gustafson's law. Given 95% of a program can be accelerated with factor N, the total acceleration a depends on whether the problem size (Amdahl) or the total run-time (Gustafson) is fixed.

.

maximum problem size is the primary interest of the user and researcher. Image processing is well-suited to adapt the complexity of the problem to the resources available, either by choosing a finer resolution, a larger area or volume, a longer time series, or a combination of the former.

John L. Gustafson noticed in 1988 that the time needed to compute the strictly serial part does not increase with the problem size in first approximation [30]. Hence, the workload that can be done in a fixed time grows proportionally with N for the parallel part $(1-s)$, and the acceleration increases linearly with offset s (Eq. 2.2).

$$a_{\text{Gustafson}} = s + (1-s)N \tag{2.2}$$

For a pipeline, the parallelism corresponds with the number of operators utilized at a time. An increased transistor count of an FPGA makes more CLBs available for configuration and therefore allows us to increase the length of the pipeline or the number of pipelines, leading to a corresponding speed up of the problem as a consequence. The serial part is usually caused by the time until data can be streamed into the FPGA from the periphery. The time spent between the start of streaming and the appearance of the first results is well below one millisecond and can be neglected for most use cases.

2.2 Dataflow Computing

CPUs and graphics cards are based on the von-Neumann architecture of computation. Von-Neumann machines consist of memory that holds both instructions and data, and a compute unit that executes the instructions serially one at a time. The data is then manipulated according to the current instructions [31]. Many-core CPUs contain multiple independent execution units (MIMD), while in graphics cards large groups of compute units perform the same instruction at a time (SIMD).

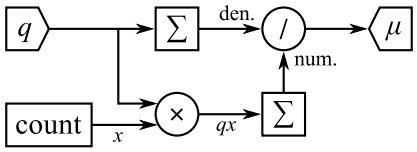


Figure 2.6: Dataflow graph of a center-of-mass calculation. The directed graph features an input (q), a generator (count), two stream operators ($\times$,/), two reductions ($\sum$) and an output (μ). It describes the calculation of $\mu = \sum_i q_i x_i / \sum_j q_j$ for all weights q that were supplied at its input. The values of x are generated by the counter.

Dataflow computing maps the data flow of a program into hardware by instantiating pipelines that perform all operations in parallel on the data stream (MISD). The concept requires the developer to leave the von-Neumann architecture behind and re-write the program. Compared to the von-Neumann architecture, dataflow computing has the advantage of utilizing the hardware evenly. In the ideal case, all operators execute a meaningful operation on the data at every clock cycle.

For the von-Neumann architecture, the serial connection between storage and the arithmetic units of a CPU acts as a bottleneck, and every data item has at least to be transferred between registers one at a time before it can be processed. The speed of main memory grows slower than processing speed and the resulting divergence is known as the von-Neumann gap that must be attenuated with the cache hierarchy [32]. A pipeline in dataflow programmed implemented with custom hardware transfers all data between operations simultaneously, leading to a much higher bandwidth for register transfers.

A dataflow program can be represented by a directed graph, with the nodes as operators and the edges as connectors between the respective sources and sinks. An operator is scheduled to run when data is available at its inputs and its output does not stall. Some nodes do not feature inputs, and some may not produce an output at every cycle. In this section, we will first classify the different nodes in the graph before examining the different kinds of scheduling.

An example dataflow graph can be seen in Fig. 2.6. It computes the center-of-mass of an input stream with weights q. The position is generated from a counter. With every cycle a data item travels along the edges and gives the center-of-mass of the weights processed so far at the output. This logic could be used for feature extraction from a line of pixels in an image.

2.2.1 Primitives

In VHDL or Verilog, a design consists of registers and combinatorial logic. The primitives of dataflow computing reside above the register transfer level. They form a higher abstraction and can be assigned to the algorithmic level on the Gayski-Kuhn hierarchy instead.

Stream operators Nodes with inputs and outputs that perform an operation on the inputs and forward the result to the output are named stream operators. Most of the actual computation is carried out by these operators that perform arithmetic or logic operations, select between sources as multiplexers or cast numbers between different formats, ranges and precisions. Albeit stream operators may contain pipeline registers, they are state-less in some sense: the result of a current operation does not affect the result of the following ones. For more complicated operations, such as floating point processing, an operator may contain many stages of registers that increase its latency, but make high clock frequencies possible and therefore lead to a high throughput.

Generators A node with no input can still be of use if it generates a stream from a constant value for a following calculation or a pattern of output values from an internal state. A counter, for

example, may increment its internal value with every clock cycle by a fixed amount. The output can then be used to keep track of the position in the stream and perform special actions for boundary conditions, such as the border of an rasterized image. Other use cases are generators for bit masks and random number generators.

Reductions A stream reduction is a node that collects information in an internal state and modifies it depending on the input values. The canonical example is an accumulator that adds every input value on top of its inner state until it is reset. Other reductions accumulate bits in a register or track the greatest value in a stream that has passed through the node. The internal state is accessible at its output, and often only of interest at the end of the calculation. They can usually be found at the tail of a pipeline.

Stream navigation For some applications it is required to compare a value in a pipeline with the previous one and calculate the difference. On the register transfer level, this can be done with an extra registers that delays the current signal by one clock cycle. The delayed signal will then be provided at the output of the register. For pipelines that are scheduled automatically, a stream navigation node indicates to the scheduler that a value is to be delayed. With more registers, the navigation range past the current value can be further extended, and a BRAM FIFO will allow to freely tap the data stream within the FIFOs range.

Inputs and Outputs The global inputs and outputs of the dataflow program are described with corresponding input and output nodes in the dataflow graph. Depending on the implementation, these nodes may often require the most resources of all nodes in order to receive and send data over the PCIe bus, the IP network, a video link or another high-level protocol.

Memory If reductions and stream navigation are not sufficient to maintain state information between items in the data stream, memory nodes provide storage with random addressing. On the FPGA they are most likely provided by BRAM that can be read and written at the same time at every clock cycle, allowing one stream of values to be stored and one stream to be extracted in parallel. Alternatively, BRAM can be used as ROM with its content to be set during configuration of the FPGA. The addresses for read and write access are set explicitly from a second set of user-defined data streams.

External memory can be used when more than a few megabytes of data must be accessed. Usually implemented as Dynamic RAM (DRAM), its latency is higher. Its relative cheap price makes it suitable to store many Gigabytes of data, similar to the working memory of a CPU system.

Finite State Machines For more complicated state transitions than the ones generators or reductions offers, a programmer may describe an Finite State Machine (FSM) and embed it into the dataflow graph as a node. The state transition will then be executed either with every clock cycle for maximum flexibility, or when the pipeline it is embedded in advances. The FSM may be described in its own language such as an RTL language. FSMs are closer to von-Neumann machines by design and may not be supported by the dataflow description tool.

The dataflow developer has multiple options to describe the graph. Depending on the tool, it can be drawn in a graphical user interface, created as the result of a program with library support, or from a compiler that comes with its own descriptive language.

2.2.2 Scheduling

After the dataflow graph has been described, it is scheduled such that only data items with the same latency are fed into the nodes, and the nodes only execute if data is available and the nodes connected to its output are ready to accept the result. The scheduling can be done dynamically during runtime, with extra logic to guard each node, or in advance and statically during compile time.

Dynamic scheduling

One of the first dataflow architectures, the Manchester dataflow engine [33], bundles each data item with a label into a token, and a node is executed only if all of its inputs have tokens available and the labels match. The labels ensure that only data items are processed with the same latency, and allow to time share a node with unrelated data streams. The concept is flexible and exploits the currently achievable pipeline parallelism at runtime. Operations that require data not yet available block on an individual level until they can continue execution.

When implemented as general-purpose hardware, the bottleneck of this and similar designs is found in the distribution of work. For the Manchester dataflow engine, the supervision of node inputs and matching of the labels requires content-addressable memory, which did not become available in sizes large enough for big applications. However, the architecture was successfully applied later to implement out-of-order execution in modern CPUs.

With custom hardware without resource sharing, the coordination between producer and consumer of data items can be implemented as a small FSM that governs the execution of each node. This requires at least a registered output that holds the data from the producer until every consumer has read it, as well as a signal to indicate availability of data set by the producer and an acknowledgement signal set by the consumers for the handshake protocol [34]. For small operations, the overhead of the FSM and extra signaling can be comparable in area to the actual logic. The individual node should therefore contain as much functionality as possible to keep the overhead small.

Static scheduling

Static scheduling aims to schedule the data dependencies of the dataflow graph a priori during compile time. This requires that the production and consumption of values is known before. Ideally, every node consumes and produces one data item with each clock cycle. No out-of-band signals are required in this case alongside the edges of the graph, because every cycle contains valid data and the whole system can operate in lock-step. The overhead is limited to observing the global inputs and outputs of the graph. The operation of the entire pipeline is then paused if a global input runs empty or an output stalls.

The architecture is very resource efficient, but puts major constraints on the logic it can describe. Some nodes, such as reductions, will produce unneeded intermediate results most of the time, and therefore require extra logic to keep track when the end result is present at their output. For complex designs the hardware parts of the design may approach dynamic scheduling again on top of static scheduling, leading to a convoluted dataflow description and a loss of abstraction.

Combined forms

In order to preserve the flexibility of dynamic scheduling and the resource efficiency of static scheduling, both approaches can be combined for dataflow computing on reconfigurable hardware. Nodes that can operate in lockstep are fused into a statically scheduled pipeline. These pipeline

elements are then connected with buffer FIFOs at their inputs and outputs, and the fill level of the FIFOs is centrally supervised to decide if a statically scheduled pipeline can run or must be paused. Depending on the abstraction level of the tool, the developer will either need to specify the boundaries of the statically scheduled subgraphs of operators, or the decision is made automatically and a set of options is provided for finer control.

2.2.3 Image Processing

Dataflow computing performs the same sequence of operations on every data item in parallel in a pipeline. An algorithm that processes a rasterized image and performs the same operation on every pixel is therefore well-suited to be implemented as a dataflow pipeline. At the input, a pixel is consumed by the pipeline with every clock cycle, and the result with the algorithm applied is presented at the output. The following primitives can then be combined at will to form a network of image-processing operations.

Point operations

An image processing algorithm that operates only on individual pixels is known as a point operation. It is solely defined by a function that has the individual pixel as an input and returns a new pixel value. The function is then repeatedly applied on every input pixel until the entire image has been processed and a new image is obtained. For color images, the input and the output of the function is a tuple that describes the color value. Point functions define the most basic effects that can be applied to an image, such as negation, thresholding, conversions to gray-scale or other color adjustments. Point operations can often be found at the early stages of an algorithm and help normalize the image for the later stages.

Point operations are efficiently implemented with dataflow computing. The image can be streamed into a pipeline pixel by pixel and line by line, and the new image is received at its output. Point operators can be extended by an additional input in order to merge two images of the same size. If the second image is a delayed frame of the same recording, image arithmetic can be used to compare images with a time offset and use the difference to quantify changes over time.

Convolutions

Point operators are limited to individual pixels. Convolutions take into account the neighborhood of the pixel at the current position. They are image-processing operators that move a small matrix, called kernel, over the image and multiply every kernel value with the underlying pixel values. The sum of the products then defines the pixel value of the output. The calculation is repeated for the entire input image until an output image of the same size is constructed. For an image that is defined by the lines and columns of matrix I, the convolution with (much smaller) kernel matrix S is defined in Eq 2.3 and yields the matrix I' of the resulting image [35]. The convolution kernel is sometimes also called stencil or mask.

$$I'_{x,y} = \sum_{i,j} I_{x-i,y-j} S_{i,j} \tag{2.3}$$

The exact operation is defined by the values and the size of the kernel matrix. Eqs. 2.4 give three example kernels [36]. S_{id} is the identity operation. The elements of S_{blur} approach a Gaussian distribution and the convolution will produce a blurred image. I_{sharp} is a Laplacian filter and is used to sharpen an image. Its effect vanishes for constant levels in the image, while intensity variations

are emphasize in the result. Convolutions can be used for noise reduction, image enhancement, edge detection and all other linear filters.

$$S_{\text{id}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad S_{\text{blur}} = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \qquad S_{\text{sharp}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{2.4}$$

To receive an equally sized output image, the input image must be extended at its borders by half the width of the kernel matrix. This can be done by adding a zero margin, by extending the image periodically or by mirroring it.

A convolution can be efficiently implemented for dataflow computing with a pipeline that keeps the image lines needed by the convolution kernel in a chain of fixed-sized FIFOs. Every clock cycle, one pixel is put line by line into the first FIFO. The pixel values needed for the convolution are tapped from between the FIFOs, multiplied with the corresponding kernel elements and finally added to form the resulting pixel at its output. A CPU implementation that performs a convolution in-place would need to save the pixel values in the neighborhood before overwriting them. Dataflow computing hides these performance-adverse register transfers in the FIFOs of the pipeline. The effect increases with the size of the convolution kernel.

Image convolutions can be extended to allow a more general neighborhood operation that is not limited to multiplications and a final addition. By introducing thresholds a neighborhood operation can also erode or dilate a shape in an image. Another use case is pattern recognition where the neighborhood of a pixel is compared with the pattern stored in the kernel matrix.

Reductions

At the end of an image processing pipeline it is often required to not produce an output image, but to quantify a certain feature of an image. In the section before, the center-of-mass an object in the image was already presented as an example. The creation of a histogram reduces a picture to a vector of values that describe the frequency of a intensity levels or color. In dataflow computing these reductions can be calculated with a stream reduction if the access pattern on the input image can be made linear.

Operations with non-linear access patterns

Some image processing operations require a non-linear read pattern for the input data or a non-linear write pattern at the output. Geometric operations, for example, can only be implemented without buffering the entire image in RAM if the orientation of the image is preserved. The pipeline will then either discard values at its output to scale the image down, or insert cycles that do not read from the input for up-scaling. Rotations, reflections and general affine transformations require the image to be fully buffered in BRAM or a different randomly addressable storage, and the output image is started to be generated only after the input image was buffered. This disturbs the flow of data in the pipeline and increases the latency accordingly. Other common operations that require a non-linear access pattern are bucket fill and the discrete Fourier transformations.

Chapter 3

State of the Art

Computer hardware and the tools to program it have kept constantly improving since the invention of the programmable computer. The best-known measure for integrated hardware is probably Moore's law. It successfully predicts a doubling of the transistor count every 18 to 24 months since 1965 [37] and is still in use as a road map for the semiconductor industry today [38]. Since the end of clock frequency scaling within the last decade, the growing size of computational hardware has lead to increased parallelism, and it has created a need for tools that allow a description of programs and FPGA configurations that allow the developer to keep up with the growing complexity of the hardware. This chapter gives an overview of the state of the art on parallel hardware suitable for high-performance computing and then focuses on the software tools to utilize dataflow computing on FPGAs. The applications presented in the following chapters of this thesis were accelerated with the FPGA systems from Maxeler Technologies.

3.1 Application Accelerators

The power consumption of a digital microprocessor scales linearly with its clock frequency [22]. The resulting heat dissipation problem has forced the scaling of clock frequencies to a halt at about 4 GHz as an upper limit in 2005. The effort-less acceleration of software fueled by frequency scaling of the hardware ended [39]. Since then, the still-increasing number of transistors is used to parallelize microchips for high-performance computing, and requires the programmer to re-think the application logic to harness the new options. Depending on the hardware, either the number of compute cores is increased, or the length of the compute pipelines is extended. Many-core CPUs and graphics cards belong to the first type. For CPUs, acceleration is reached through increasing the number of cores, with each core capable of performing an independent calculation. Graphics cards also gain compute power by adding cores, but require large groups of cores to execute the same program. FPGAs benefit from an abundant transistor count by increasing the length of the compute pipelines, or by multiple instantiation of the same pipeline.

Application accelerators are considered distinct systems that are controlled by a host CPU. For all three types, PCIe cards exist that can be added to an existing compute cluster with conventional hardware. The data transfers to and from the accelerator are controlled by the host system, and the accelerator acts as a co-processor for the compute-intense parts.

3.1.1 FPGA accelerators

The main vendors of FPGAs are Xilinx and Altera. Together, they dominate the market with a combined share of over 80% [40] and provide the FPGAs with the most reconfigurable resources. FPGAs are soldered on extension cards in order to use them as application accelerators or, for

bigger systems, are mounted into a case into a compute rack and connected through a network link with the controlling CPU. Other systems are built on FPGAs that contain a small dedicated CPU to free the remaining FPGA from setup, control and I/O logic, and can therefore operate independently of an external CPU.

FPGAs suitable for high-performance applications are, similar to CPUs and graphics cards, only available from commercial vendors that do not disclose all levels of the design. They can therefore only be configured through proprietary tools that form the base of the tool chain. The front-end and the compiler that translates the source to an intermediate representation may be offered by a different supplier.

The peak performance of an FPGA accelerator card is given by the actual FPGA model and, for data-intense task, by the properties of the external memory and the bandwidth of the connection with the host CPU. For a Xilinx Virtex-7, up to 5.3×10^9 multiplications and accumulations (MACs) can be performed with the DSPs found on a Virtex-7 1140T [41]. The actual peak performance depends very much on the width of the chosen floating-point encoding. For single precision, at least two DSPs are needed per operation, resulting in 2.6 GFlops. For double precision, ten DSPs are needed and the peak performance is pushed down to 0.5 TFLOPS [42]. However, care must be taken when comparing the peak performance with other accelerator architectures, because the hardware designer may choose a more efficient custom floating-point encoding that saves hardware resources, or create a design that abandon floating point altogether for a fixed-point encoding.

Maxeler Technologies

The application accelerators from Maxeler Technologies [2, 43] are available individually as PCIe extension cards or bundled in a case and connected through an Infiniband network. Each card contains the FPGA and multiple DRAM modules for up to 90 GB of on-board RAM. The PCIe controller is part of the FPGA configuration. Some models feature SFP connectors for optical Ethernet networks.

The cards are controlled through a Linux kernel module (MaxelerOS) and a C library (MaxCompilerRT) that allows the executable on the host to configure the FPGA and stream data to and from the accelerator. An overview of the system is given in Fig. 3.1. The on-card FPGA is shown in the center alongside DRAM memory. The host is a CPU system that controls the card and moves data via Direct Memory Access (DMA) through the PCIe connection or Infiniband. For systems with multiple extension cards the FPGAs can be connected with a proprietary MaxRing connection for data sharing with low latency.

The memory ports of the on-card DRAM modules are operated in parallel, such that the burst length for DRAM access is the sum of burst lengths of the individual modules. This ensures a maximum throughput for serial reads, but requires extra care for random reads smaller than one combined burst length. The on-board hardware does not automatically sync data that is distributed in different locations. Data to be shared between FPGAs, DRAM and CPU must be send and received explicitly by the configured logic and the application on the host CPU.

The motherboard of the host system must support 2×PCIe x8 through a 16-lane PCIe connector. Maxeler Technologies offers evaluation systems on request that bundle CPU, motherboard and FPGA accelerator card in a single system. Two of these systems were used for the applications presented in this thesis.

The accelerator cards are programmed and configured with the proprietary tool chain, and are not directly accessible for configuration with the tools of the FPGA vendor alone. Instead, the FPGA configuration is described as a dataflow graph in Java. The graph is converted to partially encrypted VHDL in an intermediate step before the bitfile is generated with the synthesis tools of the FPGA vendor. The FPGA are produced from either Xilinx (Virtex series) or Altera and may be

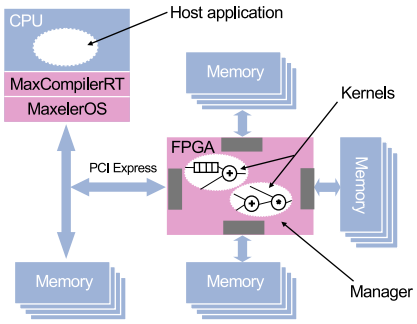


Figure 3.1: Architecture of the application accelerator from Maxeler Technologies. Image: Maxeler Technologies [2]

chosen from a pre-selected list.

The first applications of the company were centered around seismic applications for oil and gas, where a range of applications got accelerated by a factor of 60 to 200 compared to the same space required by CPU nodes. The focus has since been extended to risk analysis for financial institutions and high-frequency trading. The university program MAX-UP was founded in 2010 and provides price-reduced hardware for research and education.

Convey Computer

The FPGA accelerators from Convey act as closely integrated co-processors on the system bus, similar the numeric co-processors found in the pre-Pentium area. CPU and FPGA share the same address space in memory. The execution of the co-processor is triggered by extensions to the x86 instruction set of an Intel CPU. The configuration of the FPGA co-processor is loaded into the FPGA at application start-up. It is chosen by the application developer to suit the hot spots of the program. The FPGA configurations are called “personalities”, some of them are already offered for applications covering bioinformatics, Big Data and further mathematically functions commonly needed in high-performance computing. If needed, a custom personality can be created with the Personality Development Kit on top of the Xilinx tool chain [44].

The tight integration between CPU and FPGA allows random memory access by the FPGA that is sped up by the cache hierarchy of the CPU. The memory latency is not impacted by the latency of the PCIe bus as it is the case with most accelerator cards. Existing applications can therefore be adjusted to benefit from the co-processor with less modifications to the program’s data streams, especially when compared to the dataflow approach. The applications stays in the realm of the von-Neumann architecture and achieves its speed-up through (static) parallelism on the instruction level executed on the FPGA. Applications from bioinformatics, such as genome sequencing, are reported to achieve an acceleration factor of six to fifteen [45].

Convey did not offer the dataflow parallelism that is subject of this thesis. Reported acceleration factors have also consistently been stated to be smaller than for Maxeler hardware. Applications that do not follow a streaming approach, such as graph processing in Big Data with a random memory access pattern, can still benefit to a larger extent from a tightly integrated co-processor thanks to shared memory. [46]

Silicon Software

Silicon Software provides accelerator cards similar to the ones offered by Maxeler Technologies, but optimized for live image processing from a video source. The cards receive the frames from a Camera Link or GigE Vision connector. The frames are processed in real-time pixel by pixel in a

system of pipelines on an embedded Xilinx FPGA. The results are then received by the host CPU via PCIe. [47]

The application domain is focused on machine vision and optical quality assurance, for example as part of a production line in a factory. The cards are not designed for arbitrary data exchange between host and on-board memory. Instead, data processing is optimized for image frames. The hardware development kit relies heavily on end-of-line and end-of-frame signals that require workarounds when non-image data formats need to be processed. The application for localization microscopy that is presented later in this thesis was ported to the hardware of Silicon Software by Manfred Kirchgessner within the scope of his diploma thesis [48].

3.1.2 Graphics Cards

Graphics cards are application accelerators for graphics processing, especially for the rendering and presentation of virtual 3D objects. Their rendering pipeline is optimized to process the different stages from a set of triangles towards a representation that approaches photo realism. Some of the pipeline stages can also be used for general purpose computing, notably the vertex shaders intended to transform the geometry of an object by executing a shader program on the vertices of the triangles. The same program is executed by the graphics card with an SIMD architecture in a highly parallel manner to achieve the original goal of rendering frames with a frequency higher than the human eye is able to distinguish. [49]

General Purpose Computation on Graphics Processing Units (GPGPU) benefits from the massive parallelism introduced with the rendering pipeline, but shares also some of the drawbacks, such that individual instances of the shader program all have to execute the same instruction at the same time. Double precision number formats, while available on all major CPUs, must be emulated at lower speed because these formats were not requirement for the original use case.

Since their introduction as graphics accelerators, graphics cards have evolved towards application accelerators, and the main vendors offer special graphics cards for high-performance computing, some of them even missing a connector for a video screen. The mass market of 3D games drives the number of graphics chips and impacts the scale effect of chip production favorably, leading to low prices per floating-point operation per second (FLOPS) when compared to general purpose CPUs.

Graphics cards can be programmed in the Open Compute Language (OpenCL) [50], an open standard consisting of a subset of the C programming language without recursion, system calls or dynamic memory management. The developer writes a compute kernel in the language, and the graphics card executes it in SIMD fashion on a C array of data, once for every item. The kernel can be thought of as the body of a for-loop on the input array, and the hardware defines the parallelism of the kernel execution. An individual thread of execution runs slower than it would be on a CPU, but the program is accelerated by the execution of thousands of threads in parallel. For programs without data dependencies between loop runs, the computing model is similar to dataflow (MISD) computing, but with vector instead of dataflow parallelism. In both cases the compute kernel is implicitly wrapped into a for-loop that iterates over an array and maps the data to a second array of the same size, or reduces it to a smaller set of values.

Nvidia Tesla

Nvidia, the biggest vendor of graphics cards, has focused on GPGPU as a second source of revenue with the introduction of the Tesla series. The Tesla K40 [51] is the current top model with 2.880 cores and a peak processing power of 4.3 TFLOPS per card for single and 1.4 TFLOPS for double precision. The series is optimized for high-performance computing in a server rack and omits the display connectors for video output.

Nvidia introduced the Compute Unified Device Architecture (CUDA), a programming model based on subsets of the C and C++ languages to describe programs on its graphics cards. A Fortran implementation exists as well. Besides the limitation mentioned above, the programming model discourages the usage of branchings in the kernel. If a branching occurs, the other treads running in parallel will need to wait until the branch is completed and execution can resume for all threads at the end of the branching again with the same instruction. For floating-point operations, some features from the standard are missing, such as subnormal numbers, NaN and some rounding modes [52]. Since 2008, all CUDA-enabled graphics cards also support OpenCL as an alternative language.

AMD Fusion

Advanced Micro Devices (AMD) offers graphics cards similar to NVidia that can be programmed with OpenCL as application accelerators. The top model, the FirePro S10000, achieves 5.9 TFLOPS on 3584 compute cores for single precision and 1.5 TFLOPS for double precision. A lower performance for double precision and a higher energy usage restrict their usage to only one system at rank 59 in the TOP500 Supercomputer Sites, while Nvidia cards appears 37 times in March 2014 [53].

Still, the design of AMD's accelerators promises certain advantages with the "Fusion" concept [54]. AMD produces both CPUs and GPUs and is therefore able to integrate the general purpose unit closely with the graphics card on the same die. Both units are connected with the system bus that is also used to access the RAM. The latency of the PCIe bus is avoided and the energy usage decreased. CPU and GPU are programmed with OpenCL. The Fusion design is currently focused on gaming in the Sony Playstation 4 and Microsoft XBox One.

3.1.3 Many-Core CPU accelerators

The multiplication of independent processing cores was, along with bigger caches, the main driver of recent advancements of CPU throughput [39]. General purpose CPUs are still limited to a single-digit number of CPU cores. A way to overcome this limit and increase the number to dozens of cores was found in the simplification of the individual cores to save silicon area, and in connecting them through a ring bus to avoid locking of the interconnects. This concept is implemented by the Xeon Phi accelerator from Intel.

The Intel Xeon Phi 7120D was released in 2014 and contains 61 compute cores connected by a wide ring bus (Fig. 3.2). Each core is made of the original Pentium design, heavily upgraded with 64-bit support, a larger cache and 512-bit wide vector instructions accompanied by 32 registers. All cores combined, the system has a peak performance of 1.2 TFLOPs for double precision. [55]

As an advantage the system can be programmed in code close to the x86 standard, unlike the previously described accelerators. The tool chain must be modified only slightly and supports general C++ as a preferred language. This enables researchers to quickly build libraries for the architecture [56]. At the writing of this thesis the fastest supercomputer, Tianhe-2, was build with Xeon Phi accelerators, alongside with 14 other systems in the TOP500 of March 2014 [53]. So far the Xeon Phi is the only many-core CPU accelerator card in the ranking.

The programming relies on compiler-driven parallelization and vectorization. The accelerator can achieve the highest speed-up if the data structures are kept simple and allow for massive parallelism [57]. While the overall effort needed seems lower to translate a program into a high-performance implementation for the Xeon Phi, good knowledge of the internal architecture is still a requirement during development.

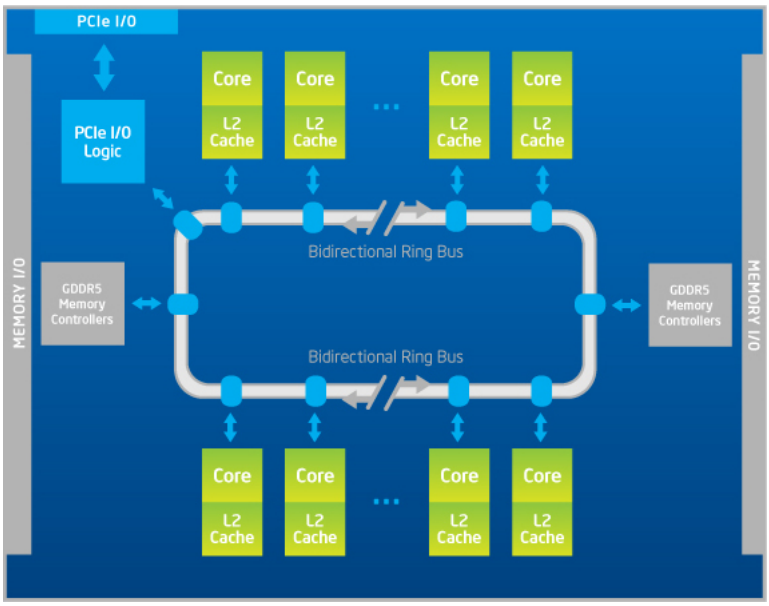


Figure 3.2: Intel Xeon Phi architecture overview. The design contains up to 61 compute cores, each connected to a bidirectional ring bus. The cores are based on the Pentium design with added caches, vector instructions and 64-bit support. The host CPU is accessed through PCIe. Image: Intel

3.1.4 Summary

All accelerators require at least some special domain knowledge to keep up the promised speed-up. Some, like the Xeon Phi, are closer to the imperative code found in the most popular programming languages C and C++ that maps well to x86 hardware. The FPGA accelerators with dataflow computing can be found at the alternate end.

The speed advantages of the individual systems can only be partially compared. The peak performance in FLOPs gives only a first hint. The ability to re-write the original software and change its data structures and parallelism are important factors to consider as well. For FPGAs, pipeline parallelism promises the best results and requires the adaption of the program's data types and access patterns. For MISD architectures, namely graphics cards, the algorithm is best accelerated through massive vector parallelism. The Xeon Phi, which may act as a pioneer for future CPU architectures, is still young, but has already gained its place for accelerated multi-core processing in the supercomputing world.

In terms of flexibility during development, the architectures offer very different approaches. CPU accelerators and GPUs are relatively flexible during development, due to the usage of C/C++. dialects. In terms of hardware flexibility, the FPGA offers the biggest range of supported parallel taxonomies. Any architecture between MISD and SIMD can be instantiated, allowing the accelerated execution of programs that are unsuitable for the other accelerators. Pipeline parallelism (MISD) usually fits best, and can be extended by multiplying the number of pipelines if the FPGA still contains unused resources (MIMD), and if the problem allows for additional vector parallelism.

3.2 High-Level Hardware Description

Since the creation of VHDL and Verilog as hardware description languages for FPGAs in the late 1980s, efforts have been taken to lift FPGA programming from the register transfer level to a higher abstraction. The different levels of abstraction in hardware design are shown in Fig. 3.3. The level

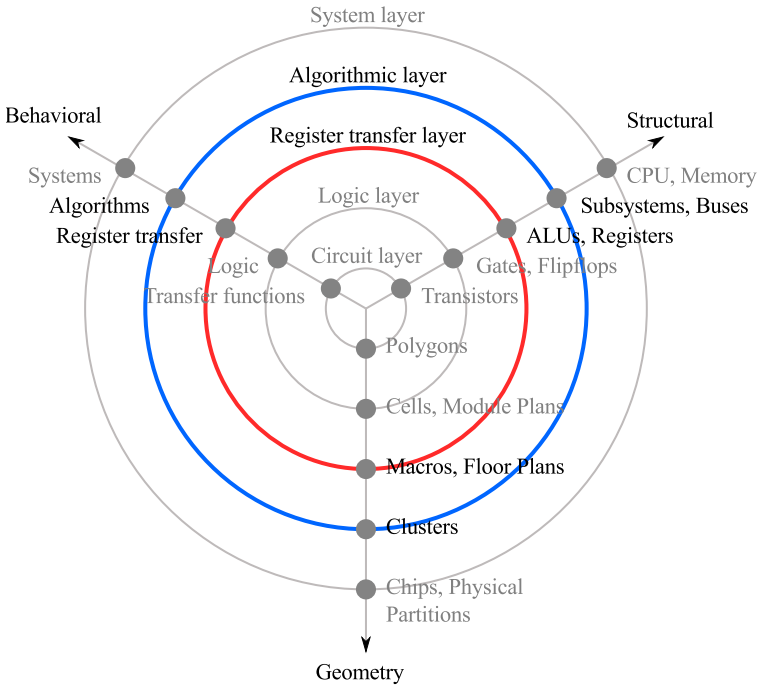


Figure 3.3: Gajski-Kuhn chart of hardware design. VHDL and Verilog operate on the complexity level of register transfers. High-level hardware languages move the description of the design into the algorithmic layer.

of abstraction rises from the inner towards the outer rings. The chart was created first by Daniel Gajski and Robert Kuhn in 1983 [58].

On the FPGA the logic layer at the center of the chart is found at the actual wiring of the FPGA silicon chip and the implementation of its active elements. This layer is inaccessible to the programmer and its design is kept closed by the vendor. The next layer, the logic layer, contains the logic elements of an FPGA: LUTs, FFs, DSPs and similar elements. VHDL and Verilog describe hardware on the layer above, the register transfer layer (RTL). Here, the behavior of a design is described as the transfer and transformation of logic values through combinational logic between registers. Despite the fact that both languages are able to describe algorithms with imperative statements on the higher levels, the subset that can actually be synthesized to hardware largely remains on the register transfer level. The higher-level constructs, such as loops with a variable number of iterations, are limited to simulation. The development of large system is eased by intellectual property cores (IP cores) that act as reusable units of logic and range in complexity from simple adders to entire CPUs. These building blocks can then be combined by hand with glue logic on the register transfer level.

In this section an introduction is given to the efforts of describing hardware on the algorithmic level. The primitives found in these languages for constructing more complex designs already span multiple registers with combinational logic in between and can be assigned to the algorithmic level in the Gajski-Kuhn chart. The hardware designer typically combines mathematical primitives, such as arithmetic operators, and does not have to keep track of the data orchestration on the levels below. A higher-level description promises a shorter training period for the developer, gains in productivity and, with FPGA silicon areas increasing, the option to design logic that would otherwise overwhelm the capabilities of a development team due to its complexity.

3.2.1 Imperative Languages

The 10 most popular computer languages can all be assigned to the paradigm of imperative programming [59]. Imperative programming languages describe computation as a sequence of statements embedded into control structures (e. g. branchings and loops) that manipulate the state of a program. Among them are C, C++ and Java, and most programmers are able to write source code in at least one of them. VHDL and Verilog, on the contrary, do not follow the concept of linear execution of statements. Instead, all component are executed in parallel. VHDL and Verilog require special training, and a compiler that could automatically translate an imperative description towards a hardware description would make hardware programing available to a much larger group of (software) developers [60].

Handel-C

Handel-C is one of the earlier imperative languages for high-level hardware description for FPGA programming and inherits most of its syntax from C. It was created in 1996 at the Oxford University Computing Laboratory. The language features most of the data types from C except for unions and floating-point encodings, which are included as library functions. Integer types can be changed in size during variable declaration to make use of the flexibility of reconfigurable hardware. The language was also reduced by recursion, side effects in expressions and dynamic memory allocation [61].

Plain C executes statements linearly. To describe hardware parallelism, Handel-C introduces the `par` block (Listing 3.1). At the beginning of the body the execution flow splits and every statement (here `f()` and `g()`) is executed in parallel. The statements after the `par` block are executed sequentially again after every statement in the body has finished.

Information can be exchanged safely between parallel parts of a program through channels. These are FIFOs with a given capacity. When the FIFO is full, adding an element will block the sender until an element has been removed. The receiver will block accordingly when the channel is empty. Channels with zero capacity can be used to synchronize sender and receiver. The concept follows the notion of Communicating Sequential Processes (CSP), which was developed at the same university and is known in high-performance computing as the underlying concept of the Message Passing Interface (MPI).

Handel-C does not attempt to auto-parallelize code. The statements outside of the `par` block are executed sequentially, with every assignment consuming one clock cycle. The `delay` statement can be used to introduce an idle cycle. With a combination of `par` and `delay`, pipelines can be created for MISD parallelism, although the programmer has to schedule the data streams manually.

```
par {
  f();
  g();
}
```

Listing 3.1: The `par` block in Handel-C. Statements within the body (f() and g()) are executed in parallel by the synthesized hardware. The functions are expanded at compile time and do not require a stack.

Xilinx Vivado High-Level Synthesis

With the introduction of the Vivado design suite for its FPGAs of the 7-series in 2012, Xilinx also included an imperative high-level language to describe IP cores. Formerly known as AutoPilot, Vivado High-Level Synthesis (Vivado HLS) uses an extended subset of C for hardware design. The language omits, similar to Handel-C, the implementation of recursion, memory allocation and

system calls. Numeric data types can be refined to arbitrary precision, and a standard library for common mathematical operations is included. [62]

The entry point of an IP core is given by a top-level C function that can be embedded into a C or C++ test bench or synthesized to hardware. The arguments of the function later form the inputs of the created IP core at the RTL, and the outputs are derived from the function's return value. C arrays are translated to BRAM and add further inputs and outputs if they appear in the top-level function. Compilation produces VHDL, Verilog or SystemC as an output that can then be embedded into a custom design of the same RTL language.

The compilation process creates a mixture of pipelines and controlling finite state machines (FSM) to map C programs to hardware. By default, the hardware executes the C statements in a strictly sequential manner, similar to Handel-C, but without functions expansion. This can lead to hardware elements performing no operation for most states of the controlling FSM. Pipeline parallelism can be manually added, though, to address performance bottlenecks with pipeline directives that appear as C `#pragma` in the source. When a sequence of functions is pipelined, every function is translated to a hardware block and the blocks are connected with FIFOs, allowing each block to operate in parallel. When a loop is pipelined, the compiler will attempt to execute the body of the loop with pipeline parallelism if the data dependencies allow it. If pipelining fails the compiler will generate a performance report to debug the data dependencies and resource conflicts. Other directives exist to unroll loops or inline functions.

Vivado HLS is one of the few tools that supports both the dataflow and control flow domain. First results show that the resource footprint on the FPGA of the synthesized hardware resides remains low for image filter applications when compared to other tools for high-level synthesis [63].

ROCCC 2.0

The Riverside Optimizing Compiler for Configurable Computing 2.0 (ROCCC) is a C-to-hardware compiler that produces strictly pipelined hardware modules from the input [64]. It translates the most restricted subset of C to hardware modules in VHDL. By connecting the inputs and outputs of these modules via data streams more complex systems can be created in a second step. For this, ROCCC strictly distinguishes between module code and system code. Both are written in C, but support different subsets. Module code operates on individual data items, while system code handles data streams through a C-array representation.

Owed to the requirement that all module code must be translated to a statically scheduled pipeline, the module code in ROCCC does not support short-circuit evaluation. This kind of evaluation is a feature of C that causes the second operand of the binary operators `&&` and `||` to be evaluated only if the value of the first operand requires it. These operators and, for the same reason, the `?:`-operator have been removed from the language. Further restrictions in module code forbid generic pointers, non-for loops, C-library functions calls and non-pure functions. As with the systems described before, system calls, recursion and dynamic memory allocation are neither part of the language. [64]

Inside a module, the compiler unrolls all loops into a pipeline and must therefore know the iteration count at compile time. Outside, at the system level, the compiler will not unroll loops over the streams that connect the modules. Instead, the streams are represented as C arrays. The compiler determines which elements are accessed during an iteration and will determine the size of a stream window that covers all accesses. This stream window is then implemented with BRAM and caches the access to the data stream. The feature is especially valuable when an image convolution is to be implemented. The BRAM will automatically form a buffer for multiple lines such that the convolution kernel can access all covered pixel in parallel. The convolution kernel itself can be

implemented as a module. In practice, care has to be taken to not let the stream window grow too far and consume all available BRAM resources [63].

ROCCC is open source and its license allows the code to be modified. Researchers can study high-level transformations of the imperative source when mapping procedures, loops, memory access patterns and other constructs to hardware without the need to build their own compiler [65].

3.2.2 Dataflow Descriptions

The semantic gap between imperative descriptions of an algorithm and its final form as a dataflow pipeline or FSM mapped into the silicon of an FPGA creates tensions between source code and hardware. From the compilers shown above it can be concluded that at the current state of the art the semantic gap leads to either heavy hand-optimization of performance-critical hardware modules or to a very restricted subset of C as the input language. In both cases, the developer must be aware that the target architecture imposes major restrictions to the design process.

Dataflow descriptions follow a different approach. The hardware developer is required to re-write the algorithm into a pipelined form prior to coding. Afterwards, the hardware is described as a directed graphs with nodes that represent computations. The transformation is then more straight-forward and limited to optimization, while the overall structure of the input description is preserved. As an advantage, resource usage and performance can be estimated from the input description, with every primitive corresponding to a certain resource footprint and maximum throughput. The actual process of porting the algorithm is moved to the beginning of the design process.

MaxCompiler

MaxCompiler from Maxeler Technologies uses a Java dialect to translate a dataflow graph to VHDL [2, 66]. Contrary to former efforts that aimed to translate Java to hardware [67], the programming language is only used to build the data structure of the dataflow graph. The Java program is not translated itself. The Java dialect used for this is called MaxJ, a superset of Java that contains operator overloading to connect the graph edges with arithmetic operators. For simple, sequential-only algorithms that operate on a data stream the result resembles imperative code. In Listing 3.2, the multiplication of `x` with itself acts on a stream of values, and the entire kernel object will get translated to a statically scheduled pipeline. More advanced structures, such as the shown accumulator, are instantiated as Java objects. Other node types for the graph include value nodes to produce a stream of constants, counters for bit pattern creation, and I/O nodes. Stream offset nodes shift a stream into the past or the future within a fixed number of clock cycles and are resolved into delays during scheduling.

Branchings have to be implemented with the `?:` operator or by explicit instantiation of a multiplexer. Loops must be scheduled mostly by hand and are created by connecting the output of a pipeline segment (identified by an object of type `HWVar`) back to one of its input. The pipelines from the `Kernel` object are then embedded into a manager that connects the kernels with FIFOs to the host CPU, a network device on the accelerator card or to other devices. A manager can also hold multiple kernels, and will run them only if none of their inputs or outputs blocks. Multiple kernels with FIFOs in between can then be used for computing problems where the data cannot be piped in lock-step through a single pipeline,

The code is developed in the Eclipse IDE [68] with the support of the MaxJ plug-in (Fig. 3.4). Running the MaxJ program will then produce the VHDL source from the dataflow graph. The VHDL will be passed to either the Xilinx or Altera tool chain afterwards, depending on the vendor of the FPGA. The MaxCompiler library also produces an annotated version of the MaxJ source

code that labels every line of code at the left margin with the number of FPGA resources it has occupied.

The Maxeler tools were used in this thesis for the implementation of the example applications. The included functions do not specifically support image processing, but the needed functions could be easily created. The usage of a general purpose language for the generation of the dataflow graph makes the creation of libraries possible within the same language, where each MaxJ function generates a dataflow subgraph that can be parametrized and re-used. Another use case occurs when the FPGA is not fully occupied after synthesis, and the graph can be extended for multi-piping by moving the responsible lines of MaxJ into a loop that instantiates a new subgraph with each iteration.

For testing, parts of the dataflow graph can be translated automatically to a C program that simulates the hardware with bit accuracy. Input creation and output verification is done in MaxJ as well. For program parts that do not fit the dataflow paradigm, the developer may use FSMs instead or revert to VHDL components.

```
public class AccuKernel extends Kernel {

  protected AccuKernel(KernelParameters params) {
    super(params);

    HWType intType = hwInt(32);

    HWVar x = io.input("x", intType);
    HWVar x2 = x * x;

    Params accuConfig =
            Reductions.accumulator.makeAccumulatorConfig(intType);
    HWVar a = Reductions.accumulator.makeAccumulator(x2, accuConfig);

    io.output("a", a, intType);
  }
}
```

Listing 3.2: Accumulator in MaxJ. The generated pipeline squares the values of the input stream ”a“ and produces the accumulation at its output.

Silicon Software Visual Applets

Contrary to the hardware description languages shown above, Visual Applets from Silicon Software does not use a text representation for hardware description, but a graphical user interface [47]. The user drags operators and links with the computer mouse to form a pipeline. The software then performs most of the synchronization, schedules the pipeline and synthesis the hardware with the Xilinx tool flow. Fig. 3.5 shows the top-level view of a design.

Silicon Software offers special boards with Xilinx FPGAs that can be directly connected with a camera. The results of the image operation are then shared with the host CPU through PCIe, similar to the accelerator cards presented before. It is also possible to feed imagery from the host CPU into the FPGA, as shown in the figure with the entry node ”DmaFromPC“.

Visual Applets distinguishes between two types of operators. O-type operators are simple operators for arithmetics and logic functions. They can be combined freely to form a point function and act on the individual pixel or perform basic reductions. M-type operators contain memory to buffer parts of a frame. They perform more complex tasks, such as convolutions and object detection. Due to the buffering, special care has to be taken by the user when the output of multiple

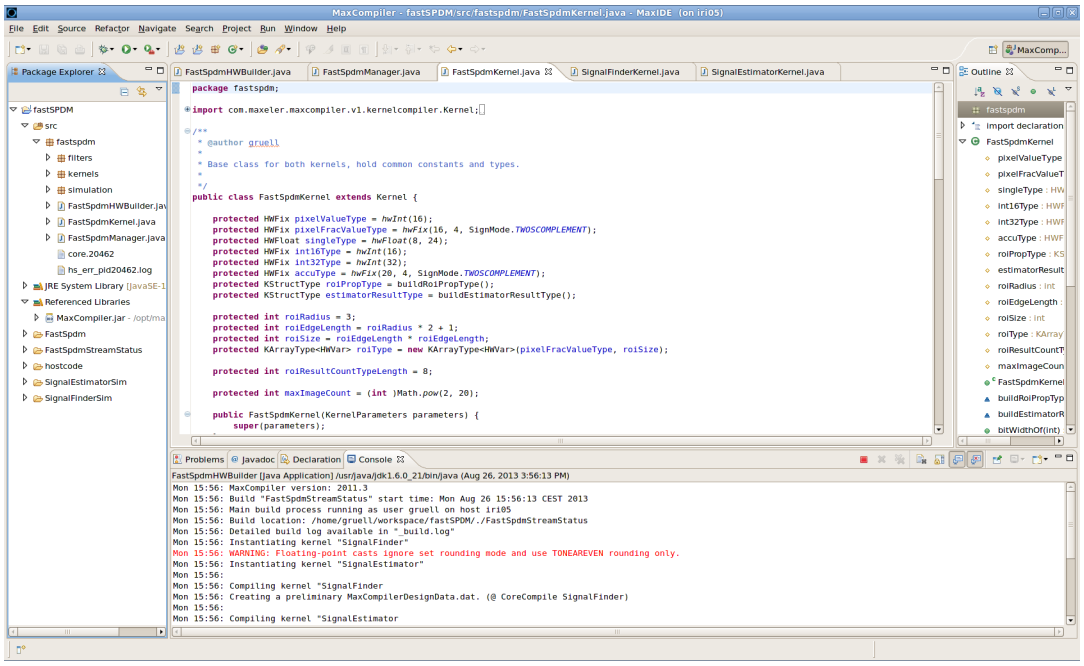


Figure 3.4: Screenshot of the MaxIDE 2011.3 integrated development environment. The dataflow graph is described with the MaxCompiler software library and MaxJ, an extension of the Java programming language. Synthesis can be started directly from the graphical user interface or through makefiles.

M operators is to be used as input for the same operator. The differences in latency may otherwise cause a dead-lock in the design. [69]

The programming environment focuses on real-time image processing from a camera. The offered operators all rely on hidden out-of-band signals that indicate the end-of-line and end-of-frame when an image is transferred through the pipeline pixel by pixel. For data that does not constitute an image, the format must be changed such that it can be still processed. An array, for example, will need to be treated as an image with one line of pixels, and the pixel values encode the array elements.

One of the example applications presented in this thesis, the acceleration of localization microscopy, was carried out with Visual Applets by Manfred Kirchgessener within the scope of his diploma thesis [48]. He used the ability to feed data into the card from the host CPU in order to process the imagery that was collected before.

When this thesis was started, Visual Applets 1.4 did only support linear pipeline graphs. Loops could only be physically implemented by adding a daughter board on top of the accelerator card and routing the signals back into the FPGA. This has been solved in later versions.

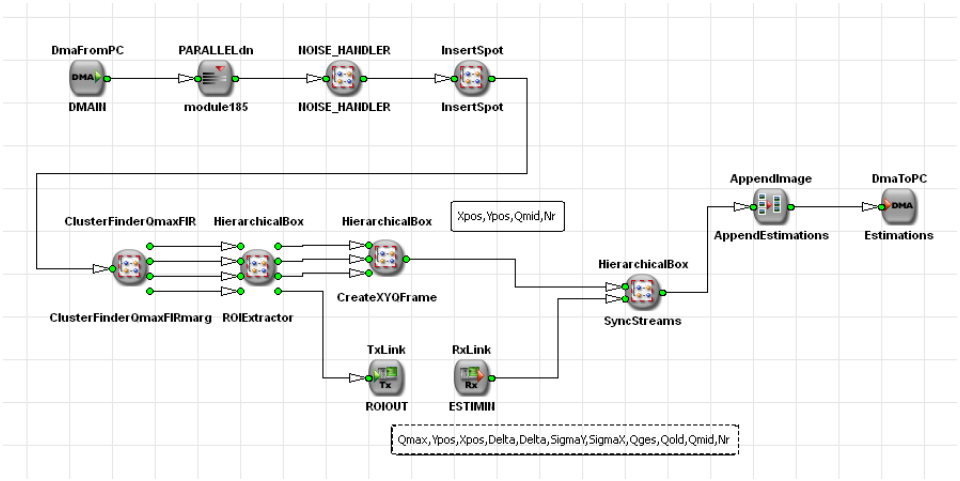


Figure 3.5: Graphical dataflow description in VisualApplets (Silicon Software). The image shows the top-level view of the pipeline implementation for localization microscopy. Each ”Hierarchical Box“ operator contains other operators. Image: Manfred Kirchgessner [48]

Chapter 4

Acceleration of Imperative Code with Dataflow Computing

In this chapter we will examine how the semantic gap between imperative code and a pipelined hardware architecture can be bridged. Imperative computer languages like C and Fortran, but also object oriented languages like C++ and Java define a program with statements to control the flow of execution. CPUs directly support jumps in the control flow within their instruction set, and the development of computer languages has started with the goto statement. Imperative languages then moved on to branchings and loops to organize jumps in the control flow in a more structured manner. Functions in functional languages and polymorphic member functions in object oriented languages provide further abstractions to describe the control flow.

Dataflow computing, however, lacks these kind of control structures. The aim of a dataflow description is to organize the flow of data instead of the flow of execution. The dataflow description of a program can be mapped well to an FPGA by transforming the dataflow graph to a system of pipelines in hardware, where each pipeline operate in lockstep. An implementation of control flow logic, however, would require state machines or simple processors that would serialize operations and could not fully utilize reconfigurable hardware, because at each time step all resources not required by the current instruction would remain unused. Before laying out the paths for porting imperative code towards a dataflow description we start with an overview of the techniques that arise naturally when processing streams in a set of pipelines.

4.1 Relation to list processing

A stream of data in hardware is a serialization of values, where one value from a series is presented at a time in forward order. Without buffering, data that has been received in the past cannot be read again, and data from the future can only be accessed when we wait accordingly. The absence of free navigation in the data is a major constraint that immediately affects the design of a dataflow application. It contributes to the difficulties that arise when programmers from imperative languages learn dataflow computing for the first time.

In this section, we want to show that dataflow computing can be related to a concept that already exists in all popular computer languages. It will serve as an introduction to the semantics of dataflow computing. A stream of data in a pipeline has, beside from being navigable in only one direction, the property of immutability: data can only be read from the output of the pipeline elements that form the inputs of the next pipeline element (Fig. 4.1). The resulting data stream is a new stream and does not overwrite the input streams. The streams may be used as inputs by other pipeline elements as well and are not affected by their consumers. Dataflow computing on the

Figure 4.1: An immutable, flat and single-linked list closely implements in space the semantics of a stream of data in time. Both formats can only be read in one direction.

pipeline level therefore copies immutable data at every pipeline stage.

The properties of unidirectional navigation and immutability can be found in software in *single-linked lists*. This data structure can be traced back to the first computer languages [70] like C and Fortran. It is the primary data structure of Lisp [71] and can also be found as a first-class data structure in functional languages such as Haskell and declarative languages such as Prolog. Flat single-linked lists can be seen as a mapping of the temporal properties of a data stream into storage space. This picture enables us to go through the set of essential functions for list processing and state a dataflow implementation for each of them. Building on top of the pipeline primitives from section 2.2 on page 14, these solution will then allow us to describe the translation of imperative code on a higher level in the following sections.

List processing is also closely related to image processing. Raw images and 3D voxel data can easily be streamed to and from an FPGA by storing them as a multi-dimensional array. The data is then serialized by accessing it row by row, column by column and, for 3D volumes, layer by layer.

The relationship of dataflow computing and stream processing was already noted for general purpose dataflow machine with token labeling [72, 73]. Hardware pipelines that operate in lockstep need static scheduling that has to be carried out at compile time, rendering them less flexible. However, all concepts of list processing can be implemented, in some cases through the support of the memory controller. If the pipeline cannot be designed to consume or emit one data item per clock cycle, inputs and outputs have to be used that can be disabled for some cycles.

The standard library "Data.List" in Haskell [74] for processing of immutable, single-linked lists will be used as a guide to structure dataflow functionality in FPGA pipelines. C++ also provides a standard library for linearly traversable data structures in the <algorithm> library [75] that can be used similarly, but is less exhaustive and was written with a focus on mutable lists.

The data to be streamed to or from the FPGA does not have to be stored as a linked list, of course. Instead, the data can be held in memory as an array to save storage space. The memory controller can then read the array linearly and pipe it into the FPGA, as well as write back the results accordingly. For the following implementations we will assume that the input lists are stored in memory, either on the FPGA board or the host computer, and the results are written back to memory. To implement non-linear access patterns, the memory controller can optionally accept a stream of read or write addresses that define the storage location of each data item. An overview of such an FPGA system is pictured in Fig. 4.2.

When comparing Haskell's list processing functions with dataflow computing, we cannot pass around functions as first-class objects in dataflow computing. However, inlining can be used to statically modify the generic list processing facilities to a certain extent. List functions that are specializations of other functions already described have been omitted in the following comparison.

4.1.1 Basic functions

head, last, tail and init: These functions return the first element of the list (the "head" function in list processing languages), the last element ("last"), all but the first elements ("tail") or all but the last elements ("init"). These functions may not seem useful on its own for dataflow computing, but are often necessary to exclude elements from the output of a pipeline. A pipeline generates

one data item per clock cycle at its output, but complex pipeline system may produce intermediate results that should be filtered out before piping the output into the next pipeline or memory. An implementation on an FPGA can consist of a counter and an output that produces values dependent on whether the counter counts the first element, the last element, or an element in between. By repeated application of "tail" and "init" a stream can be shortened by more than one element.

length: The length of a stream with a finite number of elements is usually stored in a register in advance from outside of the pipeline. It can then be compared to a counter for the "last" and "init" functions.

append: The append function takes two lists and appends the second list to the first one. For dataflow computing in a pipeline, we use two inputs, one for each data stream. At the beginning, only the first input is enabled and its stream is copied to the output. After the first stream has been processed, the first input is disabled, the second input is enabled and the second input stream is copied to the output. A counter will be necessary to determine when to switch inputs.

Note that the combination of shortening a stream with multiple applications of "init" and the extension with "append" at its beginning by the same number of zero-initialized elements is equivalent to delaying a dataflow stream by introducing registers on the FPGA. This way, a stream element can be efficiently related to a previous one a fixed number of clock cycles ago. Compilers with automated pipeline scheduling also offer to advance one stream by a fixed number of clock cycles by actually increasing the latency of all other streams it affects.

4.1.2 Transformations

A list transformation creates a new list from an existing list. For a dataflow implementation this means that the input and output of the pipeline correspond to the input and output lists, respectively.

map: A mapping creates a new list from the input list element by element. The mapping is defined by a unary function that takes a list element as an input and outputs the new list element. The return value does not depend on previous input values.

If the mapping function is already implemented as a pipeline, we can create a transformation in the dataflow picture by just piping the input values through the pipeline. The output then produces the transformed data. Input and output data can be stored in external memory and accessed linearly. With DRAM, this enables us to read from and write to memory with maximum throughput using burst mode.

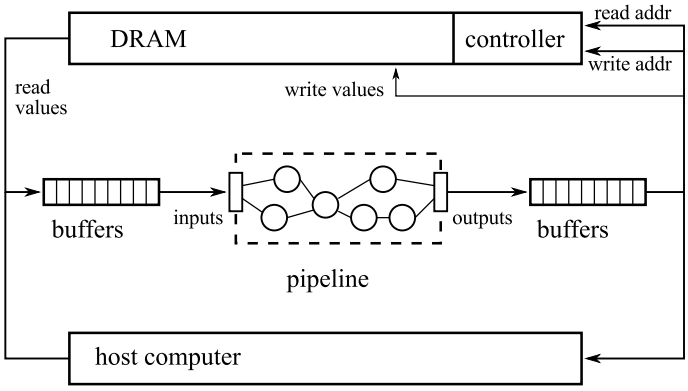


Figure 4.2: FPGA system overview with external memory and connections to the host computer.

reverse: The reverse function inverts the order of a linked list. In software a reverse operation is implemented by using the value of the first element as the last element of the new list, and adding consecutive elements from the input to the head of the output list.

An implementation with a pipeline alone would need to buffer all data items before it could start to produce the last input data item as the first output data item. A pipeline cannot reverse a large stream of data due to the limited BRAM storage available on FPGAs. However, an FPGA implementation can utilize the storage controller for writing the data stream in reverse by sending a stream of declining storage addresses to the controller. To make use of the DRAM burst mode the pipeline becomes a wire network that reads up to one burst per clock cycle, reverses it and presents the result at its output where it will be written back.

intersperse: The intersperse function takes a list l and a scalar value. It inserts the scalar value behind every list element except the last one. On the FPGA, the stream input is disabled every other clock cycle, and instead of forwarding the input value the scalar value is presented at the output. The scalar value can be fetched from a register. If needed the implementation can be combined with the implementation of "init" to ensure that the output list is shortened by one item and the last element at the output is the last element of l.

Nested Lists

The following functions from Haskell's Data.List library provide similar functionality to the functions above, but operate on nested lists. A nested list is a list that contain sublists as elements. If all sublists have a (small) fixed length we can widen the pipeline and process one sublist per clock cycle. For large sublists, a hardware counter can be used to keep track of the position of each element relative to the containing sublist. A common example for this use case are matrices or images where every row is stored as a sublist.

For sublists with a variable length, a serialization protocol has to be used. For sparse matrices, for example, often only the non-zero elements are stored, reducing the number of values significantly. A protocol will therefore suppress all zero elements in a row and then compress the column and row coordinates of non-zero elements. An overview about storage formats for sparse matrices can be found in [76].

intercalate: The intercalate function works like the intersperse function, but inserts a list k instead of a scalar value in between the elements of list l. For small lists, k can be stored in the BRAM. The input of the pipeline can then be halted for the clock cycles where k is to be inserted. Often, the values of k do not matter and are inserted only to give the downstream part of the pipeline extra clock cycles. We can then generate k from zeros or repeat the last value read from l.

transpose: Matrices and images are transposed by swapping rows with columns. All columns must have the same length, as well as all rows. Therefore, we do not need a special format to nest rows into the data stream, but can rely on counters to keep track of the row and column number of each data value. Like the reverse function, we cannot transpose the whole matrix or image in the pipeline and need to rely on the memory controller to re-order the data. However, if we do not serialize the data by saving row after row, but store it as a series of square sub-matrices of the size of one memory burst, we can utilize burst mode similar to the reverse function. The pipeline should then be as wide as a burst and transpose it in a wire network.

subsequence: To output all subsequences of a given input data stream with fixed length l, we can send it repeatedly to the FPGA and use a counter with bit width l to generate a bit pattern for every

repetition. The bits set to one in the counter value then provide a mask for which elements to output and which elements to suppress.

permutations: Like the previous function that re-orders the data stream, the permutations function relies on the memory manager. From a data stream of length l, it generates all $l!$ permutations. Donald E. Knuth's Algorithm L [77] provides all permutations of a multi-set in linear time, enumerated by lexicographical order. It enables us to provide one data item per clock cycle in an FPGA implementation. For short input sequences FPGA implementations exist that can provide an entire permutation at each clock cycle. [78].

4.1.3 Reductions

A reduction consumes a (sub)stream and calculates a single scalar value from it. In functional programming reductions that operate on lists are also known as folds. The name "fold" will only be used in this section as the term is also used in image processing for functions that produce a new image from a given one by applying a function to every pixel and its (limited) environment.

foldl, scanl: A left fold on lists applies a function to each element from left to right and uses the result of the previous application as a second parameter. A good example is an arithmetic accumulator with folding function $f(a,b) = a+b$, that calculates the sum of all elements when used with a zero starting value for the second parameter.

For streams, a left fold follows the natural order in which the stream elements are presented at its input. However, the applied function must be able to consume one element per clock cycle, which poses restrictions on the folding function. The dependence on the previous result creates an implicit loop with a latency that may not fit into one clock cycle. For summation, the accumulator will be covered in section 4.3.3. Other examples for fold functions are binary "or" to accumulate all ones in a stream of bit masks, $\max(a,b)$ to obtain the maximum value of a stream, or a predicate in combination with "and" to check if the stream contains at least one element that satisfies the predicate. The "scanl" function returns all intermediate results during accumulation and directly corresponds to the accumulator output on the FPGA. The "foldl" function return the final accumulated value only. Hence, on an FPGA the intermediate values have to be suppressed at the pipeline output similar to the implementation of the "last" function.

foldr, scanr: A right fold applies the fold function from the back to the front of a list. For dataflow computing, the stream therefore has to be reversed first. If the fold function is commutative and associative, a left fold can be used instead.

4.1.4 Generation

A list can be generated by repeating a constant or by running a loop that feeds the result of a function back into it as an argument. The concept is important in dataflow computing to generate auxiliary streams needed for computations on the input data.

repeat: The "repeat" function takes a scalar value as an argument and generates an infinite list by repeating it as a list element. This corresponds to the use of a stream of constants in dataflow computing, where the value can be hard-coded or stored in a register before the pipeline is started.

iterate, unfoldr: The "iterate" function takes a scalar value a and a unary function f to generate an infinite list. The first element is the scalar value, and the following elements are created by repeated application of f on the previous list element. If f is chosen to be the identity function "iterate" will be equivalent to repeat. In an FPGA pipeline the next trivial example is the counter without wrap and changeable initial value. It implements the "iterate" function with $f(x) = x + 1$. Similar to reductions special care has to be taken due to the implicit loop that must produce one value per clock cycle, preventing the use of some functions. "unfoldr" implements the same functionality, but additionally lets f control when the list generation is to be ended. On the FPGA it can be implemented by disabling the output of the pipeline.

cycle: This function generates list by repeating a given sublist infinitely. On an FPGA, the sublist can be stored in the BRAM if small enough and read repeatedly by connecting the output of a counter with wrap to the BRAM address port. For bigger sublist the same principle can be used, but the sublist has to be stored externally in DRAM. Due to the linear address pattern and missing dependencies between the cycles the function can be implemented by computing the linear address pattern a fixed number of clock cycles in advance, hiding the latency of memory access and making use of burst mode for DRAM.

4.1.5 Sublists

All sublist functions apply an explicit or implicit predicate function on each list element to determine which elements to return in one or more sublists and which elements to suppress. The order of the list elements is preserved. A hardware pipeline can implement this functionality by enabling the pipeline output for certain data items only, much like the implementations for "head", "last", "tail" and "init".

take, drop, splitAt: All functions take a list l and an integer n as arguments. The "take" function forwards the first n list elements and "drop" forwards all but the first n list elements. "splitAt" return two lists, the first consisting of the first n elements of l and the second consisting of all other elements. For a hardware pipeline all functions can be implemented efficiently by introducing a counter and comparing its output with the value of n to enable or disable the outputs.

takeWhile, dropWhile: "takeWhile" and "dropWhile" both accept a list and a predicate function. "takeWhile" returns a prefix of the input list until the predicate evaluates to false for the first time. "dropWhile" returns a suffix of the list, starting with the first element where the predicate evaluates true. Hence, both "takeWhile" and "dropWhile" produce disjunct lists when called with the same arguments.

An implementation for a hardware pipeline relies on a pipelined implementation of the predicate and an output that can be disabled. The one-bit output is then to be passed through a left fold operating with either $\min(a,b)$ for "takeWhile" or $\max(a,b)$ for "dropWhile". Finally, this data stream can the be used to control the output that forwards the input stream for all clock cycles when enabled.

span, break: Both functions take a list l and a predicate function as arguments and return two lists. The first list returned is the longest prefix of l where the predicate is true for "span" and false for "break". The remaining suffix of l is returned in the second list. The functions can be implemented in a pipeline similar to the "takeWhile" and "dropWhile" function, but with a second output that produces the otherwise discarded elements.

stripPrefix: "stripPrefix" takes two lists l and p as arguments. It checks whether p is a prefix of l, and, if true, returns l without the prefix. A pipeline implementation for streams of arbitrary length would need two passes over the data. In the first pass, p is checked whether it is a prefix of l. Depending on the result, l is stripped off the prefix in a second pipeline by enabling the stream output only after the prefix has been consumed.

For prefixes with a sufficiently small maximum length the stream of p can be buffered, and only one pass over l is needed. If the suffix does not need to be forwarded immediately and l is stored as an array in memory, an alternative is to let the pipeline only check whether p is a prefix of l and shorten l in memory by updating its start address and length afterwards.

group: This functions transforms a list by grouping adjacent elements with the same value into tuples and returning them as items of a new list. In a pipeline, every element must have the same width due to a fixed number of wires. Therefore, a protocol is needed to serialize the tuples. As all elements in any tuple have the same value, we can encode a tuple by storing the value in the upper bits and size of the tuple in the lower bits. To calculate the size we need to compare the present element at the input with the previous one and feed the signal into both a counter and the enable signal of the output. If the present and previous input elements are the same, the counter is increased by one and the output stays disabled. Otherwise, the counter is reset and its former value is sent to the output, alongside with the previous input element.

inits, tails: Both functions accept a list l as argument and return all possible prefixes ("inits") and suffixes ("tails") of l. On the FPGA, an efficient implementation of the same functionality can be achieved with the memory manager if l resides in DRAM, as the address pattern can be computed prior to memory access. The access pattern can be computed with two counters, where one counter creates the addresses and the second counter increases the start value ("inits") or the wrap value ("tails") of the first counter every time it wraps by one. The stream from memory then contains the desired prefixes or suffixes.

isPrefixOf, isSuffixOf: The "isPrefixOf" and "isSuffixOf" functions both accept two lists l and s and return whether s is a prefix or suffix of l, respectively. In hardware, "isPrefixOf" can be implemented by comparing the elements of l and s in a pipeline stage and accumulating the result with an "and" reduction to check if all pairs of elements are equal. A counter is used to emit the state of the accumulator when the stream of s has been consumed. For "isSuffixOf", a similar implementation can be used that skips the first $(\text{length}(l) - \text{length}(s))$ elements of l before it compares the remaining pairs. If s is sufficiently short, its elements can be held in either DRAM or BRAM storage.

isInfixOf: This function has the same signature as "isPrefixOf" and "isSuffixOf" and checks whether s is an infix of l. For short infix lists, s can be stored in $\text{length}(s)$ registers, where the nth register is compared to the stream delayed by n clock cycles. A pipeline would then first read and store s in its registers before processing l. A stream reduction then stores whether all registers values have been equal to their corresponding stream values at any clock cycle. The state of the reduction forms the return value at the end.

It is also possible to search for matches of regular expressions instead of infixes if the regular expression is known at compile time. A regular expression can be translated to a nondeterministic finite automation, which can be reshaped into a deterministic FSM [79] and implemented on an FPGA. The FSM consumes one value per clock cycle. It can therefore be embedded into a pipeline and scheduled without extra logic. The tranlation of regular expression into FSMs finds its limit in

the number of states required. It increases exponentially with the length of the (very convoluted) regular expression in the worst case.

4.1.6 Searching

elem, notElem, lookup: All functions take a list l and a value v as input. "elem" returns true iff v is an element in l, while "notElem" returns true iff v is not an element in l. "lookup" compares v to every first element in a list of pairs, and returns the second element of the pair of the first match. "lookup" can therefore by used to implement an associative array with a list of pairs where the first element acts as the key.

For a pipeline implementation of "elem" and "notElem", the present value of the input stream representing l is compared to v and the boolean result is fed into a "and" or "or" reduction. The last element of the reduction is then the wanted return value. The "lookup" function is similiar to the "elem" function, but with some extensions. The pairs of the input streams are encoded as the lower and upper bits. The comparator only compares the lower bits with v, and the upper bits are fed into a register that stores the value only when the return value of the reduction switches from false to true.

find: "find" takes a list and a predicate and returns the first list element the predicate accepts. For a hardware implementation a pipelined version of the predicate is needed. The list is forwarded to the pipeline output, which is only enabled when the predicate returns true for the first time. The control logic can be implemented by combining the output of the comparator with a stream accumulator based on the "max" function, which in turn is connected to an edge detection to enable the output only once at most. If a negative result must be signaled as well a second output can be added that submits the value of the stream accumulator in the last clock cycle.

filter, partition: "filter" takes a list and a predicate as arguments and returns all elements of the list where the application of the predicate evaluates true. In hardware, the list input is connected to the pipelined version of the predicate, and also forwarded to the output. The predicate filters the stream by controlling the enable signal of the output.

"partition" shares the same signature as "filter", but additionally returns a list of elements that cause the predicate to return false. The implementation can built on top of the solution for "filter" by adding an additional output that is also connected to the input stream and only enabled when the first output is disabled, creating two disjunct streams.

Indexing lists

All searching functions can be easily modified to return the position of the searched value instead of the value itself. To do so, a counter is needed that tracks the position of the present value in the data stream. When the value is found in the stream, the counter value is presented at the output.

4.1.7 Zipping and Unzipping

zip: The function "zip" takes two lists as arguments and returns a list of pairs, where each pair is composed of the elements of the input lists in forward order. A pair of values can be encoded in hardware by abstracting the wires of both inputs as a single entity. No logic is needed to implement the function in a pipeline.

unZip: This function is the reverse of "zip" and splits a list of pairs into two separate lists. Similar to "zip", no logic is needed to assign the wires to separate streams in a pipeline.

zipWith: This function accepts two lists much like "zip" and a binary function. The output is the resulting list of the element-wise application of the input lists. In a hardwire pipeline, a basic binary operator has the same semantics, and can be combined with other operators to implement more complex operations. Any pipelined binary operation is therefore a hardware implementation of "zipWith", and variants that accept more than two input lists can be implemented in a pipeline accordingly.

4.1.8 Set operations

Set operations treat lists as sets from set theory. A pipeline can only access a limited window in the data stream, and hence requires as an additional constraint that the elements of the input streams are to be ordered for set unifications, intersections and differences. The next section will explain how to sort lists in hardware if needed. The semantics of the comparator used for sorting must be the same for the set operator.

nub: The function returns a list where all duplicates have been removed from the input list. The same semantics can be implemented in a pipeline by forwarding the input stream to the output only if the present and the previous stream values are different. The input list has to be sorted beforehand.

delete: This function accepts a list l and a value v and deletes the first occurrence of v from l. In hardware the present stream value is compared to v. If both values are equal for the first time, the output is disabled, otherwise the stream is forwarded. The same control logic as for "find" can be used to ensure a deletion at only the first occurrence.

difference: The "difference" function accepts two lists l and m as arguments and returns a list that contains all elements of l that are not present in m. In hardware, both input streams have to be sorted. If both inputs present the same value, no value is forwarded to the output. If the value at the input of l is less than the present value of m, the value of l is consumed and forwarded to the output. Otherwise, the present value of m is consumed and no output is produced.

This function cannot be implemented in MaxCompiler as a pipeline due to the fact that the value of the read-enable signal for the inputs depends on the input value and MaxCompiler requires the enable signal to be known three clock cycles in advance. However, an FSM can be used instead that controls the inputs by comparing their values and checking for empty or stalled FIFOs in a single clock cycle. This implicit cyclic dependency between input controls and input values puts stress on the timing of the design because the comparator logic cannot be pipelined. On modern FPGAs a comparator can be run at 150 MHz for data streams with up to 64 bits, and at 100 MHz for up to 128 bits.

union: This function returns the unification of both lists l and m provided as arguments, returning a new list that contains all values that are elements of l, m or both. In hardware, both input streams have to be sorted, similar to "difference" and "intersect", and can be implemented as an FSM. If both inputs present the same value, both inputs read in the next value and forward the present value to the output. Otherwise, only the smaller value is consumed and forwarded to the output.

intersect: The function returns the intersection of two list: only elements that are present in both lists are part of the resulting list. In hardware, both input streams must be sorted, and the function has to be implemented as a FSM. The present value of both lists must be forwarded to the output

only if the present values are equal. If not, the input with the smaller element must be advanced and discarded.

4.1.9 Ordered lists

sort: This function takes a list as an argument, sorts it following a stable algorithm and returns the ordered list. The function cannot be implemented with a single pass through a pipeline for streams of arbitrary length, since the pipeline has only access to a limited window of data items at a time. All comparison-based sorting algorithms have a time complexity of at least $O(n\log(n))$ for n data items [80]. Hence, a pipeline would need to process the data in at least $O(log(n))$ passes.

A hardware implementation can make use of merge sort, which is stable and requires only few logic for merging. Similar to the implementation of the set operations, an FSM can be used that forwards the smaller value of its inputs to the output. A naive implementation would repeatedly read in the data from memory in blocks with a length of a power of two, merge them, and write them back to memory with the support of a custom address generator. After $\lceil \log_2(n) \rceil$ iterations the data would then be sorted. A more advanced hardware implementation on FPGAs speeds up the initial passes with a sorting network and then processes multiple merges in parallel [81].

insert: "insert" accepts a sorted list and a value. It inserts the value into the list and preserves the order. The function is a special case of "union" with a one-element list as second argument and can be implemented accordingly.

4.1.10 Summary

Functions that consume and produce one data item per clock cycle can be combined freely in a longer pipeline to form more complex functions. These are the functions "map", "scanl", "zip", "unzip" and "zipWidth". Functions that generate data streams can be used at the beginning of a branch of the pipeline and replace pipeline inputs otherwise fed from the host or DRAM: "repeat", "iterate", "unfoldr" and "cycle" are members of this class. Functions that remove elements from the data stream must be wired to the enable signal of the output and can be serially combined by feeding the individual enable signals into an "and" node. Examples are the "head", "last", "tail" and "init" functions as well as the sublist functions and all predicate functions. All other functions, namely the set operations "difference", "union" and "intersect", the sort function and "permutations" can be combined by using separate pipelines that are connected through buffer FIFOs with each other, the DRAM memory or the host computer.

For image processing the "map" function directly implements point functions on rasterized image data. Point functions modify each pixel independent of its neighborhood and are needed for color transformations. The second most important operation on image data is a fold that creates a new value for each pixel by applying a function f to its former value and the value of its environment. Depending on the function and the size of the environment, a fold can implement FIR filters and other convolutions on image data as well as erosion and dilation for morphological image operations [82]. On the FPGA, the data stream of pixels is delayed by multiple clock cycles to create streams that represent the pixels in the environment (see "append" and "init"). A pipelined version of the function f is then applied (see "zipWith") and calculates the new pixel values, one per clock cycle.

4.2 Identification of Throughput Boundaries

To successfully accelerate an algorithm with hardware support it must be understood in detail. Set-up and clean-up code must be identified and separated from the computational core of the algorithm, the bounds of size and frequency of the data streams between its distinct parts must be estimated or measured, and the inherent bottlenecks that limit its performance on CPUs must be considered as a starting point for acceleration.

Profiling the program gives quantitative data about the behavior of the software. It increases the knowledge about the run-time behavior and provides insight which functions are part of the computational kernel. The different load levels of the system's components indicate whether the speed is held back by the CPU or the memory and I/O system. The following paragraph lists the different performance bottlenecks, ordered by increasing latency. The latency of the individual components is shown in Fig. 2.3 on page 12.

compute bound: The program is limited by the speed of the CPU and would benefit from an increase in processing capacity. Programs that are compute bound have a low ratio of stalled CPU cycles during execution. The processor either accesses the memory and I/O subsystem seldom, the working set of data fits well into the fastest cache or both. If the underlying algorithm can be formulated as a pipeline a CPU-bound program can be accelerated by exploiting the massive parallelism of the FPGA hardware.

cache bound: A program that is bound by the cache would benefit from a larger cache. Depending on the access frequency, the program can be limited by the first level cache or one of the larger, but slower caches behind. On the FPGA, the performance of a cache-bound program can be increased by holding the working sets in BRAMs and registers. The data can then be accessed in parallel at the frequency of the computing components, given that the algorithm has been parallized before in one or more pipelines

memory bound: The bandwidth of applications that are held back by the speed of memory access is throttled by up to three orders of magnitude for random access when compared to cache-bound allocations. Random memory access that causes cache misses in the entire cache hierarchy must be eliminated as far as possible for a speed-up. For both FPGAs and CPUs the data layout should be changed to reduce the size of the working set, either by tiling the data and computing on the individual tiles separately in the cache, or switching from random access towards mostly linear access, taking advantage of the burst mode of DRAM.

Further, data can be compressed if it contains redundant information to reduce the size of the working set. Lossy or non-lossy compression virtually increases the memory bandwidth at the expense of the computing resources dedicated to compression and decompression.

I/O bound: The execution speed is bound by I/O if the program gets stalled when waiting for network or disk I/O. These operations can reduce the bandwidth by up to three orders of magnitude, too, when compared to a memory-bound program. Few options remain for acceleration with computing hardware alone: compression can be used similar as for memory-bound programs, and algorithms can potentially be changed to compensate a reduction in data bandwidth with better processing.

4.2.1 Profiling in Software

Acceleration is measured in terms of improvements to computing time. To identify the compute kernels of a program for acceleration it is therefore necessary to measure the time that is spent in every function. Software profilers rely on the compiler, which inserts additional instructions during

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 58.57      0.89     0.89     1998     0.45     0.64  tiff_image16_ref::subtr_and_update_bg(tiff_imag
 14.48      1.11     0.22 40919040     0.00     0.00  short const& std::min<short>(short const&, shor
 10.53      1.27     0.16 40919040     0.00     0.00  int const& std::max<int>(int const&, int const&
  7.90      1.39     0.12     1993     0.06     0.09  clfinder::find(tiff_image16_ref&)
  3.29      1.44     0.05        1    50.01    50.01  global constructors keyed to TypeString(unsigne
  2.63      1.48     0.04    26101     0.00     0.00  estimator::estimate(tiff_image16_ref&, roi, dou
```

Listing 4.1: Output of the software profiler gprof. Manipulations of entire images belong to the longest operations, while the utility functions $\min(a,b)$ and $\max(a,b)$ are called most often. Set-up code is called only once.

compilation that record when each function is entered and left at runtime. Profiling information can also be sampled statistically by inspecting the instruction counter regularly at the tick of a high-frequency timer interrupt. From the value of the instruction pointer the currently executed function can be traced back. Modern profilers use both method to generate an overview of program execution [83]. For languages such as Java that are compiled to byte code the virtual machine provides the recording infrastructure [84].

To identify the most time-consuming functions, the profiler subtracts the runtime of all callees from the runtime of the caller before ranking them. Otherwise, the main function would appear at the top with an execution time of close to 100%, while the actual computation is carried out by the much more interesting sub-functions. The functions that appear on the top then need to be examined individually to research the limiting factor of their execution time and their potential acceleration.

A section of an example output of the gprof tool [83] is shown in Fig. 4.1. The program that was profiled is the C version of the analysis for localization microscopy, presented in chapter 5. All functions listed where ported to the FPGA and process image data on the pixel level, except for the constructor for character strings. The cumulative time of a function also includes the time of the functions it calls, while the self time excludes it. Functions may benefit from acceleration if they run a long time, such as the `subtr_and_update_bg()` function that is applied to an entire image and removes its background, or because they are short, but called very often, such as the `min()` and `max()` functions that compute their mathematical counter-part. Finally, the constructor for character strings is called only once and belongs to the set-up of the program, making it unsuitable for dataflow acceleration.

Software profilers can also record memory allocations, system calls and other runtime data. While this information is crucial to acceleration in software, memory allocators and system calls to the operating system stay in the realm of the CPU and are not ported to a hardware pipeline. For hardware acceleration, a different set of data is more important to decide which code could benefit the most: bottlenecks in computing can be widened with MISD or SIMD parallelism, and bottlenecks in data access caused by cache and memory latencies are removed by pipelined data access with pre-known access patterns. Both kinds of bootleneck are largely invisible to applications in user space. Support from the operating system and the CPU hardware is needed to obtain these performance measurement with high accuracy.

4.2.2 Profiling the CPU System

Modern CPUs contain hardware counters in the Performance Monitoring Unit (PMU) that can be programmed to track performance events. Every time the event occurs, which can be a cache miss, a memory reference or any other performance critical trigger, the counter is incremented. When a specified maximum count is reached the counter is reset and a high-priority interrupt is emitted to notify the operating system [85]. During process profiling, these counters can be read through

```
Performance counter stats for './main data/07.tif':

  87161.268021  task-clock-msecs         #      0.992 CPUs
         12042  context-switches         #      0.000 M/sec
           388  CPU-migrations           #      0.000 M/sec
         20499  page-faults              #      0.000 M/sec
  234550320502  cycles                   #   2690.993 M/sec
  359160744267  instructions             #      1.531 IPC
     406552059  cache-references         #      4.664 M/sec
       6487996  cache-misses             #      0.074 M/sec

  87.826693930  seconds time elapsed
```

Listing 4.2: Output of the CPU profiler perf for an already optimized program. Few cache misses and more than one instruction per clock cycle indicate that a program is compute bound.

a kernel interface by user-space profilers and combined with the process context. By setting the maximum count to a low value, the performance event of interest can be monitored with high resolution and the locations in the source code that triggered the event is known through the value of the program counter. A high maximum count, on the other hand, reduces the overhead and avoids to distort the measurement. Tools like "perf" [86] for Linux adjust the maximum count during runtime such that a constant user-defined sample frequency is reached.

The capabilities of the PMU depend on the processor it is part of. However, all popular architectures support performance events to track the performance of the memory hierarchy and arithmetic computations. This information is crucial for the identification of bottlenecks and to classify the boundness of the application. With this information, an FPGA engineer can then already estimate the speedup of hardware acceleration.

In Lst. 4.2 the result of a run of "perf list" with default parameters is shown. The program profiled is the already optimized C version of the analysis for localization microscopy, which was profiled in software in the previous subsection. With the additional information, we can now determine that the hardware bottleneck by going through the list of possible bounds from bottom to top. An I/O bound program would show a large number of page faults, as would a memory-bound program. The data however indicates that the number of page faults is only 20499 during an execution time of 87.8 s. The next bound is the cache, which is missed in less than 2% of all cache references. Since last-level cache has 7% of the latency of memory (Fig. 2.3), the program must be compute bound. This is also indicated by the number of instruction per clock cycle (IPC), which is larger than one, and proved by the acceleration that was achieved with an FPGA system later. Further optimization with vector instruction, of course, could easily increase the number of IPC and make the program cache-bound.

4.2.3 Profiling Dataflow Designs

After the compute kernels have been identified and while the program is being ported to the FPGA, it is often necessary to also profile the execution in the reconfigurable hardware. When a statically scheduled pipeline produces one result per clock cycle its throughput can be calculated easily by multiplying the data width times the clock frequency, and its latency is given by the number of pipeline stages. However, pipelines often show a more complicated behavior. The pipeline can run empty if the input data is not provided fast enough by the host, the memory controller or another pipeline. It may also be forced to wait when one of its outputs stalls. Last, but not least, controlled inputs and outputs can be used to introduce cycles where the pipeline does not consume or emit values.

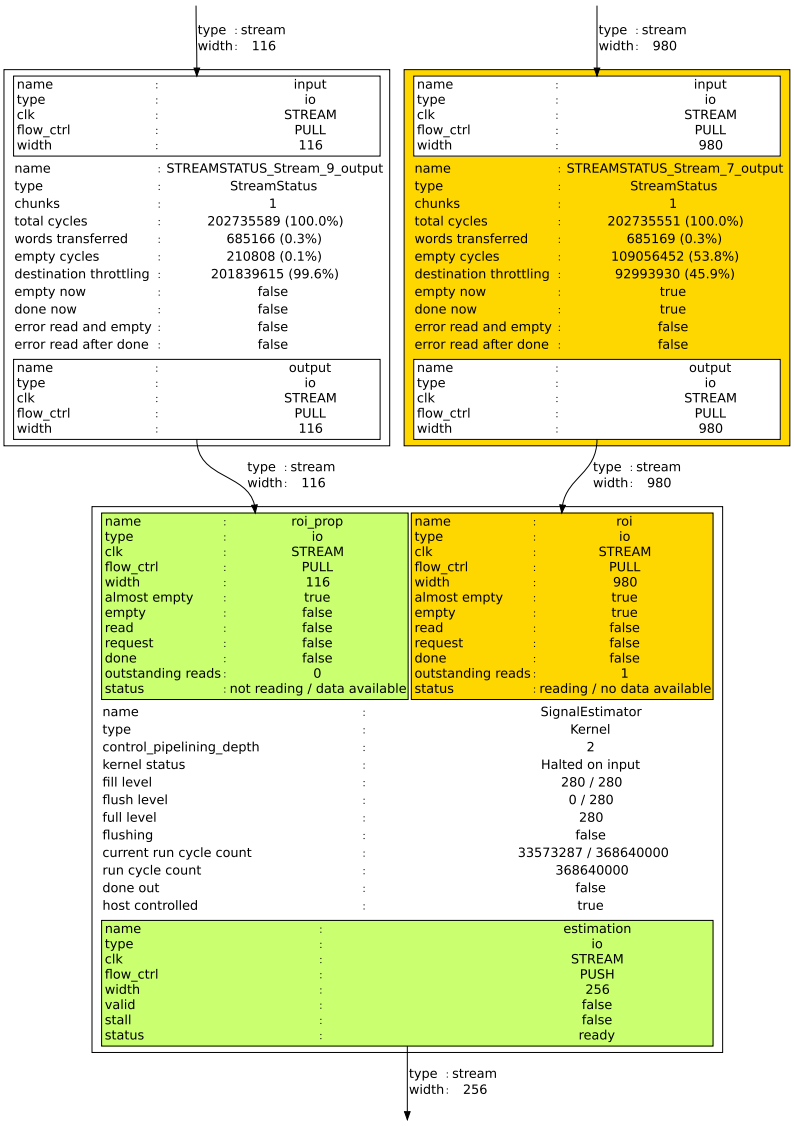


Figure 4.3: Profiling of a pipeline with MaxDebug. The values of the stream counters are shown after a run for a statically scheduled pipeline (kernel SignalEstimator) and for two of its inputs.

These potential performance bottlenecks can be detected by introducing counters that monitor when the pipeline was running empty, stalled and computing. The pipeline designer is then able to read out the according registers after a run and relate the values to the total number of clock cycles. If one of the pipelines is causing the drop in performance, it can be examined for multi-piping, and I/O bottlenecks can sometimes be widened by compression or a modified encoding.

Fig. 4.3 shows the performance counters obtained through the “maxdebug” tool for a pipeline built with MaxCompiler. The performance counters can also be embedded into a stream between kernels to monitor the fill level of FIFO buffers. The ratio of the amount of words transferred versus the total number of clock cycles informs about the utilization of the output and input that head and tail the buffer. The number of stalled clock cycles indicates a limitation at the receiver, and the number of empty cycles relates to a slow sender. Since FIFO buffers form the interfaces of statically scheduled pipelines, their performance numbers are valuable indicators to track down a performance bottleneck in hardware designs. In the final design these counters can be omitted to return their resources to the application.

4.3 Pipelining Imperative Control Flows

After the performance bottlenecks have been identified, we can start to translate the compute kernels of into a pipelined hardware design. In this section we will assume that the software was written in an imperative language like C, Fortran or C++ which are popular choices for high-performance applications. It can be shown, however, that all computer programs that run on a CPU can be ported to an FPGA: a CPU only consists of combinatorial logic and registers, and both elements are available in pipelined computing as well. Combinatorial logic can be build with stream operators, and registers are covered by stream reductions. Given that the FPGA provides enough resources, the CPU can then be instantiated as a soft core and the program can be run on it.

The aim of this section is to not only port software to hardware, but to port it with a focus on efficiency. A soft core processor cannot provide an efficient implementation due to an increased consumption of silicon area and a decreased clock frequency by a factor of about an order of magnitude when compared to a legacy CPU. We have seen in section 2.1 that general purpose hardware is elaborately designed to perform well with any kind of code. The complex control and cache logic that surrounds the arithmetic core of the processor becomes dispensable for FPGA hardware that is configured to execute one program only. The now spared silicon area can then be used to perform the actual calculation.

In the best case, the program can be translated to a single pipeline that runs in lockstep and contains few control logic. Here, almost all hardware resources can be utilized immediately to perform the actual calculations of the program. To accomplish a design that follows this principle as close as possible we need to examine how the semantical concept of imperative computing can be translated towards such a pipeline. The building blocks of imperative computing are sequences of statement expressions, branchings and loops.

Imperative programming contains further concepts [87]. Procedures and function calls maintain a structure. Both can be inlined if they do not perform recursive calls. Recursion can be mapped to sequential code with a loop and an explicit auxiliary stack. Object-oriented code, which is part of the imperative code family in a wider sense, relies on polymorphism. When needed for a calculation, the type identification is to be made explicit in a first step and function dispatch can then be performed with branchings based on the type information.

More advanced concepts in imperative programming are rarely found in compute intense kernels of applications. These are exceptions, which can be substituted with an additional return value for each function that encodes an arising exception. The concept is especially known from system programming in C where functions return an invalid value on error and set the global variable “errno”, and more recently from the Go programming language [88]. Continuations, which can be implemented in C with `setjmp()` and `longjmp()` to mimic tail call optimization, are even less common. Last, but not least, the “goto” statement is “considered as harmful” [89] to good software design and can be substituted with branchings and loops.

The structured program theorem [90] proofs that indeed any computable function can be re-written to only contain sequences, branchings and loops. Each control structure and its translation to a dataflow description will be examined below. For declarative languages, which form a superset of logical languages and functional languages a straightforward, low-level translation to the dataflow paradigm can be very difficult: both of them make mandatory use of lazy evaluation and garbage collection, two concepts that require a uniform memory layout in CPU implementations and are contrary to stream processing. On a higher level, the probably most popular members of both language families are Prolog and Haskell, which both make heavy use of processing of single-linked and immutable lists. When re-written accordingly, the processing of lists is then close to dataflow computing again, as we have seen in section 4.1.

```
float length(float x, float y) {
    float s = x * x;
    s += y * y;
    return sqrt(s);
}
```

Listing 4.3: `length()` with multiple assignments to `s`

```
float length(float x, float y) {
    float s = x * x;
    float s2 = s + y * y;
    return sqrt(s2);
}
```

Listing 4.4: `length()` following static single assignment form

Figure 4.4: A function in C to calculate the length of a 2D vector.

4.3.1 Sequences

The simplest building block of an imperative computer language is the statement expression. These are arithmetic and logical expressions that produce a value and optionally assign it to a variable. Contrary to variables in mathematics or functional languages, the values of variables in imperative languages can change after initial definition. Expressions may also have side effects. These cause other variables than the variables on the left-hand side to change its value, too. In C and related languages, side effects are caused by the increment (`i++`, `++i`) and decrement operators that alter the value of a variable before or after is has been read. Statement expressions can call functions that may have side effects as well on global variables or variables passed by reference.

We have seen that dataflow programming corresponds to the processing of immutable list. Mutable variables which are a core concept in imperative languages are not supported directly in the dataflow paradigm. Instead of altering a variable, its value is read or copied, fed into an operator and the resulting value is assigned in the initialization of a new dataflow variable. The semantic gap can be bridged by transforming sequences of assignments to static single assignment (SSA) form, where each variable is assigned a value at exactly one place in the source code. Variables V that are assigned multiple times are substituted with a set of new variables V_i (for $i = 1,2,3,\ldots$) until all variables are assigned only once [91] .

In listing 4.3, an example sequence of statement expressions in C is shown inside of a function that calculates the length of a 2-dimensional vector. The function `length()` computes the length of a 2D vector. In its body, the variable `s` is assigned a value twice, first at declaration and a second time in the line after. By introducing a new variable `s2` for the second assignment and by renaming `s` to `s2` in the following source code, the double assignment was removed in listing 4.4.

After the code has been transformed to SSA, the dataflow graph of the algorithm can be drawn immediately. The nodes in the graph represent the operators in the source code. The edges indicate the transfer of values and are directed from variable declaration towards variable readout. SSA ensures the acyclic property of the graph as every variable is assigned only once and cannot be read before it has been declared. For expressions with more than one operator the expression is mapped to the dataflow graph according to its syntax tree and defined by operator precedence. For the `length()` function, the dataflow graph is shown in Fig. 4.5. It is also the graph of the resulting pipeline.

The generated pipeline is able to process one vector per clock cycle and keeps the utilization of

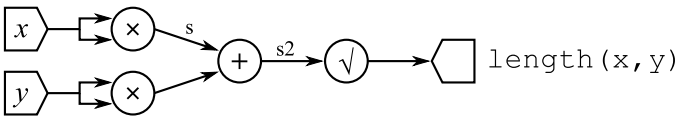


Figure 4.5: Dataflow graph and pipeline of the vector-length function.

each processing element at 100% as long as both inputs can provide the data stream and the output does not stall. Between the operators and inside of some of them registers are inserted to keep the longest path in the combinatoric logic short and the clock frequency high. These register increase the latency, but do not slow down the computation for fixed clock frequencies. Complex operators like the square root may have a latency of more than one cycle, but can still process one data item per clock cycle. Sequences of assignment expressions can therefore be implemented with very high efficiency on an FPGA. More complex pipelines than the one shown here will be discussed in chapter 5.

For the pipeline to run embedded in a hardware design, few extra logic has to be instantiated that stops the pipeline when one of the inputs runs empty while enabled or one of the outputs stall. In this case, the pipeline can be stopped by halting each register that connects combinatoric logic. On the FPGA the registers are built from flip-flops that usually already come with a clock-enable signal input. Setting all of them to the low logic level will therefore halt the entire pipeline.

The dataflow path reveals that a software implementation could make use of instruction-level parallelism: the multiplications that square x and y in the example do not depend on each other and can be calculated in parallel. A compiler could analyze the dataflow as we did and use a vector (SIMD) instructions, especially if the vector had a greater number of components. The hardware implements an extra level of parallelism as it also computes operations where the input of one operation depends on the result of a previous one (MISD). The implication leads to a different understanding of state in sequential code and pipelined dataflow computing. A pipeline computes l instances of the same computation in parallel, where l is the latency of the pipeline. Therefore, the parallelism grows with the number of pipeline stages, and C-like code can be parallelized by creating deep pipelines that represent the algorithm first (MISD) and afterwards multiplying the pipeline (MIMD) until most of the resources of the FPGA are occupied. Translated back into a software paradigm, the parallel computation of a sequence corresponds to list processing, where the list elements visualize the timely flow of data streams in storage space.

The transformation presented here from sequences of assignments to dataflow pipelines is straightforward and can be implemented in a compiler or software library to be performed automatically. In the next section we will examine branchings. The translation of branchings builds on top of the translation of sequences.

4.3.2 Branchings

Branchings in control-flow languages like C divert the flow of control depending on the result of a previously calculated condition. On the processor level, a branching is a conditional jump of the program counter. In this section, we will examine the conditional jump that does not lead to a loop in the flow of execution. In C and related languages, it is implemented as the if-else statement or the switch statement. The switch statement acts a syntactic sugar and can be expressed with multiple if-else statements. Other occurrences of branchings in C are the ternary operator "`?:`" and the conditional "and" (`&&`) and "or" (`||`) operators that evaluate the following expressions dependent on the result of the first one. These operators can be re-written with if-else statements and by introducing a new variable for the value of the sub-expressions.

Listing 4.5 shows the `abs()` function that returns the absolute value of a signed integer. If the input value of `x` is negative, it is inverted, otherwise, `x` is returned. The hardware implementation of the same function can be seen in Fig. 4.6. Contrary to the software implementation that will execute only one branch depending on the condition, both branches must be implemented in hardware and consume FPGA resources, and the correct result is selected afterwards by a multiplexer. The branching itself is done by wiring, and it is only the re-union of the data streams that requires LUTs to implement the multiplexer.

Note that the branching creates two meshes bordered by three paths in the dataflow graph, and all three inputs of the multiplexer must have the same latency to ensure the data arrives synchronized. In the example, a delay element was inserted before input "a". These delays, implemented using extra registers or FIFOs in BRAM, need to be added until each path has the latency of the longest one. As a side note, the predicate $(\cdot < 0)$ can be implemented as a single wire from the most-significant bit for two's complement encoding of x and will then require a delay as well. For more complex meshes in the dataflow graph the minimal number of delay registers can be calculated automatically from its directed acyclic graph (DAG) by optimizing a system of linear equations [92].

Using multiplexers has the disadvantage that all branches need to be implemented in hardware, but only one result is used later while the other intermediate result is discarded. For branchings with short branches like the shown `abs()` function this is tolerable. Multiple options exist when the overhead becomes too large.

- **Separate synchronous pipelines per branch:** Especially if the branching has one branch only, a second pipeline for this branch connected with a buffer can help save resources. Only if the condition evaluates true, the data is passed to the second pipeline. The speed of the second pipeline can then be adjusted according to profiling data from the condition. The second pipeline can either be clocked slower, yielding a higher freedom in circuit design, or make use of resource sharing and consume multiple clock cycles per data item. An example of the latter can be seen in section 5.1, where the second kernel requires 49 clock cycles to process one ROI. Loosely coupled pipelines move the scheduling of the design from static to a more dynamic one at the expensive of extra buffers and control logic.

- **Finite state machines:** Pieces of code that contain a large number of branchings, such as compiled regular expressions, parsers and lexers with long switch statements, can be implemented as a FSM instead of a pipeline and share resources between states. The mentioned use cases consume one data item per state transition and can therefore seamlessly integrated into a pipelined design. FSMs are also well-suited for branch-heavy control logic that would become convoluted in the pipeline paradigm.

- **Partitioning between host CPU and FPGA:** Code branches that are executed seldom or only at the start or the end of a calculation and are responsible for only a minor part in the total computation time can be left on the host CPU to save FPGA resources. These are typically setup and cleanup tasks, and code branches that are connected with low data bandwidth with the rest of the system.

- **Multiple bitfiles:** Code branches that are executed repeatedly in a loop, but where the condition evaluates to the same value for long sequences that last hundreds of millisecond, can

```
int abs(int x) {
    if(x < 0) {
      return -x;
    } else {
      return x;
    }
}
```

Listing 4.5: Description in C of the abs() function.

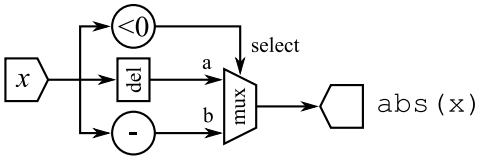


Figure 4.6: Dataflow graph and pipeline of the abs() function.

benefit from separated FPGA configuration. When the condition changes, the configuration is changed, too, and no resources are wasted for unused branches. Configuration, however, take tens to hundreds of milliseconds and is only an option if the condition changes seldom. Partial dynamic reconfiguration can decrease reconfiguration times.

- **Multiple FPGAs:** If the system consists of multiple FPGAs, different bit files can be used to specialize them for the multiple, but disjunct paths of the data flow. To fully utilize all chip resources, the option requires static scheduling and a constant ratio of the individual branch decisions in short time scales and may increase latency due to additional buffering.
- **Changes in the algorithms:** An FPGA design can greatly benefit from algorithmic co-design as shown in the application chapter (Chap. 5). Good sources for algorithmic alternatives with less branches can be found in the software world: compiler architecture avoids branchings, too, as they slow down the processor pipeline. The `abs()` function, for example, can be calculated without a branch with the functions "xor", "and" and subtractions alone [93]. A second resource are time-independent cryptography functions that avoid branches as well.

The options presented here for branchings still have to be chosen by hand. Only the static scheduling with multiplexers as described first can be automated by transforming the code to SSA. In SSA, the Φ function that assigns values to the variables at the end of the branching, dependent on the branch taken, is then substituted with a multiplexer [91]. Up to date, the best options can only be selected after profiling as described in section 4.2 and understanding the algorithm.

4.3.3 Loops

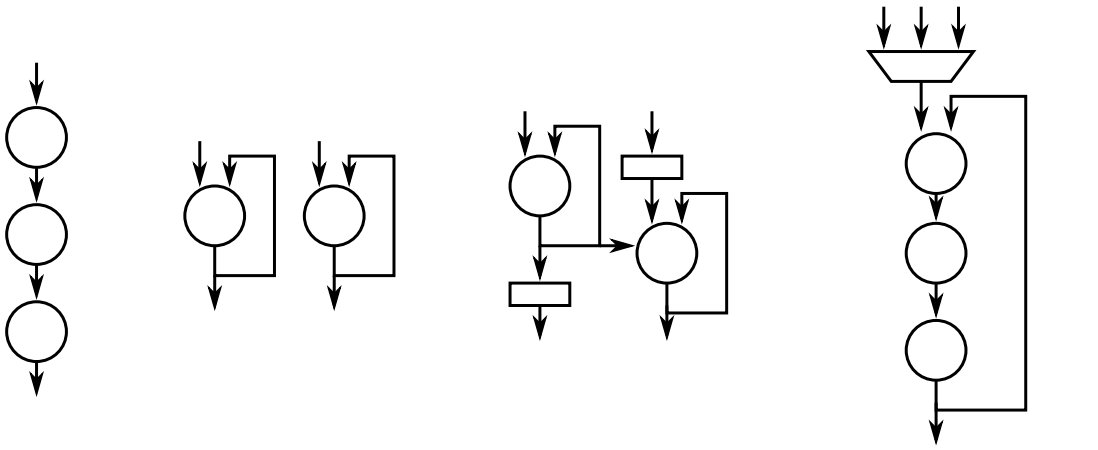
With sequences and branchings alone, the state in the registers between pipeline elements is flushed out when the pipeline advances. Hardware that needs to implement functions with an arbitrary long impulse response must make use of feedback loops to keep state in the pipeline. The output of an accumulator, for example, depends on all previous input values since reset, and therefore needs a feedback loop to add the current input value to the output value from the previous clock cycle.

We will first focus on loops that can be unrolled or where the loop body can be calculated in a single clock cycle. These loops are straightforward to implement and fit well into synchronous pipelines that process one data item per clock cycle. In the first case, the loop vanishes and the dataflow graph processes all iterations in parallel. In the second case, a loop with a latency of a single clock cycle for the loop body is constructed in hardware by simply connecting the output to one of its inputs. Loops with larger loop bodies cannot always be converted and will form loops with a latency of more than one clock cycle. These loops result in a more complex scheduling that is known as loop tiling, and will be examined last. An overview of all presented loop techniques is shown in Fig. 4.7.

Note that all long-running software contains loops in one way or the other in the control flow. Otherwise, a gigahertz CPU would traverse the instructions of a program with the size of multiple gigabytes within a few seconds and terminate. Therefore, the computational kernel of a long-running program must contain at least one loop.

Loop unrolling

Loop unrolling can be used if a loop has a fixed number of iterations. Given that the number is small enough to not exhaust too many chip resources, the loop can be unrolled to a sequence of statements, and the resulting acyclic pipeline can be laid out on the FPGA. The resource usage is determined by the number of resources consumed by the loop body times the number of iterations. Besides of resource constraints loop unrolling does not impose any restrictions to the source code



(a) Unrolling (b) Parallelization (c) Cascading (d) Tiling, Interweaving

Figure 4.7: Loop implementation techniques for static synchronous data flows. Circles indicate operations with a clock latency of a single cycle. These operations can form a loop by simply connecting the output to one of its inputs. Otherwise, the loop must be unrolled, dismantled into loops with single-cycle latency, or manually scheduled.

to be applied; moreover, information can be passed between non-neighboring iteration more easily than in up-rolled form. Loop unrolling removes the loop and therefore puts no constraints on the latency of the pipeline of the loop body. It comes with the advantage of maximum throughput and a simple integration with the other parts of the design, but may cause high resource usage for non-trivial loops.

An example can be seen in Listing 4.6. The function is known as bit population count. It counts the number of ones in the binary representation of an unsigned integer by applying a moving mask to the integer. As the number of bits in an unsigned integer is fixed, the number of iterations is known in advance and the loop can be unrolled. Data structures that are generated from the control variable `i` become constants, and as a consequence the masks in the example become a set of constants as well.

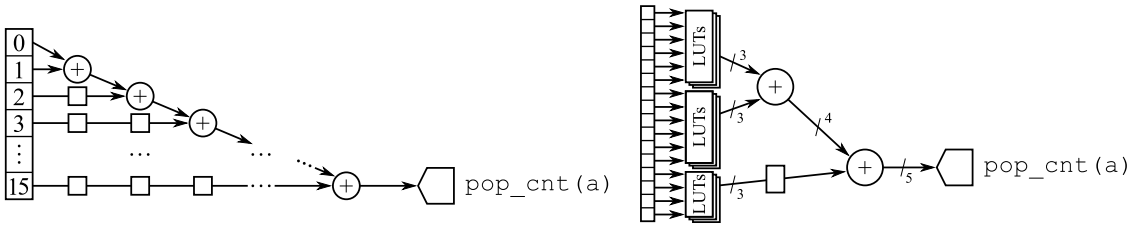
```
int pop_cnt(uint16_t a)
{
  int cnt = 0;

  for(int i = 0; i < 8 * (int) sizeof(a); i++) {
    int16_t mask = 1 << i;
    cnt += (a & mask) ? 1 : 0;
  }

  return cnt;
}
```

Listing 4.6: Bit population count in a loop in C++.

The corresponding pipelined hardware version of the unrolled algorithm is shown in Fig. 4.8a. It features 16 adders to calculate the bit population count, assuming standard integer encoding. Because addition is an associative operation the adders can be re-arraigned into a tree shape to save flip-flops in the delay registers (Fig. 4.8b. The lower stages of the adder tree have been substituted with three times three look-up tables, limiting the number of needed adders to only two.



(a) Linear implementation (b) Tree-shaped implementation

Figure 4.8: Pipelined version of the bit population count function in hardware. The tree-shaped implementation saves delay registers and adders.

Loop parallelization

Applications that process large amounts of data from memory following a certain access pattern or from I/O channels in a linear order usually apply the same processing steps to the data iteratively. In imperative software, the process is described using an outer loop or, for multi-dimensional data, a group of nested loops. In hardware, however, the outer loop is not immediately visible, as the I/O subsystem takes care of feeding data into and fetching results from the pipeline.

A loop in software affects its hardware counterpart wherever values are written in one iteration and read at a later one, which represents a special case of a read-after-write dependency. In this case, a feedback loop must be implemented that holds the updated value and routes it back to the pipeline stage where it has to be read. Loops in software will generally pass the updated value to the very next iteration. In a hardware pipeline, the affected stage will process the next iteration in the next clock cycle. Therefore, the latency of the hardware feedback loop is limited to a single clock cycle and must have both the input and output connected to the same pipeline stage. Hence, if a loop in software does not write to a variable (or any other storage location) after the same variable has been read in the loop body, the generated hardware will not contain a feedback loop. In this section, we will focus on loops that lead to feedback paths, since the former kind can be handled like a sequence of statement expressions.

Loop parallelization and loop cascading are two design patterns that help to re-write feedback loops such that they are compatible with the one-cycle latency constraint. If both techniques cannot be applied due to technical reasons or excessive resource usage, loop tiling can provide a solution that supports higher latencies.

Loop parallelization aims to untangle multiple independent read-after-write dependencies. Contrary to automatic loop parallelization with SIMD in software compilers [94], these loops can still be mapped to hardware. An example is shown in listing 4.7 that computes the arithmetic mean and the variance from a data series in a naive way. While the first for-loop merely accumulates the data for the calculation of the mean, the body of the second loop is more complicated: two subtractions, one multiplication and finally an addition on top of `var` certainly will not fit into a single clock cycle without severely slowing down the clock frequency of the entire pipeline. Moreover, the second for-loop depends on `mean`, the result from the first loop.

To calculate both loops in parallel notice that the variance of a data series can be calculated without incorporating the mean at every step. Instead, we accumulate the values of x_i^2 in linear time and subtract the value of $\bar{x}^2$ at the end (Eq. 4.4). The transformation removes the data dependency between the loops.

We can then accumulate x_i and x_i^2 in parallel to compute $\bar{x}$ and $\mathrm{Var}(X)$ (Listing 4.8). The final division by N and the subtraction of the squared mean $\bar{x}^2$ at the end can either be processed on the FPGA or on the host computer, given the data sets are large enough to not slow down the overall throughput. The hardware implementation then looks like Fig. 4.7b with integer adders as

operators and with an input stream of x_i for the first accumulator and an input stream of x_i^2 for the second accumulator. A counter could be used to only enable the output for the last result from the accumulators.

$$\bar{x} = \frac{1}{N}\sum_{i}^{N} x_i \qquad \text{(arithmetic mean)} \tag{4.1}$$

$$\mathrm{Var}(X) = \frac{1}{N}\sum_{i}^{N} (x_i - \bar{x})^2 \qquad \text{(variance)} \tag{4.2}$$

$$= \frac{1}{N}\sum_{i}^{N} x_i^2 - \frac{2\bar{x}}{N}\sum_{i}^{N} x_i + \bar{x}^2 \tag{4.3}$$

$$= \frac{1}{N}\sum_{i}^{N} x_i^2 - \bar{x}^2 \tag{4.4}$$

The example was taken from the application for localization microscopy that is presented in section 5.1 and common for center-of-mass feature extraction. The conversion of the formula for the variance was rather simple, but cannot be done automatically with todays tools. The translation of imperative software to hardware remains a process where the underlying algorithm of the software must be co-designed with the hardware.

```
pair<float, float>
mean_var(int x[], int N) {
  float mean = 0;
  float var = 0;

  for(int i = 0; i < N; i++) {
    mean += (float) x[i];
  }
  mean = mean / N;

  for(int i = 0; i < N; i++) {
    var += (x[i] - mean)
           * (x[i] - mean);
  }
  var = var / N;

  return std::make_pair(mean, var);
}
```

Listing 4.7: Calculation of the arithmetic mean and variance in C++.

```
pair<float, float>
mean_var(int x[], int N) {
  int x_acc  = 0;
  int x2_acc = 0;

  for(int i = 0; i < N; i++) {
    x_acc += x[i];
  }

  for(int i = 0; i < N; i++) {
    x2_acc += x[i] * x[i];
  }

  float mean = (float) x_acc / N;
  float var  = (float) x2_acc / N
               - mean * mean;
  return std::make_pair(mean, var);
}
```

Listing 4.8: After re-writing, both for-loops could be calculated independently.

Loop cascading

In some loops a read-after-write dependency may be present that cannot be removed by re-writing the algorithm. If the loops can still be simplified into loops with a body that is short enough be calculated in a single clock cycle, loop cascading can be applied. Here, one loop acts as a producer of a stream of values that is consumed by one or more secondary loops. Loop cascading can therefore implement algorithms in hardware that could not be ported with loop parallelization alone.

```
struct uint128 {
  uint64_t high, low;
};


uint128 acc_uint128(uint128 a[], int N)
{
  uint128 sum, prev = {0, 0};

  for(int i = 0; i < N; i++) {
    sum.low = prev.low + a[i].low;
    uint8_t carry = prev.low > sum.low ? 1 : 0;
    sum.high = prev.high + a[i].high + carry;
    prev = sum;
  }

  return sum;
}
```

Listing 4.9: An accumulator for unsigned 128-bit integers in C++.

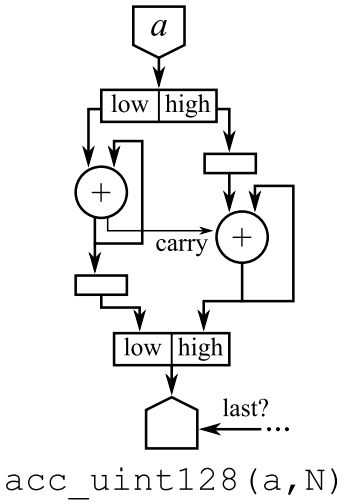


Figure 4.9: Accumulator implementation with loop cascading.

In listing 4.9 a software implementation of a 128-bit accumulator for unsigned integers is shown. Mainstream processors cannot sum 128-bit integers natively. The function performs the summation by spitting the integer into two 64-bit unsigned integers. After the lower halves of both integers have been added, a possible overflow is detected and the higher halves are summed alongside with the carry flag of the first operation.

On the FPGA, we certainly could instantiate a single 128-bit adder, but the most significant bit of the result will depend on all less significant bits in the bit vector of the inputs and will lead to a carry logic with long paths. To not reduce the clock frequency of the chip too much, it is advisable to similarly spit the hardware accumulator into two (or more) adders for high and low bits.

The read-after-write conflict in the function `acc_uint128` prevents us from splitting the loop into two independent parts in software. The carry bit is written by the first addition of the lower bits and read in the second for the higher bits. In hardware, however, the loop can be segmented into two accumulators, with a wire for the carry bit connecting both (Fig. 4.9). Delay registers allow the second loop to operate on the data one clock cycle later than the first loop, such that the carry bit is available in time. The enable signal at the output allows only the last result to be forwarded. It has been omitted from the figure for simplicity and can be implemented with a counter and a comparator, where the counter consists of an accumulator with its input set to constant one.

The example for loop unrolling (listing 4.6) could also have been implemented with loop cascading. The first loop contains a left shift and provides the bit mask, while a second loop counts all occurrences where the “and” operation of the mask and the input does not result in zero.

Both loop parallelization and loop cascading require logic that can provide its result with a latency of a single clock cycle. These are additions and subtractions of (sufficiently short) integers and fixed-point numbers, and all binary operations that operate locally, such as the binary primitives not, and, or, shift, rotate or related functions. Logical functions that are more complicated enforce a throughput-limiting reduction of the clock frequency owed to long critical paths.

Loop tiling

A loop in software cannot be transformed to a hardware loop without extra measures if the body of the loop must be transformed to a pipeline with $l > 1$ stages. A feedback loop in hardware would calculate l iterations of the loop body in parallel, and data could not be exchanged between these iterations, splitting the loop into l pieces. If loop unrolling cannot be used due to resource constraints and both loop cascading and loop separation are prevented by data dependencies, loop tiling can act as an option to generate an resource-efficient translation into hardware.

As an example, a floating-point accumulator that calculates the sum $\sum_i x_i$ in hardware cannot be implemented with a floating-point addition that would have a latency of only a single clock cycle. To add a floating-point number, several steps must be carried out in a linear manner. The summands must at least be normalized, the exponents must be aligned, the mantissas are to be shifted accordingly, and only then the mantissas can be added and the floating-point number is encoded according to the standard again. All these steps require one or multiple clock cycles in latency to keep the clock frequency at an acceptable level. On a Xilinx Virtex-6, an IEEE floating-point addition is synthesized to a pipeline that has a latency of $l = 12$ clock cycles for single and double precision. Feeding the result of the addition back as an input to create an accumulator would actually create an accumulator that splits the input data into l independent and interleaved sub-streams and will accumulate them independently of each other. An illustration of the issue can be seen in Fig. 4.10.

To solve the problem an auxiliary adder pipeline could be appended that sums the l interleaved sub-streams up again into a single result stream. The resource usage of such an adder would be linear with latency l, and therefore solutions have been developed to keep l small [95]. These solutions can only work for operations like addition that are commutative, as the order of execution is modified. Number-crunching applications, however, often apply a single instruction stream to multiple independent streams of data (SIMD) in nested loops and can therefore benefit from a technique called "loop tiling". The term was first coined by Maxeler Technologies for static scheduling in hardware [96] and is similar to loop tiling in software compilers. In software, a loop gets re-ordered and split into multiple sub-loops by the compiler to improve cache re-use and enable vectorization [97]. For hardware design, the loop is re-scheduled into work units of l independent calculations.

Equation 4.5 gives an expression to calculate the total gravitational force F_i of a mass m_i at position $\vec{x}_i$ as part of the n-body problem by summing up all contributions from all remaining masses m_j. We are interested in all force vectors $\vec{F}_i$ to calculate the change in momentum of all masses later in an exemplary simulation with discrete time steps. The problem arises in all simulations with long-range interactions like galaxy simulations or molecular-dynamic simulations.

$$\vec{F}_i = \sum_{j \neq i}^{N} \underbrace{G m_i m_j \frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|^3}}_{=:\vec{f}(\vec{x}_i, m_i, \vec{x}_j, m_j)} \qquad i = 1 \ldots n \tag{4.5}$$

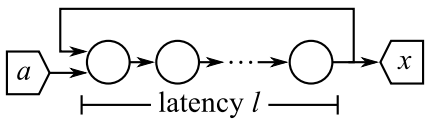


Figure 4.10: An accumulator implemented naively with a series of operations for floating-point addition and a feedback loop would calculate the sums of l interleaved sub-streams, with $x_j = a_j + a_{j-l} + a_{j-2l} + \ldots + a_{j \bmod l}$ instead of the desired result $x_j = a_j + a_{j-1} + \ldots + a_0$.

```
struct vector {
  float x, y, z;
  vector& operator+=(vector const& v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
  }
};

vector f(vector x1, float m1, vector x2, float m2) {
  return vector  ...     // terms omitted
}

void nbody(vector x[N], float m[N], float F[N]) {
  for(int i = 0; i < N; i++) {
    vector f_acc = vector();
    for(int j = 0; j < N; j++) {
      f_acc += (i == j) ? vector() : f(x[i], m[i], x[j], m[j]);
    }
    F[i] = f_acc;
  }
}
```

Listing 4.10: N-body calculation in C++ with nested loops.

The time complexity of calculating all $\vec{F}_i$ is of $\mathcal{O}(N^2)$ as for every calculation of F_i the properties of all N bodies have to be read from memory. In software, the problem can be solved by using two nested loops (Listing 4.10). These nested loops provide the opportunity to re-arrange the control flow.

The example reads the input stream composed of masses m_i and positions $\vec{x}_i$ twice with index i and j, applies a reduction on it with the overloaded C++ operator += and returns an output stream F_i. Listing 4.11 implements the algorithm with equal semantics, but the input stream with index i is now split up into tiles that are internally addressed by an offset. By adjusting the size of the tiles l we can then implement a processing pipeline in hardware with correct and resource-efficient static scheduling. The tiled input stream allows the processing of l independent reductions in the hardware loop. In software, an auxiliary loop at the end of `nbody_tiled()` is needed to store the results of a tile. In hardware, this loop is not needed and the tile is simply forwarded to the next pipeline or the memory controller.

The resulting hardware pipeline can be seen in Fig. 4.11. Like the C++ program it uses the variables i, j, *offset* and *tile* in the same sequence from a set of chained counters (not shown). The

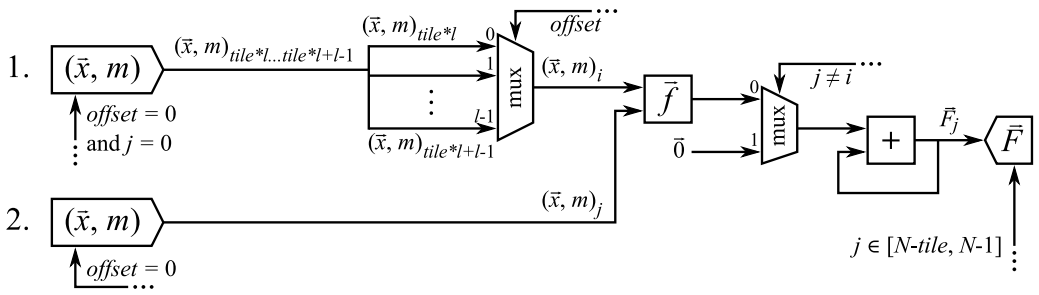


Figure 4.11: Pipeline of the n-body force calculation with loop tiling for a floating-point accumulator with latency l. The chained counters that generate the values of j, *tile* and *offset* have been omitted for clarity and follow Listing 4.11. Boxed operators have a latency greater than one.

```
void nbody_tiled(vector x[N], float m[N], float F[N]) {
  for(int tile = 0; tile < N/l; tile++) {           // loop over tiles
    vector f_acc[l] = {vector()};
    for(int j = 0; j < N; j++) {                    // loop over 2nd bodies
      for(int offset = 0; offset < l; offset++) {   // loop within tile
        int i = tile * l + offset;
        f_acc[offset] += (i == j) ? vector() : f(x[i], m[i], x[j], m[j]);
      }
    }

    for(int offset = 0; offset < l; offset++) {     // assign results of tile
      F[tile * l + offset] = f_acc[offset];
    }
  }
}
```

Listing 4.11: N-body calculation in C++ after loop tiling.

pipeline is fed by two input streams, e. g. from the memory controller, that provide the position and mass of the bodies. The first input receives every pair $(\vec{x}, m)$ once from memory, but l pairs at a time when *offset* and j are both zero. The pairs are separated by a multiplexer and fed into $\vec{f}$. The second input reads the data N times in single pairs of $(\vec{x}, m)$ when a calculation for a new tile is started ($\mathit{offset} = 0$). Otherwise, the former values are repeated by the register in the inputs. This way, the accumulator can compute l independent sums for a set of l forces and send the results to the output when done. Besides of the controlled inputs and outputs the extra logic needed for loop tiling is a multiplexer that selects the required data item from a tile and a more complicated set of counters for the control logic.

Note that for loop tiling, the input length N must be an integer multiple of the loop latency l. This property can either be satisfied by extending the input with null data, or by increasing the latency of the loop with delay registers. The introduction of delay registers is usually the preferred option since the throughput of the pipeline is maintained and spare registers are often abundant on an FPGA. The process is known as C-slow [98].

Instead of loop tiling, the data could have been reduced in memory as an alternative. Especially the forces could have been accumulated in the DRAM, and no feedback loop in the FPGA would have been needed. Each step in the calculation would have read properties of both bodies involved, calculated the force between them and updated the total force on the first body. However, the increments of the force at every clock cycles would have put more load on the memory interface. Loop tiling gives the hardware designer an additional design pattern that requires only one write operation for the reduced value, i. e. the total force on a body.

In general loop tiling can always be applied for nested loops with any reduction at the innermost level where the dependencies allow the interchange of both loops. After applying loop tiling, the application can be sped up further using the techniques described above, such as loop parallelization and (partial) loop unrolling.

Loop interweaving

Loop tiling requires at least two nested loops. If only one loop is present with a body that cannot be reduced to have a latency of only a single clock cycle, and the loop count is variable or large, non of the described techniques can be implemented and expensive additional hardware resources may be needed to counter the effects of l spitted sub-streams in the loop as described for a generic floating-point accumulator [95]. It can be advisable to search for a change in the problem domain

of the application that is to be ported to allow multiple independent input streams to be interwoven. Gustafson's law states that the user is usually interested in processing more data sets in the same time [30], which can, but does not have to require the acceleration of the individual calculation. Often, the original problem can benefit from or tolerate the parallel execution of l independent calculation. Hence, while technically very similar to loop tiling, loop interweaving was chosen as a different term to describe a change induced by hardware constraints in the problem domain.

As an example the pseudo-random generator for a constant bit width of the Java programming language [99] in Listing 4.12 can be modified to produce l interweaved sub-streams of random numbers. It is based on the Linear Congruential Method [100]. When implemented similarly as drafted in Fig. 4.10, it will produce a different stream of random numbers that are likely not to violate the constraints of the consumer of these numbers.

```
private long seed;

protected synchronized int next(int bits)
{
  seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
  return (int) (seed >>> (48 - bits));
}
```

Listing 4.12: The random number generation in Java qualifies for loop interweaving.

Finite state machines

As a last resort loops that can be computed by general purpose CPU can always be implemented as one or multiple finite state machines (FSM). Since FSMs rely on a sequential model of execution, any MISD parallelism has to be translated to a series of states. In the most extreme cases, the resulting system resembles a (small) CPU. Under the limitations of an FPGA, such as a clock frequency that is lower by an order of magnitude, FSMs do only qualify for application acceleration where they form a small part of the total design and transferring the data to the host CPU for computation would be considered too expensive. In the designs presented in this thesis, FSMs were chosen for control logic with nested loops that have a small resource footprint, but would require a complicated static dataflow design, and that would obstruct the actual dataflow logic with control logic.

4.3.4 Summary

Loops supposedly constitute the building blocks of sequential computing that need the most effort when porting them to an efficient hardware design. Unrolled, parallelized and cascaded loops can be freely combined with each other and the pipelines formed by sequences and branchings, as they all produce one data item per clock cycles. Tiled loops, however, have a different input and output pattern and require buffer FIFOs in between them for a free combination with other loops or acyclic pipelines.

The loops presented in software have all been "for" loops. The choice was motivated by the linear access of data elements that is simplified by the index variable. If the access pattern of the input data does not advance linearly, the data must be re-ordered before. Some examples have been given in section 4.1 and require the support of the memory controller.

Nested loops in software that do not qualify for loop tiling or one of the other dataflow techniques can be re-written into a single large loop with additional control variables that govern the execution of the inner loop's bodies. If the resources needed for these control loops exceed the FPGA budget they can be moved to a slower FSM with resource sharing.

Since loops in software impact hardware design decision on a wider part of the design space than (small) branchings and sequences, the identification of loops in a software program must be part of the hardware design phase right after the computational kernels have been isolated. The educational bottom-up presentation in this section hence is opposed to the top-down approach needed during the translation process.

4.4 Efficient Bit and Number Manipulations

A high-performance application can choose from multiple native encodings for numeric values on CPUs. Signed or unsigned integer encodings provide exact representations of the value and store numbers, depending on the required number range, in 8, 16, 32 or 64-bit words. Numeric values with a much higher range can be encoded as floating-point (FP) numbers that span many orders of magnitude, but limit the precision to a fixed number of significant bits. The CPU hardware, namely the ALU and the FP unit, are optimized for processing numbers with these encodings in hardware. Other encoding must be emulated, and are therefore used only cautiously by programmers.

In reconfigurable computing, all numeric operations are implemented with look-up tables and flip-flops, with the single exception of DSPs that reduce the resource footprint of multiplications. Hence, the encoding of numeric values is not predetermined by the hardware. Contrary to software applications, the most efficient design has the bit width of each stream of numbers reduced according to the minimal required precision and range. It is the task of the hardware designer to assess the required computing precision from the precision of the input values and the wanted precision of the output through error propagation or in simulation.

4.4.1 Encoding

The encoding of numbers has influence of the resources required to carry out arithmetic operations on the FPGA, but also on the amount of storage required and, as a consequence, the bandwidth in numbers per time that can be retrieved, processed and stored. An overview about the resources needed to perform basic arithmetics is shown in Fig. 4.12 for signed integers with 32 and 64 bits and floating-point representations with single and double precision. Single precision number representations have the same bit width as 32-bit integers and therefore the same amount of information. The same applies to the floating-point representation with double precision and 64-bit integers. However, the number range, the precision and the resources needed for basic arithmetic operations vary vastly.

Integers and Fixed-Point Representations

Unsigned integers are the natural representation for pixel values in a rasterized image. The width of an integer for color depth in popular image formats is defined as multiples of a byte, but an FPGA implementation can benefit from adjusting the width to the actual information content. Cameras for microscopy, for example, encode the intensity as a 16-bit integer, but commonly do provide only up to 12 bit of resolution. The extra four bits are zeroed and can be omitted without loosing information. As indicated in Fig. 4.12, reducing the bit width also reduced the FPGA resources needed for adding and subtracting by a linear factor, and by even more for multiplication and division [101].

The encoding of unsigned integers follows their representation in the binary system. For a bit vector of n bits, the range of values that can be stored is $[0, 2^n)$. For signed integers, modern FPGAs support the two's complement encoding best. It represents a negative number with absolute value a as the same bits like an unsigned integer $\overline{a}+1$ and encodes a range of $\left[-2^{n-1}, 2^{n-1}-1\right)$.

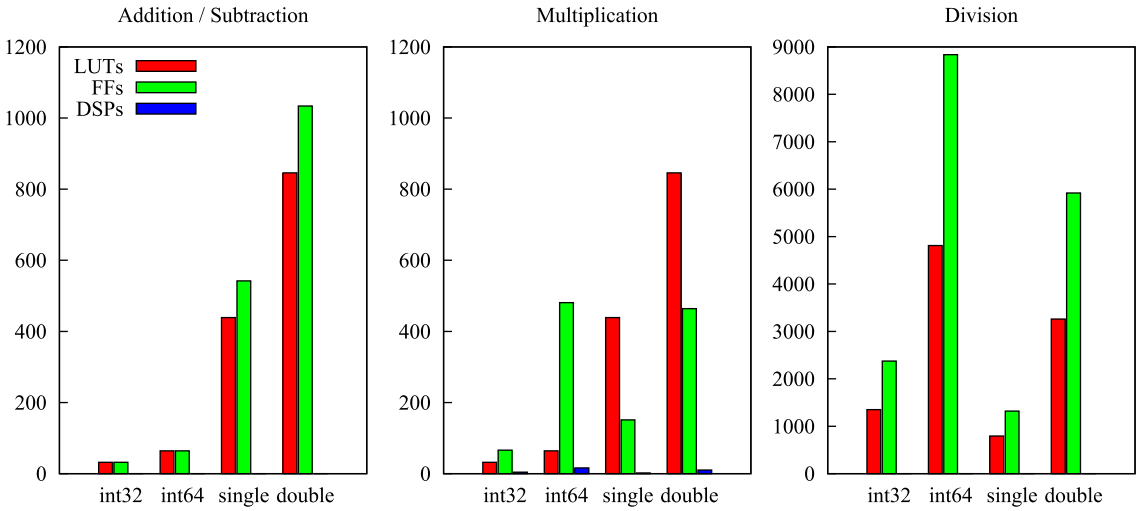


Figure 4.12: Resource usage for basic arithmetics with different data types measured on a Xilinx Virtex 6 with MaxCompiler. Addition and subtraction are cheap with integers and expensive with floating-point representations. Multiplication requires more LUTs for floating point, and division is by far the most expensive operation (note the change in scale). Virtex-6 FPGAs contain two FFs for every LUT on each logic slice.

The encoding allows integers to be added and subtracted with the same hardware as unsigned integers. Addition and subtraction are equally expensive, as subtraction can be reduced to addition by negating the subtrahend arithmetically first. The logic needed for negation consists of a binary negation and an increment by one, and can be absorbed by the look-up tables of the adder without occupying additional resources (Fig. 4.12).

To store fractional values, the integer representation can be extended naturally by adding additional bits after its least significant bit (LSB). The resulting fixed-point format with a fractional part of f bits can then hold multiples of 2^{-f} exactly and, depending on the number of fractional bits, approach real numbers with arbitrary precision. The bit layout is shown in Fig. 4.13a. The value of a signed number encoding with n integer and f fractional bits $a_{n-1}a_{n-2}\ldots a_0, a_{-1}a_{-f}$ is

$$v = -a_{n-1}2^{n-1} + \sum_{i=-f}^{n-2} a_i 2^i \tag{4.6}$$

Besides of integers and fixed-point number representations, a third class can be constructed by removing the lower bits of an integer. For values that are known to contain a fixed level of noise, the lower bits do not have to be stored or processed and can be omitted. The logic is similar as for fixed-point representations, but with a negative number of fractional bits f (Eq. 4.6).

Fractional numbers are cheap to add, subtract and multiply when compared to floating-point number representations, given that the range of values allows an efficient encoding. The encoding of numbers also affects all other arithmetic and logical operations. Integer and fractional encodings of numbers can carry out the following operations especially well. Compared to the von-Neumann architecture, where each of them requires a full cycle, these function can often be merged with other functions without affecting the timing of the FPGA design. An exhaustive overview about efficient bit manipulations of numbers in integer and fixed-point representations can be found in Hacker's Delight [93].

- **Shift:** Logical shifts and rotations by a constant amount require no FPGA resources and can be implemented by wiring alone. Arithmetic shifts that preserve the sign are also very cheap with few lookup-tables.

(a) fixed-point encoding of $2^7+2^3+2^1+2^{-2}+2^{-3}=138.325$

(b) floating-point encoding with exponent bias $b=63$ of $(-1)^0(1+2^{-2}+2^{-5}+2^{-6}+2^{-11})\cdot 2^{2^6+2^3+2^0-b}=1328.5$

Figure 4.13: Binary encoding of custom fixed-point and floating-point representations.

- **Logic functions**: The binary functions "and", "or", "not" and related can be implemented in few look-up tables. They operate locally only and do not require a carry chain.
- **Additions and subtractions:** Modern FPGAs contain special logic to implement carry chains for adders and subtractors. When compared to floating point, very few logic resources are required.
- **Multiplication:** When a number is multiplied by a constant in fixed-point representation, the multiplication can be broken down into a number of additions and shifts, one for each set bit in the binary encoding of the constant. If the constant is a power of two, the multiplication becomes a single shift operation. For general multiplication modern FPGAs offer DSPs to decrease the number of flip-flops and lookup-tables.
- **Truncation:** To avoid the introduction of a bias, fixed-point numbers must be rounded properly before truncation. Rounding half to even avoids a rounding bias and requires an adder to support rounding up, which again is cheap for fixed-point representations.

The canonical example of image processing, the convolution with a constant kernel, can be done efficiently in fixed-point encoding as the fractional part of the kernel matrix can be described with few extra fractional bits for common image manipulations. The multiplications by a constant can be split into adders and shifts, and the adders are supported well by modern FPGAs. Hence, for image processing fixed-point representations are often sufficient. Only where the values of variables span many orders of magnitude floating-point encodings provide better resource utilization.

Floating-Point Representations

The encoding of a custom floating-point format is shown in Fig. 4.13b. Numbers in floating-point representation are stored by splitting the normalized form $(-1)^s \cdot 1.T \cdot 2^E$ of a number into its significant bits T with sign s and an exponent E to the power of two for scaling. Like fixed-point encodings, floating-point encodings rely on a finite number of bits and can only store an approximation of all but a finite set of real values within their range. Whereas the maximum encoding error of a fixed-point encoding is the same for all values, the error of a floating-point encoding depends on the value of the exponent. Since many measurement processes produce a relative error, a floating-point encoding can store the results efficiently at the expense of a more complicated implementation of the arithmetic operations.

The arrangement of bits is ordered by significance and its parts are encoded in a way to allow comparison of numbers with the same hardware that is also used to compare integers. The first bit stores the sign of the encoded value. The exponent is stored next with a bias instead of two's compliment encoding to make its range non-negative. For an exponent with w bits, the value of the bias is $b=2^{w-1}-1$, resulting in a symmetric range for the exponent. Finally, the bits of the absolute value of the significand follow minus the first bit, which is known to be always one for normalized floating-point representation.

In the IEEE standard 754 for floating-point arithmetic [102], only certain lengths of significands and exponents are defined, with single (binary32) and double (binary64) precision being the formats most widely supported by CPUs and the only formats supported by graphics cards. The IEEE standard also defines a number of special values that encode ± 0, subnormal numbers, $\pm\infty$ and results of invalid operations (not a number – NaN). If these special values are known to not occur in an operation, the corresponding hardware support can be removed from the FPGA to save resources. Moreover, the format of the encoding can be changed at will. To change the range, the number of bits for the exponent can be modified, and a change in the number of bits for the significant alters the relative precision of the numbers. The standard defines the values of and operations on numbers encoded as floating-point according to the numbers of bits for significant and exponent and can therefore naturally be extended to also apply to custom floating-point encodings (Table 4.1).

During the development of an algorithm, when the correctness of the program is still unproven, double precision is the most convenient way to avoid overflows and incorrect results from insufficient accuracy. Afterwards, when speed becomes a concern, the number of bits for significand and exponent can be reduced, or the algorithm may even be changed to calculate with fixed-point encodings. For multiplications and divisions, the reward in chip area and therefore in power reduction is more than linear. On a Xilinx Virtex 6 the resource usage for lookup-tables and flip-flops almost drops by a factor of three for both addition and multiplication when moving from double to single precision. For divisions and square roots the ratio is even better and drops by about a factor of four [42]. With FPGAs, every stream of numbers can be adjusted on the bit level to only use the minimal amount of logic for precision, range and the special floating-point values shown in Table 4.1.

Alternative Encodings

Besides of floating-point and fixed-point encodings, other encodings for special demands exist. When processing numbers within a huge range, the logarithm of a number can be stored in a fixed-point format, similar to a floating-point encoding without exponent. The encoding of a value v becomes the fixed-point encoding of $\log_2 v$. Multiplication, division and exponentiation are mapped to addition, subtraction and multiplication and become much cheaper (Eq. 4.7). However, addition and subtraction in a logarithmic number system is resource-intensive. The encoding is commonly used for digital signal processing with high dynamic range [103].

$$\begin{aligned} \log_2(v \cdot w) &= \log_2(v) + \log_2(w) \\ \log_2(v/w) &= \log_2(v) - \log_2(w) \\ \log_2(v^w) &= w \log_2(v) \end{aligned} \tag{4.7}$$

exponent E (w bits)	significand T (t bits)	value
$\in [1, 2^w - 2]$		normal number, $(-1)^s \cdot 2^{E-b} \cdot \left(1 + 2^{2-t} \cdot T\right)$
$= 0$	$= 0$	$(-1)^s \cdot 0$
$= 0$	$\neq 0$	subnormal number, $(-1)^s \cdot 2^{emin} \cdot \left(0 + 2^{2-t} \cdot T\right)$, with $emin = 2 - 2^{w-1}$
$= 2^w - 1$	$= 0$	$(-1)^s \cdot \infty$
$= 2^w - 1$	$\neq 0$	NaN

Table 4.1: Floating-point encoding according to IEEE standard 754 [102]. Exponent bias $b = 2^{w-1}$, sign s. Single precision: $(w,t) = (8,23)$, double precision: $(w,t) = (11,52)$.

For image processing, the range of pixel values is limited and can be stored as an unsigned integer. Still, space can be saved by not storing the noise of an image. The number of photons N captured by a perfect sensor follows the Poisson distribution, where the noise has a width of $\sigma = \sqrt{N}$ [104]. Since the square root of a number consists of half as many bits as the number itself, half the significant bits of the integer encoding contains noise and can be omitted. To do so, the square root of every intensity is taken before it is stored [105]. For retrieving, the intensity is squared again. This saves half the bits required to store the maximum intensity and can be used for compressing images. The error Δg introduced by rounding the square root to the nearest integer with error $\Delta\delta$ between rooting and squaring is

$$g(N) = \left(\sqrt{N+\delta}\right)^2 \qquad \overline{\delta} = 0 \tag{4.8}$$

$$\Delta g = \sqrt{\left(\frac{\partial g}{\partial \delta}\right)^2 \Delta\delta^2} \qquad \Delta\delta^2 = \int_{-0.5}^{0.5} \left(\delta - \overline{\delta}\right)^2 \mathrm{d}\delta = \frac{1}{12} \tag{4.9}$$

$$= 2\left(\sqrt{N} + \overline{\delta}\right)\sqrt{\frac{1}{12}} \tag{4.10}$$

$$= \sqrt{\frac{N}{3}} < \sigma \tag{4.11}$$

The lost information Δg therefore only causes an additional error smaller than the value of the lower half of the significant bits of N. A 12-bit signal from an optical camera can be compressed to a 6-bit word. Compared to simply truncating the lower six bits, extracting the root consumes more chip resources, but preserves most of the information for small intensities. For multiplication, division and exponentiation the numbers can be processed directly the same way as integers and do not have to be squared again.

Besides of the presented encodings, a plethora of number and object formats exist. Especially for larger data structures like images, movies or matrices data can be compressed until it reaches a maximum density of entropy. Run-length and difference encoding can be implemented well with FPGAs, and also the lossy compression standards JPEG for images and MPEG for movies have been successfully ported to FPGAs [106, 107].

4.4.2 Dimensioning

The numeric data that arrives at the input of a hardware pipeline comes with a range and a precision. Both properties do not have to exploit the maximum number of bits of their encoding. It is, on the contrary, rather common to receive a stream of values encoded as floating-point with double precision where the distribution of values makes use of only a small fraction of the exponent and the significand. CPUs only support a limited set of number formats, and using a format not natively supported can lead to a performance penalty. On the FPGA, however, every saved bit will decrease the logic area. It is therefore advisable to trace the origin of the input data and reduce the bit width of the input values according to their true information content at the beginning of the pipeline, and further process only the minimal number of bits needed in the following stages. The resulting hardware will then consume less resources and will better meet timing requirements.

In the hardware pipeline itself, the range and precision grows or shrinks with each operation. In the following sections the required adjustments to the encoding are examined to avoid a loss in precision due to overflows or truncation.

Range

When the range of a value exceeds the capacity of a number encoding, a positive or negative overflow happens and the value is lost in its entirety. Even with floating-point encodings that represent these cases as plus or minus infinity, the value is trapped and will not approximate a real value again when applying basic arithmetics. It is therefore of importance to provide enough bits for the integer part of a fixed-point representation or the exponent of a floating-point representation.

For signed fixed-point encodings with n integer bits the range of possible values covers the values in $\left[-2^{n-1}, 2^{n-1}\right)$ and changes with each operation. The adjustment of the integer bits can be partially automated through a technique known as bit growth. It adjusts the number of integer bits for each operation to avoid underflows. On the level of an individual operation the bit growth for fixed-point encodings is shown in Table 4.2. For additions and subtraction this is a single bit to avoid overflows. For multiplications and divisions the integer part of a fixed-point representation must grow by a larger number of bits, depending on the number of bits of its arguments.

The bit growth of the integer part in Table 4.2 is given for the worst case, where the arguments occupy the full range of possible values. For chains of additions or subtractions with input values $a_1, a_2, \ldots, a_m$ with n_i integer bits, the number of bits n' for the integer part of the result only grows logarithmically, since the order of magnitude rises with the binary logarithm when adding the maximum value repeatedly (Eq. 4.12). The rule also applies when dimensioning accumulators for m summations.

$$n' = \max(n_1, n_2, \ldots, n_m) + \lceil \log_2(m) \rceil \tag{4.12}$$

For multiplications and divisions a similar rule does not exist. However, multiplications by a constant can, depending of their value, assign zeros to the lower bits of the fixed-point representation. If the constant factor happens to be a power of two, the multiplication or division degrades to a arithmetic shift operation, and as a consequence the bit size needed for integer and fractional part is kept constant. For multiplications with a constant that is represented by the sum of multiple powers of two the largest power defines the growth of the integer part, and the smallest power defines the shrink of the fractional part.

Extraneous bits in fixed-point numbers can also be identified by analyzing the data flow backwards, removing all bits that do not influence the result of the program. For example, a user may only be interested in the modulus of a number by a power of two, making the information carried by the higher bits redundant. An automatic forward and backward analysis of unused bits can be carried out automatically [108], but is not widely available yet.

For floating point, the range of numbers that can be encoded is defined by the number of bits of the exponent. With every extra bit, the range is extended far wider than by exponential growth. For single precision, an 8-bit exponent is long enough to store values up to about $\pm 10^{38}$, and the 11-bit exponent of double precision allows values that surpass $\pm 10^{307}$. It is therefore much less

operation	resulting integer bits	resulting fractional bits
addition	$\max(n_1, n_2) + 1$	$\max(f_1, f_2)$
subtraction	$\max(n_1, n_2) + 1$	$\max(f_1, f_2)$
multiplication	$n_1 + n_2$	$f_1 + f_2$
division	$n_1 + f_2$	∞
square root	$\lceil n_1/2 \rceil$	∞

Table 4.2: Maximum bit growth of basic operations for arguments a_1, a_2 with integer bits n_i and fractional bits f_i on signed fractional numbers. The results of divisions and square roots cannot be computed in general without loosing precision.

likely for an overflow to occur when compared to fixed-point numbers and usually sufficient to introduce a one-bit safety margin for the exponent after profiling the data. Hence, bit growth for the floating-point exponent is rarely needed, and the hardware designer can focus on the precision of the significand.

Precision

The precision needed for a computation depends on the uncertainty of its inputs and the allowed error at its output. If only the uncertainty of its inputs is taken into account and the highest accuracy is required for the output values, the number of fractional bits can be derived from Table 4.2 for fixed-point numbers. However, division, square root and other operations produce values that cannot be encoded exactly with fixed-point numbers, such as $1/3$ or $\sqrt{2}$. These results must be truncated after rounding.

Runding numbers must not introduce a bias, especially in iterative applications where a bias could add up and affect more then the least-significant bit. When rounding a fixed-point number with a fractional part of 0.5 to an integer, the number is therefore best rounded towards neither plus or minus infinity, but to the nearest even or odd integer. Though rounding half to even still introduces a bias towards infinity for even numbers and a bias towards minus infinity for odd numbers, the average of the distribution of results can be expected to be the same as for accurate calculations. Round to even is therefore also the standard mode for the significand of IEEE floating-point calculations. The error introduced by rounding and truncating a value towards a fixed-point encoding with f fractional bits is

$$\Delta^2_{\text{round}} = \int_{-2^{-f-1}}^{2^{-f-1}} \delta^2 \,\mathrm{d}\delta = \frac{1}{12} 2^{-3f} \tag{4.13}$$

The rounding error is added quadratically to the error that is propagated through the inputs. All errors are therefore shown to the power of two. The result of a function g with arguments $x_1, x_2, \ldots x_m$ and rounding, such as a multiplication without bit growth for the fractional values, will carry an error of Δy as shown in Eq. 4.16.

$$y = g\left(x_1, x_2, \ldots x_m\right) \tag{4.14}$$

$$\Delta^2_{\text{prop}} = \sum_{i=1}^{m} \left(\frac{\partial g}{\partial x_i} \Delta x_i \right)^2 \tag{4.15}$$

$$\Delta y^2 = \Delta^2_{\text{prop}} + \Delta^2_{\text{round}} \tag{4.16}$$

As a rule of thumb, the smaller error can be ignored if its absolute value is smaller by a factor of 10. The quadratic addition of errors ensures that the total error is only affected and increased by 1% by the smaller error. Therefore, a design should first focus on the errors introduced by the confidence of the input values to dimension the bit width of all numbers with a safety margin of about an order of magnitude, and only be reduced to the minimal amount of bits through simulation and deductive error propagation after the correctness of the implementation was shown.

For addition and subtraction, error propagation (Eq. 4.16) adds the absolute errors of the operands quadratically. For multiplication and division, the relative errors of the results are added to get the relative error of the result. Hence, fixed-point number encodings support the error characteristic for values that are dominated by a constant absolute error. For values that have

an error best described with a constant relative error, floating-point numbers are a more natural representation for the uncertainty.

A normal floating-point number with t significant bits has a precision limited to f bits, and its relative error $\frac{\Delta x}{x}$ is similar to the absolute error of a fractional number in Eq. 4.13.

$$\frac{\Delta x^2}{x^2} = \frac{1}{12} 2^{-3t} \tag{4.17}$$

The number of fractional bits of a floating-point encoding defines the relative accuracy of a number. It allows for a wider variation of the number of bits during resource optimization than the number of bits for the exponent, where two bits separate the occurrence of value-destroying overflows to a waste of bits.

As a summary, choosing an encoding should start with the range of the values first: very large ranges cannot be expressed well with fixed-point encodings. For image processing, sensor data from image sensors is generally restricted to 12 or at most 16 bits and very suitable for fixed-point encodings. Operations that increase the range of the result by more than a constant number of bits, such as multiplication, division or exponentiation may require a cast from fixed point to floating point. Once the design avoids overflows, the precision can be reduced such that the accuracy of the output is still maintained.

4.5 Customizing Memory Access

Configurable hardware cannot only be customized in terms of computational operations, but also how it accesses data stored in memory. The performance of an algorithm that suffers from a memory bottleneck on a CPU system can be improved on an FPGA if the access patterns of the data flow from and to memory can be adjusted to leverage its parallel capabilities. After profiling and determining which memory operations hold back the performance of the CPU implementation, the following options are available to manage state on an FPGA system.

- **Registers:** The flip-flops embedded in every logic cell of an FPGA can be combined to form registers that hold words of information. Like CPU registers, their content can be read or written at every clock cycle. Moreover, they can be accessed all in parallel and read and written at the same time, increasing the number of simultaneous register transfers immensely. Available FPGAs offer up to half a million of logic slices with eight flip-flops each, resulting in more than 60 000 64-bit registers. Although most of them are used as pipeline registers for dataflow computing, registers are still abundant on an FPGA when compared to the 16 general-purpose CPU registers of the AMD64 architecture [109]. Multiple registers can be combined with multiplexers to form small areas of addressable memory at the position where the data is needed and can be embedded into a seamless dataflow pipeline.

- **BRAM:** For larger amount of data up to the order of a megabyte most FPGAs include BRAM. It is implemented on its own and distributed in portions of multiple kilobytes on the chip to save resources, especially flip-flops. Like memory build on flip-flops, BRAM can be read and written word by word at the same time at two independent addresses. It offers an efficient way to implement on-chip caches and dynamic stream navigation within a sliding stream window.

- **On-board DRAM:** For large amounts of memory with multiple gigabytes of data DRAM memory can be added on-board. The data is accessed through an instantiated memory controller on the FPGA. For maximum throughput, DRAM must be operated in burst mode, with a burst consisting of at least 4 words [110]. Also, the address pattern must be known in

advance to mitigate the read and write latencies. Unlike BRAM, DRAM does not decrease the maximum clock frequency when its size is increased.

- **Host memory:** For FPGAs that are connected with a host computer, typically through the PCI Express 2.0 bus, additional memory can be provided as shared memory within the address space of the host system. The additional bus increases the latency and decreases the throughput, but can conveniently be used to exchange information between host system and FPGA card.

The FPGA memory infrastructure makes copying data cheap because it can be parallelized massively on the register and BRAM level. If memory access to external memory is needed, the algorithm should be re-written to follow an access pattern on the data that is known in advance and can therefore be computed earlier by at least the combined latency time of the DRAM and memory controller. Since this requirement often implicates a major re-modeling of the data flow of the algorithm, a C model should be created for verification. It can afterwards act as a semantic bridge between the initial software application and the hardware port.

4.5.1 Memory Layout and Access Patterns

In C, C++ and related languages, state is either stored on the stack or the heap. Access to the stack is limited to the topmost stack frame of the current function. On the heap, memory is abstracted as an almost infinite, linear memory space where chunks of memory for structs and objects can be obtained at runtime through a memory allocator. We have already seen that a stack can be removed by inlining and moving all variables to the heap, or, for recursive functions, by using an explicit auxiliary stack on the heap. It comes with the advantage that all stack frames are of the same size and type for most kinds of recursion, simplifying custom memory access. Since the current stack frame moves only in steps of one unit, it can be cached in BRAM when it must be stored in DRAM due to its size.

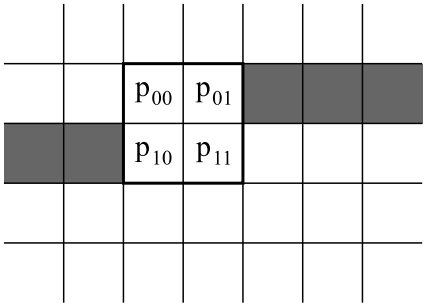
Variables stored outside of recursive functions can be absorbed by the pipeline as described in section 4.3. For large objects, however, the pipeline can become too wide, and the data is better stored in addressable memory to save resources. In the following sections the difference between arbitrary addressable on-chip memory and burst-oriented DRAM will be described with more detail, as well as the implications when porting software to hardware.

4.5.2 On-Chip Memory

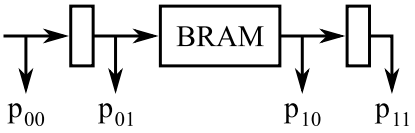
Memory on the FPGA that exceeds a few bytes can be implemented with BRAM and addressed with every clock cycle. BRAM offers two ports, so it can be read and written at the same time. This feature makes it blend in well into a statically scheduled pipeline, where it may consume and produce one word per clock cycle. In image processing, BRAM is used as line buffer (Fig. 4.14) or to store any other data temporally. On-chip memory enables to store a halo of the current stream position.

In software programs where an operation is executed during the traversal of a data set, BRAM can be used to store data that is local to the current iteration. Doing so, convolutions on data can be generalized and implemented efficiently with a dataflow design on an FPGA. For the background calculation in localization microscopy, an entire background map for the current picture is kept in BRAM (see Sec. 5.1.5).

BRAM allows for the addressing of individual words and can therefore also be used where the address offset to the current stream position is dynamic. An example can be seen in the accumulator in section 5.2.4 that uses BRAM to build a projection of a 3D volume with only weak guaranties regarding which individual accumulator will be increased next.



(a) Image convolution with buffered pixels.



(b) Hardware implementation with BRAM and two delay registers (address counter omitted).

Figure 4.14: Line buffering. The gray pixels must be kept in the line buffer for convolution in x-direction. The buffer can be integrated into a statically scheduled pipeline using dual-port BRAM.

In software, it can be difficult to immediately decide whether BRAM should be used. Indicators are the usage of small arrays of a few kilobytes that are accessed randomly, or limited offsets to the current index position in a for loop.

4.5.3 Off-Chip Memory

DRAM storage on the FPGA board is organized in rows and columns, similar to a matrix. To access a byte, the row must first be selected by the controller before the data can be read in a second step. Both operations have a latency of at least 10 ns [111]. The memory controller can pipeline these operations to a certain degree, but a random memory access still consumes at least 7 ns, limiting the access frequency for random reads to less than 143 MHz. For sequential access, much higher transfer rates than 100 MB/s are reached. Here, data is read in bursts, where the selected row is accessed in large chunks of at least four words. In theory, up to 17 GB/s of data can sequentially be read or written per module.

We have seen in section 4.1 that algorithms that can be reduced to list processing are especially well-suited for dataflow implementations. In general, algorithms that traverse data linearly can be implemented with maximum throughput from and to external DRAM. For applications that do not, data access should be re-arranged if possible.

Matrix transposition, for example, requires line-wise read access and column-wise write access at the same time. However, if the storage format is changed from a simple line-by-line format towards a collection of sub-matrices, with a sub-matrix having the size of one or multiple DRAM bursts, both operations become compatible with the coarse-grained access to DRAM. The sub-matrices are then transposed in parallel by wiring, and the address generator transposes the arrangement of the sub-matrices. For the acceleration of 3D electron tomography, the algorithm was changed to facilitate linear BRAM access for the 3D-volume it reconstructs.

The addressing of DRAM in units of a burst puts an additional constraint on programs that handle data. On the CPU, the cache-hierarchy and the memory controller hide the characteristics of DRAM. However, even on a CPU truly random access to the main memory will mostly miss the cache and limit the performance by more than an order of magnitude when compared to a program that hits the cache every time (Fig. 2.3). It is therefore advisable to avoid random DRAM access on CPU systems as well.

For data structures that are more complicated, such as maps or graphs, internal and external memory can be mixed. Meta information such as the management structures of a memory allocator can be stored in BRAM, where it is available with low latency and can therefore be modified much

faster and, if needed, in parallel. The actual content remains on external memory in chunks of multiples of a DRAM burst, similar to the various caching systems on a CPU system.

4.6 Summary

In this chapter, the transition from control flow code to a dataflow description was presented. Due to the different nature of both descriptions, an understanding of the algorithm that is to be ported is crucial on all levels. To achieve a maximum acceleration, not only must the building blocks of a sequential program be mapped to pipelined hardware, but also the higher levels. The intended number range and accuracy must be understood to save resources, and the hot spots of the algorithms are to be identified to partition the application between dataflow hardware and the parts that stay on a CPU.

The wanted gain in execution speed can then stem from the following changes to the program.

- **Massive parallelism:** Custom hardware allows for SIMD as well as for MISD parallelism and combinations of both. In the best case all stages of the resulting system of pipelines process data at each clock cycle, and little resources are occupied by auxiliary task such as control logic and caching. The transfer of values in registers and values can be parallelized massively.

- **Simplified hardware:** On a CPU, every piece of hardware is built for the most general use. Arithmetic units are designed to calculate on the full range of random inputs for the given and fixed encoding, and the CPU is expected to perform well on arbitrary programs. For custom hardware, the program is known in advance, and all logic that is not needed can be omitted, giving space to increased parallelism. The saved logic can be found on all levels, from saved caches down to constant propagation for basic arithmetic and logic units.

- **Reduced level of abstraction:** In custom hardware, abstractions such as virtual memory, function calls, call indirection and the entire software stack including the operating system are removed in for a pipelined hardware design. The missing instructions save further hardware resources for the actual calculation. They must be compensated by the high-level development tools to not impair development times and the flexibility for future adjustment.

Reconfigurable hardware benefits from these advantages, but is also impacted by a higher silicon footprint for single gates and a clock frequency. The next chapter will present two example applications from the research field of biomedical image processing and reconstruction to research the true potential of reconfigurable computing in these areas.

Chapter 5

Biomedical Image Processing and Reconstruction

In this chapter two applications and their implementation with the MaxCompiler library on reconfigurable hardware will be presented. The first one, image processing for localization microscopy was fully ported and its accelerated algorithms were in use at several research sites at the time of the writing. The second, image reconstruction for electron tomography, was also implemented on reconfigurable hardware and is shown to be faster than its counterpart on a state-of-the-art graphics card.

5.1 Localization Microscopy

Small objects like biological cells can be made visible to the human eye with a light microscope. It produces images of high contrast in the spectrum of visible light and the result can be seen immediately without further processing. For data storage good optical cameras are available. Even live cells can be observed and multiple techniques exist to dye otherwise invisible structures. The limit of conventional light microscopy is reached when the structures become smaller than the wavelength of the light source. Due to physical limits, a point in the object is mapped to a blurred spot in the image by the microscope, and spots close-by cannot be distinguished any more from each other. For visible light, the limit is above 200 nm, and hence the resolution is too coarse to image nanostructurs in biological cells that extend to few nanometers.

Localization microscopy can separate these close-by spots optically to a certain degree and increase the resolution by a factor of ten, depending on the quality of the recording. Localization microscopy relies on the preparation of the object with fluorophores that switch stochastically between different optical states over time when illuminated with laser light. The different states, e. g. bright and dark, are used to distinguish close-by fluorophores. A movie of the blinking fluorophores is recorded and the spots are later examined individually frame by frame. Typical image stacks consist of thousands of frames and span several minutes until the fluorophores are bleached out. Afterwards the center of each spot in the recording is obtained by fitting its signal distribution and the positions are plotted to create a new image of the object with superior resolution. An image that compares conventional microscopy with localization microscopy can be seen in Fig. 5.1. The increased resolution allows the diagnosis of various diseases, such as the analysis and identification of specific kinds of breast cancer [112].

The data processing on the recording, including background removal, spot finding and feature extraction, was implemented on an FPGA as part of this thesis. The existing algorithms were first re-written to be feasible for dataflow computing on reconfigurable hardware and then ported and

integrated into the workflow of the microscope.

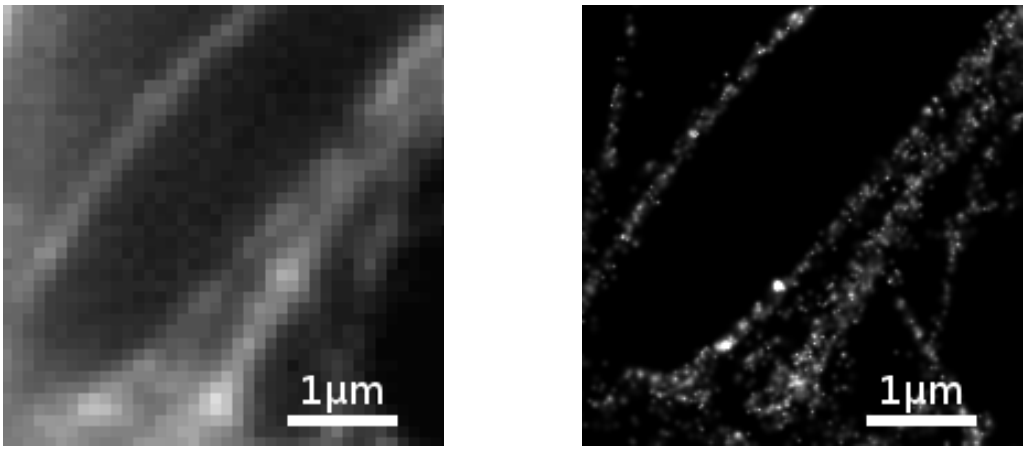
History

The first preserved written records about optical devices capable of magnifying small objects were passed down from the ancient Romans. Lucius Annaeus Seneca described how to fill glass bowls with water to built a first predecessor of the microscope in the first century AD. The availability of glass lenses in the 16th century led to the development of microscopes with multiple lenses that provided higher magnification factors. Christian Huygens invented the first microscope with a chromatically corrected system of lenses that improved image quality further in the late 17th century when lens manufacturing advanced further.

The construction of microscope stayed a craft led by experiment and experience until Ernst Abbe and Carl Zeiss developed the physical foundations of optical microscopy that arise from the wave-like nature of light. Today, the resolution limit for classical optical microscopy is know as the Abbe diffraction limit [113] and its mathematical formulation supports the creation of microscopes that achieve a maximum magnification of about 200 nm. Many important discoveries have been made since then with the microscope as a tool, including the analysis of bacteria that cause plagues, pox and tuberculosis or to gain insights into human cells.

To further increase our knowledge about the microscopic details of biological systems an ever increasing resolution was desired. Since the Abbe diffraction limit $d = \lambda/(2\,\text{N.A.})$ is proportional to the wave length λ, ultraviolet light can be used instead of visible light to increase the optical resolution. The optical aperture N.A. of the microscope, which also defines the resolution limit, can be improved by immersing the object into oil with a high diffraction index.

The nanometer and even picometer scale became directly accessible with electron microscopes. Louis de Broglie's discovered the wave-like nature of electrons [114] with wave lengths a hundred thousand times shorter than visible light in 1924. A beam of electrons in a vacuum can be bent with magnetic and electrical fields as a substitute for lenses. Contrary to light microscopy, however, the object has to be prepared for the high energy of the electron beam and for the exposure to vacuum, making it unsuitable for live cells and affecting the geometry of biological samples. Subsequently, methods to circumvent the Abbe diffraction limit were developed to achieve higher resolutions for light microscopy.



(a) conventional wide-field fluorescence microscopy (b) localization microscopy image from a recording

Figure 5.1: Localization microscopy improves the resolution of fluorescence microscopy by about an order of magnitude. The positions of fluorescent molecules are determined with sub-pixel accuracy and plotted into a new image. [10]

The approach that overcame the Abbe diffraction limit first was stimulated-emission-depletion fluorescence microscopy (STED). The method was first published in 1994 [115] and requires the object to be colored with a fluorophore that acts as a marker for a certain type of protein in a cell. The fluorophores then emit light when excited by a laser of the excitation frequency, while the remaining parts of the object remain dark. STED scans the object sequentially with a higher resolution as the Abbe diffraction limit would permit. It does so by first depleting the fluorophores in the close environment of the current scan position with a doughnut-shaped laser spot, putting them into a dark (disabled) state for a short time. Immediately thereafter, the light source is switched to a second laser that excites only the fluorophores left enabled in the center of the doughnut-shaped depletion zone. Since the center can be smaller in diameter than the wavelength of light, the resolution is improved to 35-70 nm. The final image is compiled afterwards from all scan positions to be visible to the human eye.

Localization microscopy requires fluorophores in the object as well. Instead of switching them off with a depletion laser, special fluorophores oscillate themselves between a dark and a bright state. Stochastic optical reconstruction microscopy (STORM) [116] uses the stochastic blinking of the fluorophores in the object to allow their separation in a recording. With the bright state being active much shorter than the dark state, the center of each spot can likely be determined individually. Multiple methods based in this principle have been developed since 2006 by several research groups [3, 116, 117].

The center of each spot from a active fluorophore can be determined with a higher accuracy than its width on the image sensor when it is not obstructed by the spots of neighboring fluorophores. The location of a fluorophore can be determined with an accuracy of up to 5 nm, and with about 10 nm on average, improving the resolution compared to conventional microscopy by at least an order of magnitude.

The process does not only require a laser microscope (Fig. 5.2), but also considerable computing power to find the spots during their bright stage on each frame and to determine the center by fitting the spot. Additional tasks are background elimination and noise reduction. Depending on the number of fluorophores in the region of interest, a typical recording of five minutes with 10 frames per second can take multiple hours to process until finally the super-resolution image is plotted from the results. Handling errors during preparation and recording can therefore not be discovered until hours have passed. The latency causes the work flow of the experimenter to be slowed down considerably, hindering the spread of the technique in the biomedical community. To achieve real-time performance, the image analysis has to be as fast as the recording. For a typical frame rate of 10 frames per second, the processing of an individual frame must therefore not exceed 100 ms.

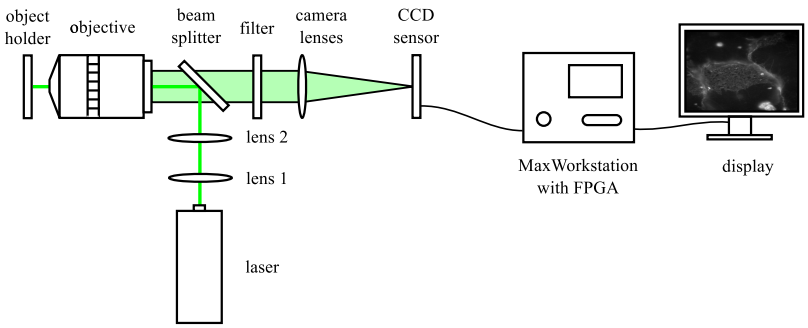
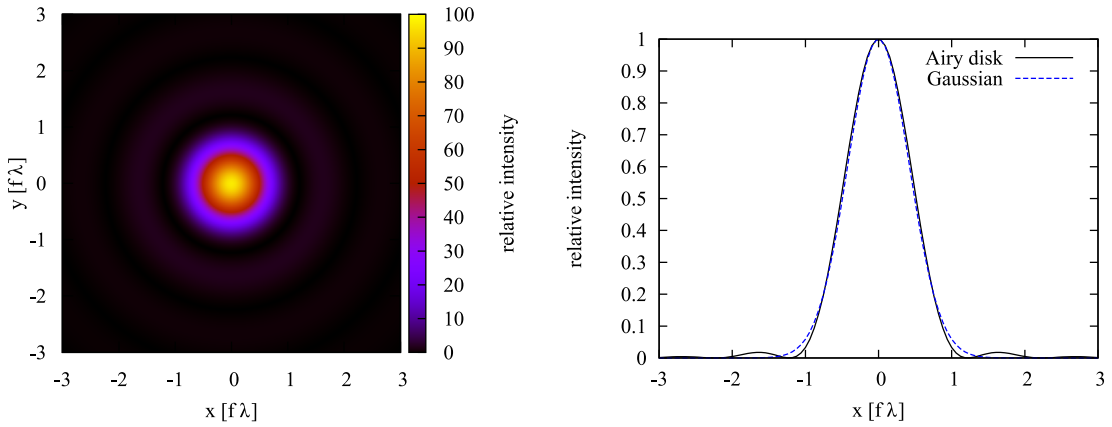


Figure 5.2: Simplified setup for localization microscopy. A laser illuminates the object and causes its fluorophores to blink stochastically. They act as optical markers and are recorded by a CCD camera. The application on the FPGA determines their positions and produces a super-resolution image.



(a) Airy disk intensity distribution.

(b) Approximation with a Gaussian profile.

Figure 5.3: The profile of an Airy disk. It describes the picture of a point-like light source. The minor maxima are barely visible. The intensity distribution can be approximated well with a Gaussian fit.

5.1.1 Physical principles

Even a perfect light microscope could not image a point-like light source in its focus area to a spot with zero width. Instead, the wave nature of light causes the spot in the image to be blurred, and the resolution of the microscope is limited by physical principles.

The first mathematical description of the diffraction of light that causes this effect was done by George Biddell Airy in 1835 [118]. When the light of the source travels through the optical system, it reaches the image on multiple paths. In the center of the spot, all waves interfere constructively and the signal is brightest here. When moving radially away from the center, the interference gradually turns destructive, producing a dark ring around the signal. Even farther, minor maxima can be seen. To calculate the pattern, the light has to be integrated for each possible path through the microscope along the optical axis. For systems with a circular aperture and a light source far away, the resulting pattern from the integral is described in Eq 5.1. It contains the Bessel function J_1 of the first kind and order one [119] and describes the intensity I at radius x from the center for a spot with total intensity I_0.

$$I(x) = I_0 \left(\frac{2J_1(\pi x)}{\pi x} \right)^2 \qquad J_1(y) = \frac{1}{\pi} \int_0^\pi \cos(\tau - y \sin(\tau))\, \mathrm{d}\tau \tag{5.1}$$

The intensity distribution, named Airy pattern after its discoverer, can be seen in Fig. 5.3 from above and in profile. It is shown in units of λf, where λ is the wavelength and f the focal width. The first minor maximum reaches less than 2 % of the relative intensity. The main maximum can be approximated well between the zeroes of the function with a 2D Gaussian profile $f(\vec{x})$ with center (μ_x, μ_y), width (σ_x, σ_y) and total (integrated) intensity Q. For objects with more than one point-like light source the resulting image is obtained by folding the input with the Airy disk as the points spread function.

$$f(\vec{x}) = \frac{Q}{2\pi\sigma_x\sigma_y} \mathrm{e}^{-\frac{1}{2}\left(\left(\frac{x-\mu_x}{\sigma_x}\right)^2 + \left(\frac{y-\mu_y}{\sigma_y}\right)^2\right)} \tag{5.2}$$

The Gaussian profile is much simpler to handle than the non-closed Bessel function in the mathematical description of the Airy profile. It is therefore preferred when fitting the spot to extract

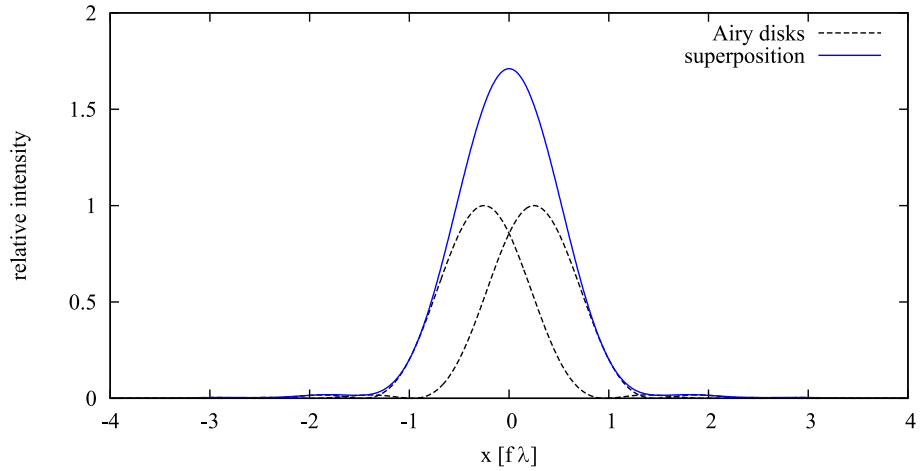


Figure 5.4: Abbe diffraction limit. The Airy disks of two fluorophores located closer than $\lambda/2$ cannot be distinguished any more when they are both in the bright state. Stochastic blinking, however, allows to fit and locate them separately.

its features. Furthermore, the Gaussian profile has a finite width $\sigma_{x,y}$ that can be used after fitting to identify spots of fluorophores that have drifted out of focus and are blurred by a larger extent.

For a microscope with opening angle θ, the Abbe diffraction limit for the minimum distance d of two points embedded into a medium of optical density n that can still be distinguished is given in Eq. 5.3 [113]. For commercial microscope, the limit is $d \simeq \lambda/2$. The plot in Fig. 5.4 depicts that if the points are closer, the Airy disks merge into a combined intensity distribution with a single maximum, and the individual locations cannot be distinguished any more from the superposition.

$$d = \frac{\lambda}{2n\sin(\theta)} \tag{5.3}$$

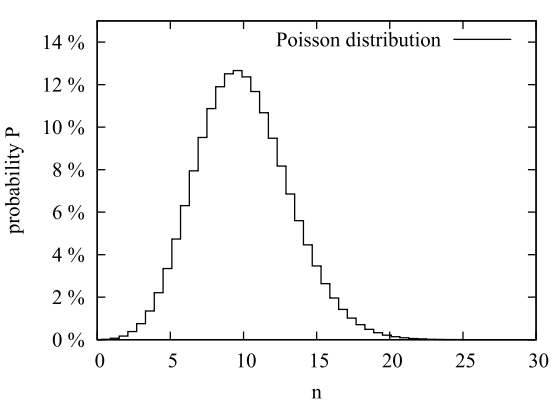
With localization microscopy, two spots with distance d most probably do not share the same optical state all the time during the recording, since the fluorophores are chosen to stochastically change their state many times during a recording. The activation times are chosen to be sparse, causing the visible period to be much shorter than the invisible. We can therefore fit the first spot when the second one is dark and vice versa, beating the Abbe diffraction limit.

Other fundamental limits still exist, and they constrain the newly improved resolution. First, the camera sensor is made of an array of discrete pixels that defines the location accuracy of a hit by a single photon. A fit takes many photons into account and mitigates this uncertainty. The individual error depends on the fit, but the more photons are available, the more precise a Gaussian profile can be fitted. Modern CCD cameras for localization microscopy have high photon efficiencies of more than 90 % to miss as few photons as possible.

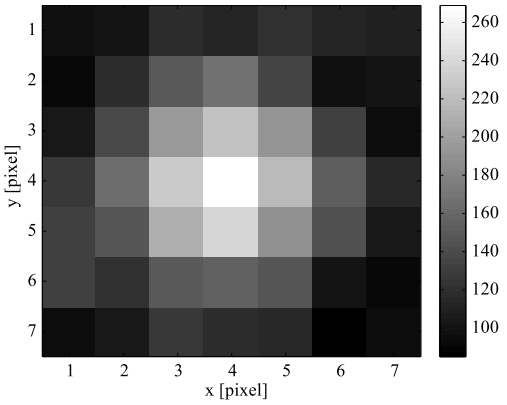
The second fundamental uncertainty is caused by the stochastic nature of photons. Any photon from a fluorophore has a small chance to hit a pixel on the camera sensor. The repeated process therefore follows a binomial distribution. For small individual probabilities, the binomial distribution approaches the Poisson distribution as a special case [104]. It is given in Eq. 5.4.

$$P_\lambda(n) = \frac{\lambda^n}{n!}\mathrm{e}^{-\lambda} \tag{5.4}$$

Its single parameter λ defines the mean and the width $\sigma = \sqrt{\lambda}$. For increasing values of λ, the distribution approaches a Gaussian distribution with the same mean and width. For small



(a) Poisson distribution with mean $\lambda = 10$ and width $\sigma = \sqrt{\lambda}$



(b) Simulated Airy disk with background on a CCD sensor

Figure 5.5: Distortion of an ideal Airy disk by pixelation and Poisson noise. Both effects limit the accuracy of a Gaussian fit and decrease the confidence of the obtained localization at its center.

λ, it becomes more skewed, but maintains zero probability for negative counts n. A plot of the distribution and a sample spot with pixelation and Poisson noise can be seen in Fig. 5.5. The relationship between width and mean allows us to estimate the noise from the measured value as $\sigma_{\text{est}} = \sqrt{n}$.

The noise is further increased by the background of the image that is created by structures out of the focus plane of the microscope. Complex objects like entire cells disperse photons from its organelles and produce an inhomogeneous background image that also follows the Poisson statistic. Other sources of noise include dark noise from thermal photons, clock induced noise that is introduced when the pixel charges are moved to the border of the sensor for read-out, and finally read-out noise that stems from the imperfect electric components before digitalization [120].

The magnitude of Poisson noise is given by the width of its distribution, and the relative width $\lambda/\sqrt{\lambda} = 1/\sqrt{\lambda}$ decreases with a brighter image, resulting in better fits and smaller localization errors. To excite as many fluorophores as possible monochromatic laser light is used that is tuned to match the fluorophores excitation frequency. Still, high intensity laser light used for illumination acts toxic and can damage or destroy a biological sample during the recording. The experimenter must therefore balance the intensity to maintain the integrity of the sample and the resolution of the localization image.

5.1.2 Localization Algorithms

The aim of this part of the thesis was to accelerate the image processing algorithm for localization microscopy. With a recording as an input that extends over many frames in time, the steps that have to be performed at least to create a super-resolution image from the input can be described as the following.

1. **Background removal:** This step eliminates the background of each frame caused by fluorophores out of focus and photon scattering, either on each frame individually or by taking into account information gathered from previous ones. The requirements depend on the capabilities of the next step.

2. **Spot detection and isolation:** The signal of every spot must be identified and its position must be forwarded to feature extraction. The quality of this processing step is defined by the

number of false positives and negatives that are easily induced by the photon noise in the recording.

3. **Feature extraction:** The most important information is the position of the fluorophore, derived from the center of the Airy disk, and the confidence it was measured with. Other features are the discrepancies from the Airy disk profile, caused by of out-of-focus fluorophores. Such spots are to be excluded from the final super-resolution image.

4. **Generation of the super-resolution image:** Finally, the positions are to be visualized. The plot should include information about the resolution at each position. The resolution depends on the confidence of the individual positions and the density of positions in their neighborhood.

The algorithm can only be split into the presented parts for recordings with sparely distributed spot signals. Here, overlapping spots can be discarded and the quality of the final localization image is still maintained. If too many spots overlap with distances at the Abbe diffraction limit, they cannot be examined any more with individual Gaussian fits. In this case, the fitting function has to be adjusted to also support approximation of multiple centers [121]. In the extreme case, clusters of computers have to be used to calculate with a hidden Markov model for typical localization input imagery with 256 $\times$256 pixels and more than 1000 frames [122]. In the following, we will focus on sparse recordings with sporadic overlaps.

Following the consequences of Amdahl's law (Fig. 2.5 on page 14), as many parts as possible of the chain of algorithms should be done in parallel on the FPGA accelerator. Therefore, a suitable algorithm is wanted that maintains accuracy and can be efficiently ported to a dataflow pipeline on an FPGA.

The software version that was used as a starting point and as a reference for this thesis was written in the programming language Matlab [4]. Its different steps follow the outline presented above, and its algorithmic parts will be described in the following sections next to the available alternatives. As we will see, porting the algorithms to an FPGA required a re-design of the program on all levels.

Background removal

The level of the background and its noise level on each frames must be known in the first place for reliable spot detection. The intensity of different recordings varies with the chosen power output of the illuminating laser, and therefore a threshold for spot detection must depend on the difference between image and background as well as the noise level. The noise level can be estimated for each pixel by calculating the square root of the photon count, since Poisson noise is the dominant source of noise in the system.

A first approach for background elimination that is implemented in the Matlab version is to calculate the difference of each frame I_n and its previous frame I_{n-1} pixel by pixel. Spots that are only visible in I_n clearly emerge from the subtraction, and for adjacent frames the background can be considered constant. This procedure is only recommended for backgrounds with strong inhomogeneities, since the Gaussian fit is also able to adjust to a certain background level. For the noise of the resulting image I' we find for the variance of the noise of each input image using error propagation:

$$I'_n = I_n - I_{n-1} \tag{5.5}$$

$$\sigma^2_{I'_n} = \sigma^2_n + \sigma^2_{n-1} \tag{5.6}$$

$$\sigma_{I'_n} \simeq \sqrt{2}\sigma_n \tag{5.7}$$

The noise level of the differential image is therefore increased by a factor of $\sqrt{2}$ when compared to the noise σ_n of the input images I_n. Removing the background should, however, only add as few noise as possible to not further disturb spot detection and feature extraction in the later steps. As an alternative, we can assume that photo bleaching changes the background only slowly. Thus, we can subtract it with few additional noise by first calculating the sliding average of the last M frames to obtain a map $B_{n,\mathrm{SA}}$ of the background.

$$I'_n = I_n - B_{n,\mathrm{SA}} \qquad B_{n,\mathrm{SA}} = \frac{1}{M}\sum_{i=1}^{M} I_{n-i} \tag{5.8}$$

$$\sigma^2_{I'_n} = \sum_{i=0}^{M} \left(\frac{\partial I'_n}{\partial I_{n-i}}\right)^2 \sigma^2_{n-i} \tag{5.9}$$

$$= \sigma^2_n + \frac{1}{M^2}\sum_{i=1}^{M} \sigma^2_{n-i} \tag{5.10}$$

$$\simeq \sigma^2_n + \frac{1}{M}\sigma^2_n \tag{5.11}$$

For $M = 1$ we would get the same increase in noise as in Eq. 5.7 for the plain difference of images. For background maps that average over more than one past frame, however, the noise introduced becomes smaller. Due to the quadratic addition the increase of noise to the variance $\sigma_{I'_n}$ is less than 5% for $M = 10$. The value of M should not be be chosen too large to ensure that the responsiveness of the background map matches the rate of change of the background in the recording.

While the drop in noise compared to simple subtraction of the previous frame is beneficial, a hardware implementation of a sliding average would multiply the number of memory accesses by the number of past images M. These images have to be stored and read every time a new frame arrives. Exponential smoothing is more efficient and has a similar effect.

For exponential smoothing only one image, namely the previous background map B_{n-1}, has to be kept in the memory of the pipeline. The algorithm is shown in Eq. 5.13. To calculate the current background map B_n only the current image frame I_n and the previous background B_{n-1} are needed. The smoothing is parametrized by the smoothing factor $1/N$.

$$I'_n = I_n - B_n \tag{5.12}$$

$$B_n = B_{n-1} + \frac{1}{N}(I_n - B_{n-1}) \tag{5.13}$$

$$= \frac{1}{N}\sum_{i=0}^{n-1} \left(\frac{N-1}{N}\right)^i I_{n-i} + \left(\frac{N-1}{N}\right)^n B_0 \tag{5.14}$$

The sequence B_n is geometric, its explicit definition is given in Eq. 5.14. We use it to calculate the increase in noise for I'_n after background subtraction (Eq. 5.18). The influence of each frame on B_n decreases over time as new frames arrive, and once again we assumed that the background and its width σ_{n-i} does not vary too much within the last N frames. When compared to the sliding average, exponential smoothing has a similar effect on the noise after background subtraction for $N = 2M$.

$$\sigma_{I'_n}^2 = \sum_{i=0}^{n-1} \left(\frac{\partial I'_n}{\partial I_{n-i}}\right)^2 \sigma_{n-i}^2 + \left(\frac{\partial I'_n}{\partial B_0}\right)^2 \sigma_{B_0}^2 \tag{5.15}$$

$$= \sigma_n^2 + \frac{1}{N^2} \sum_{i=0}^{n-1} \left(\frac{N-1}{N}\right)^{2i} \sigma_{n-i-1}^2 + \underbrace{\left(\frac{N-1}{N}\right)^{2n}}_{\to 0 \text{ for } n\to\infty} \sigma_{B_0} \tag{5.16}$$

$$\simeq \sigma_n^2 + \frac{\sigma_n^2}{N^2} \frac{1 - \left(\frac{N-1}{N}\right)^{2n}}{1 - \left(\frac{N-1}{N}\right)^2} \tag{5.17}$$

$$\simeq \sigma_n^2 + \frac{2}{N}\sigma_n^2 \tag{5.18}$$

The proposed background treatment can be seen as a high-pass filter applied to each pixel over the time of the recording. Filtering frequencies within each individual frame over *space* would be more difficult: the Airy disk profile of a spot closely resembles a Gaussian distribution, which is mapped to a Gaussian distribution in the frequency domain again. Its most defining frequencies are close to zero, and overlap with the spectrum found in the background map. Removing the background for each frame therefore would impact the signal of the spot as well.

As a last optimization, the calculated background map can be limited to not rise too fast. When a spot is seen of intensity I_{spot}, it would otherwise increase the background map at its position by I_{spot}/N. An image is only expected to vary within about $\sigma_{n,\text{estimated}} = \sqrt{B_{n-1}}$, and the following rule takes this into account:

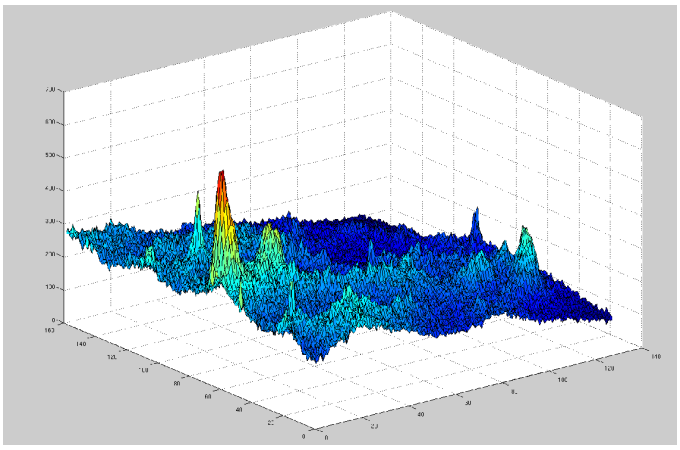
$$B_n = B_{n-1} + \frac{1}{N} \min\left((I_n - B_{n-1}), \sqrt{B_{n-1}}\right) \tag{5.19}$$

The entire process can be seen in Fig. 5.6. Note that all calculations rely on the fact that the value of each intensity is close to the number of photons that hit the CCD sensor at this pixel. For cameras that do not follow this specification the intensity must be adjusted in advance to make this assumption true.

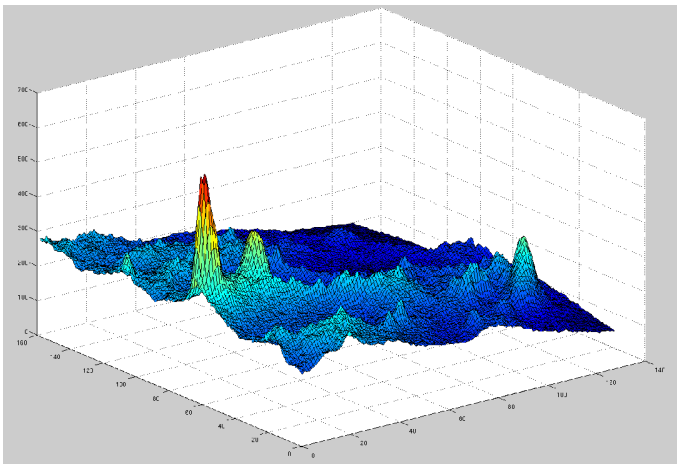
Spot detection

After the background has been subtracted, it has become much easier to identify the spots in the image. A spot must have an intensity above a certain threshold, and its center is a local maximum. For the threshold, we can use the estimated variance of the background noise $\sigma_{n,\text{estimated}} = \sqrt{B_{n-1}}$ again for each pixel and define a multiple of it as the lower limit the center of a spot has to reach. For a threshold of $4\sigma_{n,\text{estimated}}$, less than 0.004% of all pixel intensities are expected to reach the boundary, resulting in about two false positives per 256×256 px frame. The false positives are expected to be removed in the feature extraction step because they do not resemble an Airy disk.

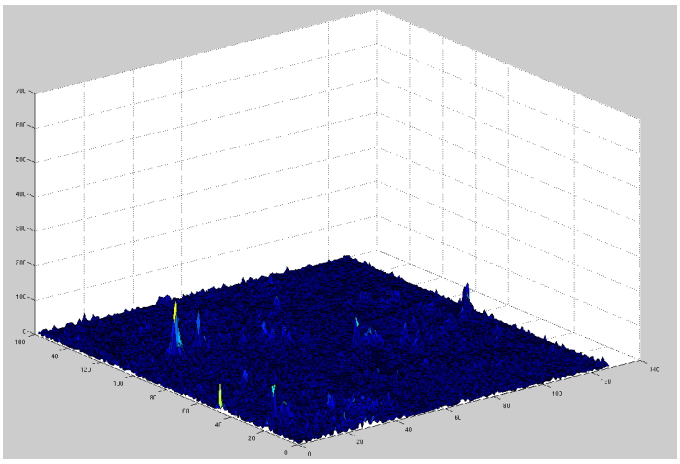
For the determination of the local maximum a comparison of every pixel with its neighboring pixel is sufficient. This can be done efficiently on an FPGA with a line buffer for the previous and the next line of the current position in a pipeline, pixel by pixel. However, the true center of spot is not necessarily at the same pixel where the intensity peaks due to noise that is still present in the image. To provide good initial parameters to the following fit, the image can be blurred for this purpose only with a small stencil. On a CPU, this step would increase the number of register transfers proportionally to the stencil size and would impact the performance. On an FPGA, however, it can be integrated into the pipeline with few additional resources.



(a) recorded input frame I_N



(b) background map B_n



(c) frame I'_n with spots after saturated subtraction

Figure 5.6: Inhomogeneous background handling. The background in an individual input frame is too high and inhomogenous for spot detection. The backround map is created with exponential smoothing over previous frames. The signals of the spots stand out only after the background map was subtracted. [10]

The Matlab version does not use a threshold to detect spots, but requires the user to provide the total number of spots expected per frame. It then applies a Gaussian least-square fit on the entire frame repeatedly. After a spot was fitted, it is blacked out and the fit starts again, likely fining the next significant spot until the specified number of spots has been found.

Feature extraction

The features of a spot are the position of its peak, its width in both dimensions and the total intensity. Additionally, the confidence of the position must be known. All parameters can be obtained by fitting, i.e. adjusting the parameters of a 2D-Gaussian profile as close as possible to the data. Two popular measures exist to estimate how close a function can be fitted to the data. These are used by the least-square method and the maximum likelihood method, respectively.

Least square For the least-square method, the differences between fit function and data are squared and added, resulting in a scalar value that is then to be minimized [123]. The difference between a data point and the fit function is also known as the residual. In our application, a data point is given by the intensity $q_{x,y} = I_n(x,y)$ of a pixel at position (x,y) in a spot, and the fit function is the Gaussian profile. The scalar to minimize is χ^2 with respect to the center $\mu_{x,y}$, width $\sigma_{x,y}$ (not to be confused with the variance of the noise), the total intensity Q of the spot and, if needed, a background scalar B.

$$\vec{a} = (Q, \mu_x, \mu_y, \sigma_x, \sigma_y, B) \tag{5.20}$$

$$\chi^2 = \sum_{(x_i,y_j) \in \text{ROI}} \left(q_{i,j} - f_{\vec{a}}(x_i,y_j)\right)^2 \tag{5.21}$$

$$f_{\vec{a}}(x,y) = \frac{Q}{2\pi\sigma_x\sigma_y} \mathrm{e}^{-\frac{1}{2}\left(\left(\frac{x-\mu_x}{\sigma_x}\right)^2 + \left(\frac{y-\mu_y}{\sigma_y}\right)^2\right)} + B \tag{5.22}$$

χ^2 cannot be minimized analytically. Instead, the minimum has to be found numerically. The optimal set of fit parameters $\vec{a}$ is approached by either following the steepest descent of χ^2 or by approximating χ^2 as a quadratic function and jumping to its assumed minimum [124]. In the first case the parameter set $\vec{a}$ is modified by $\delta\vec{a}$ along the direction of the slope of χ^2.

$$a_l' = a_l + \delta a_l \tag{5.23}$$

$$\delta a_l = \text{constant} \times \left(-\frac{1}{2}\frac{\partial\chi^2}{\partial a_l}\right) \tag{5.24}$$

$$= \text{constant} \times \sum_{(x_i,y_j) \in \text{ROI}} \left(q_{i,j} - f_{\vec{a}}(x_i,y_j)\right) \frac{\partial f_{\vec{a}}(x_i,y_j)}{\partial a_l} \tag{5.25}$$

In the second case, we develop χ^2 to the second order and use the Hessian matrix D to jump from $\vec{a}$ to the (assumed) minimum at $\vec{a}'$ directly.

$$\chi^2(\vec{a}) = \chi^2(\vec{a}_0) - \frac{\partial\chi^2}{\partial\vec{a}} \cdot \vec{a} + \frac{1}{2}\vec{a} \cdot D \cdot \vec{a} + \ldots \qquad D_{kl} = \frac{\partial^2\chi^2}{\partial a_k \partial a_l} \tag{5.26}$$

$$a_l' = a_l - \sum_k D_{kl}^{-1} \frac{\partial\chi^2(\vec{a})}{\partial a_k} \tag{5.27}$$

Both method have different realms where they perform best. Far away from the minimum, the function is unlikely to still be a approximated well with a quadratic equation, and the steepest descend method performs better. When close to the minimum, χ^2 can be expected to be smooth enough and the second method is expected to converge faster.

The Levenberg-Marquardt method [123] combines both approaches continuously and has become a standard method for least-square fits. For this, the Hessian Matrix D in Eq. 5.27 is substituted by $A = D + \lambda \mathbb{1}$. This way, the "constant" in the steepest descend method is removed. For λ approaching zero, the Levenberg-Marquardt method becomes the same as the quadratic approximation. For $\lambda \gg 1$, the diagonal elements of A dominate and the method follows the steepest descent. λ is modified at runtime by the fit algorithm to grow when χ^2 increases, and to shrink when χ^2 decreases until χ^2 almost stops decreasing. Then, the fit has reached an optimum. The confidence of the individual fit parameters can be taken from the diagonal elements of D^{-1}, which acts as the covariance matrix.

Matlab supports the Levenberg-Marquardt method with the `lsqcurvefit` library function [125], and it is used to fit spots in the Matlab version that was used as a starting point for accelerating the analysis of the raw data from localization microscopy. The fit function includes the constant background offset B as well as two extra background parameters to fit a constant slope in the x and y direction. This algorithm for fits is compute intense: To evaluate χ^2 or one of its derivatives the exponential of the fit function has to be calculated for every pixel in the ROI. Additionally, the inverse of the Hessian matrix has to be obtained for the quadratic approximation, which is an operation of at least the order of $\mathcal{O}(n^2 \log n)$ for small matrices with n being the number of fit parameters. Last, but not least the total number of iterations $\vec{a} \to \vec{a}'$ before the stopping condition is reached is not known in advance.

Maximum likelihood The maximum-likelihood method follows a different approach than minimizing χ^2. Instead, it introduces the *likelihood* L of a distribution to fit the data points [126]. It is defined for two dimensions in Eq. 5.29 and maximized by adjusting the fit parameters $\vec{a}$ for the fit. By calculating the likelihood we can determine how probable the hit of a pixel at position $\vec{x}_{i,j} = (x_i, y_i)$ in the camera by a photon γ_i is to happen. For multiple photons, these probabilities are multiplied:

$$L = \prod_{\gamma_i} f_{\vec{a}}(x_i, y_i) \tag{5.28}$$

$$l = \ln(L) = \sum_{\gamma_i} \ln\left(f_{\vec{a}}(x_i, y_i)\right) \tag{5.29}$$

Maximizing L requires us to derive the product with respect to the fit parameters. Since sums are easier to differentiate than products, and the exponential in the fit function is removed by the natural logarithm, the function l is preferred over L for maximization. The logarithm is a strictly increasing function and therefore does not affect the solution of the minimization.

We rely on the background removal in the previous step, set $B = 0$ in the fit function and continue with a Gaussian distribution only scaled by the total intensity. We determine the maximum of l by deriving the fit function with respect to the parameter set and obtain the following system of equations for $\vec{\mu}$, σ_x and σ_y [126]. Note that γ is the index for photons, while i, j labels the position $\vec{x}_{i,j}$ of the pixel that was hit. Eq. 5.32 and Eq. 5.34 do not give the total intensity Q. It cannot be derived from the maximization, but is already known to be the sum of all photons. The equations for the center $\vec{\mu}$ and the width of the spot (σ_x, σ_y) are known as the Gaussian estimator, as the center of mass or the centroid. They provide an analytic, easy to compute solution to the problem

of performing a maximum likelihood fit with a Gaussian profile, given that the background does not have to be fitted as well.

$$\frac{\partial l}{\partial \vec{a}} \stackrel{!}{=} \vec{0} \qquad Q = \sum_{\gamma} 1 = \sum_{x,y} q_{x,y} \tag{5.30}$$

$$\frac{\partial l}{\partial \mu_x} = \sum_{\gamma} \frac{x_\gamma - \mu_y}{\sigma_x} = \sum_{i,j} q_{i,j} \frac{x_i - \mu_x}{\sigma_x} \stackrel{!}{=} 0 \tag{5.31}$$

$$\Rightarrow \mu_x = \sum_{i,j} \frac{q_{i,j}}{Q} x_i \qquad \mu_y = \sum_{i,j} \frac{q_{i,j}}{Q} y_j \tag{5.32}$$

$$\frac{\partial l}{\partial \sigma_x} = -\frac{Q}{\sigma_x} + \sum_{\gamma} \frac{\left(x_\gamma - \mu_x\right)^2}{\sigma_x^3} \stackrel{!}{=} 0 \tag{5.33}$$

$$\Rightarrow \sigma_x^2 = \sum_{i,j} \frac{q_{i,j}}{Q} \left(x_i - \mu_x\right)^2 \qquad \sigma_y^2 = \sum_{i,j} \frac{q_{i,j}}{Q} \left(y_j - \mu_y\right)^2 \tag{5.34}$$

The error of the localization $(\Delta\mu_x, \Delta\mu_y)$ is obtained through error propagation (Eq. 5.35). For the pixelation error we find $\Delta x^2 = \Delta y^2 = 1/12$ when we integrate a constant probablility function over the width of a pixel. The variance of $q_{x,y}$ is $\Delta q_{x,y} = \sqrt{q_{x,y} + B_{x,y}}$ for the assumed Poisson noise. As expected the localization error $(\Delta\mu_x, \Delta\mu_y)$ decreases if the total number of photons Q rises, and increases for high background levels $B_{x,y}$.

$$\Delta\mu_x^2 = \sum_{\gamma} \left(\frac{\partial \mu_x}{\partial x_\gamma}\right)^2 \Delta x_\gamma^2 + \sum_{i,j} \left(\frac{\partial \mu_x}{\partial q_{i,j}}\right)^2 \Delta q_{i,j}^2 \tag{5.35}$$

$$= \frac{1}{12Q} + \sum_{i,j} \left(\frac{x_i - \mu_x}{Q}\right)^2 \left(q_{i,j} + B_{i,j}\right) \tag{5.36}$$

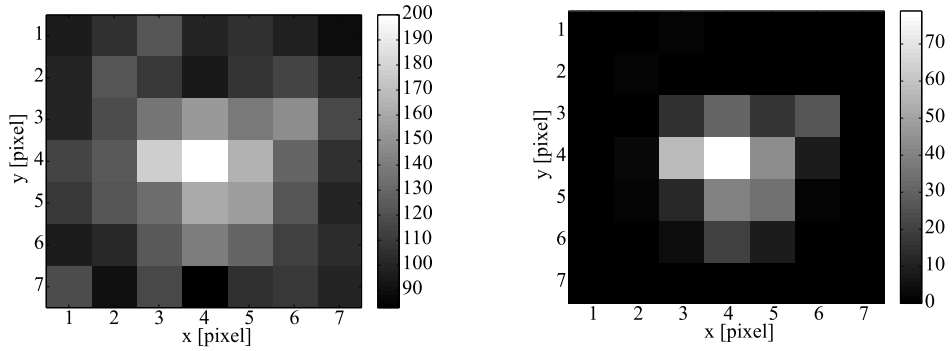
$$\Delta\mu_y^2 = \underbrace{\frac{1}{12Q}}_{\text{pixelation}} + \underbrace{\sum_{i,j} \left(\frac{y_j - \mu_y}{Q}\right)^2 \left(q_{i,j} + B_{i,j}\right)}_{\text{Poisson noise}} \tag{5.37}$$

When fitting a Gaussian we expect the pixels remote to the center $\vec{\mu}$ to have vanishing values $q_{x,y}$. If not, we can already see that the equation for the center $\vec{\mu}$ of the Airy disk is susceptible in this case: the term $(q_{x,y}/Q)x$ wrongfully draws the calculated center towards the non-vanishing pixel. In localization microscopy this might happen at two occasions, either due to noise in the image or by the tail of a second spots that leaks into the neighborhood of the spot to be fitted.

To address noise we may simply subtract e.g two times the width of the noise, set any negative values to zero and thereby remove 99% of the noise where the signal should be zero. The effect of removing the background and additionally performing a saturated subtraction of $2\sigma_{\text{noise}}$ can be seen in Fig. 5.7. Setting values to zero near the border is equal to excluding pixels from the calculation of the location and the width, and to tailoring the ROI to the shape of the spot.

For the effect of other spots moving the center towards where their tail touches the current spot, we can find the minimum in intensity between both and set all values behind it to zero that belong to the other spot. Setting values of $q_{i,j}$ to zero removes them from the sum in Eqs. 5.32, excluding them from the fit.

The size of the quadratic ROI where the fit takes place in must be chosen between two extremes. A ROI too small will cut off valuable information, it must therefore be at least twice the width of



(a) with background

(b) background and $2\sigma_{\text{noise}}$ subtracted

Figure 5.7: A simulated noisy spot with total charge $Q = 1000$, background level $N_B = 100$ and width $\sigma_{x,y} = 1.4\,\text{px}$ at $\vec{\mu} = (4.1, 4.2)$. By setting pixels with $q_i < 2\sigma_{\text{noise}}$ to zero, the feature extraction becomes robust against ROIs with non-vanishing values at the border. Here, the center was determined with an sub-pixel accuracy of 0.16 px or 16 nm. [10]

the spot. A ROI too big, however, may include noise at the borders and increase the chance that a close-by spots leaks into the ROI, disturbing the fit. We will examine the fine tuning of the entire algorithm later to find the optimal parameters for localization microscopy with visible light.

Super-resolution image generation

Finally, when all spot locations have been computed from the recording, the data is visualized by plotting the spots into a new image with a higher resolution. The pixel size of an optical camera is build close to the optical resolution, as a finer resolution would decrease photon efficiency and increase noise. For a recording with a resolution of 256×256 pixels, it is therefore common to increase the resolution of the final plot by a factor of ten, much like the precision improvement that is an order of magnitude. A super-resolution image will then have a resolution of 2560×2560 pixels.

In the most simplest case, the positions are plotted as single white dots in the final image. A better plotting technique that was used in the localization image shown in Fig. 5.1b (p. 72) is to blur each dot with its confidence before. This way, the localization error is visible to the human eye. Other imaging techniques also require the distance of its nearest neighbor(s) to determine the blur radius for visualization.

With recordings typically having tens of thousands of fluorophore position to be plotted, the visualization may take a considerable amount of time. Each position must be blurred, which is implemented by using a Gaussian distribution to draw the gray-values around a spot position. For this the Gaussian function must be calculated repeatedly. The total time for image generation can approach multiple minutes.

5.1.3 State of the Art

Starting from initial algorithms that relied on the least-square fit for localization, a couple of solutions have been developed to mitigate the long hours consumed by the analysis. The approaches for acceleration so far can be split into improvements of the algorithms and portings to special hardware.

The center-of-mass calculation as presented before was examined before with regard to the selection of a suitable threshold and the exclusion of noisy pixel values for localization [127]. The background must be determined and subtracted manually before analysis can start. A flavor of the

presented center-of-mass algorithm is also available as a plug-in for the free software ImageJ [128].

The least-square fit and the Gaussian estimator, although seemingly different algorithms, can be combined by using the previously known Gaussian profile as a mask of weights $M_{x,y}$ for the calculation of the center of mass [129]. The center of the mask must be the same as the center of the spot $\vec{\mu}$, and the algorithms uses an iterative approach until both positions are similar (Eq. 5.38). As an advantage, noisy pixel values at the boundary are continuously masked out from the calculation of $\vec{\mu}$. Still, the algorithm is iterative and therefore time-consuming for a large number of spots.

$$\mu_x = \frac{\sum\limits_{i,j} x_i q_{i,j} M_{i,j}}{\sum\limits_{i,j} q_{i,j} M_{i,j}} \tag{5.38}$$

The fluoroBancroft algorithm was developed as a non-iterative alternative to least-square fitting [130]. Its accuracy is comparable to a Gaussian fit for spots covering only a small number of pixels with a low background level and a high signal-to-noise ratio. The width of the spot must be known in advance, but can usually be measured or computed before, since all spots from the focus area should share the same profile only scaled by the total intensity. The development of the algorithm was inspired by the also closed-form (non-iterative) Bancroft's algorithm used for position determination in the Global Positioning System (GPS) [131]. The calculation of a localization with the fluoroBancroft algorithm requires the natural logarithm for each pixel and a matrix inversion to be evaluated. Equation 5.39 gives the required algorithm for the computation of a localization $\vec{\mu}$ in two dimensions with intensity I_i and background $N_{i,\mathrm{Bg}}$ at position (x_i, y_i).

$$\vec{\mu} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \left(\left(B^{\mathrm{T}} B \right)^{-1} B^{\mathrm{T}} \right) \vec{\alpha} \tag{5.39}$$

$$\alpha_i = \frac{1}{2} \left(x_i^2 + y_i^2 + 2\sigma^2 \ln\left(I_i - 2N_{i,\mathrm{Bg}} \right) \right) \qquad B = \begin{pmatrix} x_1 & y_1 & 1 \\ & \vdots & \\ x_n & y_n & 1 \end{pmatrix} \tag{5.40}$$

It is beyond the scope of this thesis to provide the derivation of the equations, but we can see that the logarithm cannot be evaluated for pixel values with a background level of half the intensity or above. These values must at least be removed from the set of pixels. The algorithm does not provide the confidence of the extracted localization, but it was shown that it scales with the inverse square root of the total number of photons [132], similar to the Gaussian estimator.

As a summary of the algorithms, all methods provide very similar results for well-defined signals from spots with low noise and a high total intensity. For weak signals center-of-mass and least-square fits introduce a large bias, but behave differently [133]. Least-square fits tend to not converge or to produce a fit with a center far off the true location. On the other hand, center-of-mass algorithms are biased towards the center of the ROI that was used to cut out the spot. Center-of-mass also shows a bias towards the center if the background has not be subtracted completely, leading to pixel locking in the final image, i.e. every location is biased towards the center of the underlying pixel. It is therefore of high importance to estimate the background level correctly. Least-square fits can compensate for homogeneous backgrounds with additional fit parameters and are therefore easier to deploy at the cost of computing time.

The runtime of localization algorithms so far has only been sped up with graphics cards as application accelerators. The *MaLiang* method [9] is a port of an iterative maximum likelihood fit. Since the implementation includes the background level in the fit as a parameter, it does not resolve to the Gaussian estimator. Despite its iterative nature, it provides real-time capability even for fast

cameras. Compared to fluoroBancroft, the analysis of a typical frame is faster by more than a factor of ten on an NVIDIA GeForce 9800GT graphics card, and faster by more than a factor of 20 than a Gaussian least-square fit in software.

5.1.4 Analysis of the Algorithm

The analysis of the algorithm for localization microscopy started with an implementation in Matlab that was provided by the research groups of Prof. Christoph Cremer from the Institute of Molecular Biology Mainz and Prof. Michael Hausmann from the Kirchhoff Institute for Physics at Heidelberg University. It is embedded in a suite of tools for analysis and visualization.

The algorithm consists of a chain of operations. First, the recording is read, encoded as a stack of frames in either the Khoros Standard Data Format (KDF) or the Tagged Image File Format (TIFF) format. For imagery with a high inhomogeneous background level, an option is provided to perform the analysis on the difference of the current with the previous frame, increasing the noise level as described in section 5.1.2 above. An iterative least-square fit with a Gaussian profile is used for the feature extraction. Finally, the super-resolution image is generated by plotting the locations and blurring them with their fit error.

As a starting point for acceleration the program was profiled with the Matlab profiling tool. Since profiling slows down the execution, an image stack with only few frames was used. The result can be seen in Fig. 5.8. The operation that consumes the most computing time is the fit function `nonlin2` that performs a least square fit on the ROI of each spot. It computes the Gaussian function 26 times on average per spot, which is returned by `gauss2dbPL3D` for a given position. Afterwards, the fit has converged or was aborted. Both functions consume about 56% of the total compute time.

`dip_image.dip_image` is a converter function that makes a frame from the TIFF image stack available for the DIP library functions, a collection of image processing function to simplify common operations [134]. It copies the data and can be eliminated with our own implementation of a TIFF library. The remaining functions have a small self time: `ofindSPDM` searches for spots in an image and `fitSPDM` is the top-level function.

Overall, a big amount of time is spent in the least-square fit that could be saved with the non-iterative Gaussian estimator. We started with an examination of the fit function and then re-implemented the remaining parts of the application.

Methods

While an FPGA application could certainly accelerate the iterative least-square fit, it seemed that the time-consuming feature extraction could be already accelerated by moving this part of the algorithm to a fit with a closed form. Since a modification is very likely to change the properties of the extracted locations, the precision of all known closed-form fits had to be evaluated, notably the Gaussian estimator and the fluoroBancroft algorithm.

The accuracy of a fit is given by its fit error, in our case the difference between the true location of the spot and the extracted center of the Gaussian profile. However, real data coming from the microscope cannot be used for this purpose, as the true locations are unknown. It is therefore necessary to generate artificial spots that follow the characteristics of real spots. In particular, the noise of the background and the spot have to be simulated as closely as possible to the real-world data.

The Monte-Carlo method provides such a simulation [135]. It combines deterministic elements such as the true position, the width and the total intensity of a spot with random elements for noise generation. The noise was known to be dominated by Poisson noise and could be generated accordingly with a pseudo-random number generator with the undisturbed profile of an ideal spot

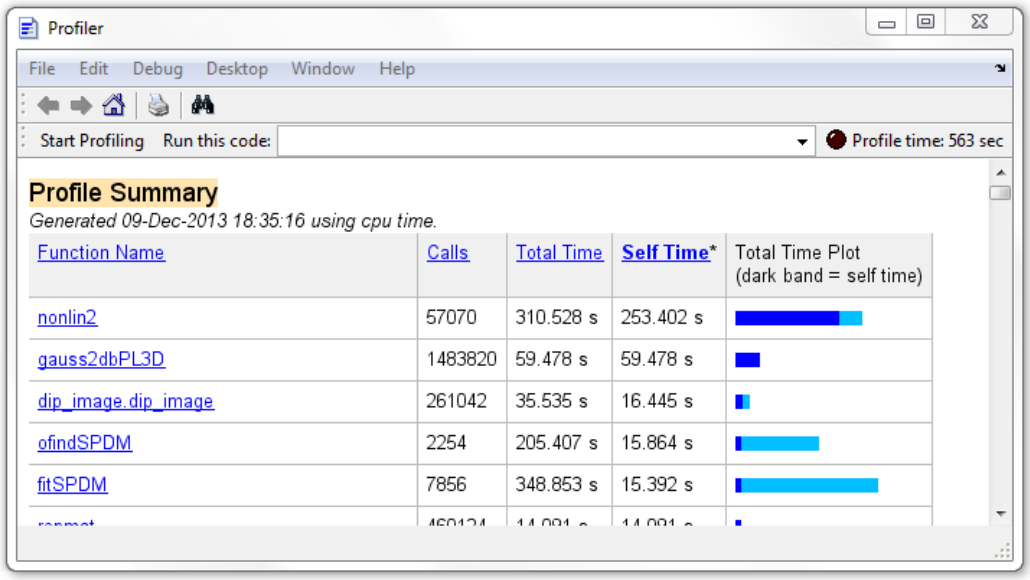


Figure 5.8: Profiler result of the Matlab implementation for Localization Microscopy.

as the input. After a large number of simulated spots and performed fits the average fit error and its distribution could be determined. For each set of width (σ_x, σ_y), total intensity Q and background level B, 2000 virtual frames were simulated.

The original algorithm was already present in Matlab and therefore the simulation environment was written in the Matlab programming language as well. The aim of the process was to find an algorithm that maintains accuracy and at the same time can be implemented efficiently in the dataflow model on an FPGA.

The accuracy of the entire algorithm does not only depend on the fit algorithm alone, but also on the starting parameters and, to an even greater extent, to the ability to identify spots on a noisy background in the first place. Therefore, a sufficiently sized frame had to be simulated. The true position was varied such that it occupied all positions within the boundaries of a pixel with equal probability, as a fit may have a bias towards the center of a pixel or the intersection between pixels (pixel locking).

For the background removal, several approaches were examined. Iterative fits and fluoroBancroft are able to include the background into the fit, while the Gaussian estimator relies on the subtraction of the background in advance.

After simulation, the following algorithm proved suitable in both accuracy and dataflow for implementation on an FPGA:

1. **Background elimination:** Exponential smoothing was chosen to determine the background map $B_n(x,y)$ for frame n. The rise of the background was limited to the width of the Poisson noise $\sqrt{B_{n-1}}$ to limit its rise when a spot is encountered. The smoothing factor was chosen to be $N = 8$, large enough for typical spots that are visible only in a single frame before becoming dark again, but small enough to follow changes in the background caused by bleaching.

$$B_n = B_{n-1} + \frac{1}{N} \min\left((I_n - B_{n-1}), \sqrt{B_{n-1}} \right) \tag{5.41}$$

2. **Spot finding:** After the background was subtracted and any negative values have been set to zero, a copy of the resulting image is made and blurred by convolving it with a stencil

$S = 1_{33}/9$, that is a 3×3 matrix where each element equals $1/9$. While the copy is unsuitable for feature extraction after blurring, this step is beneficial for spot finding. The fluctuations of the noise are greatly reduced and the true center of a spot is more likely to be at the pixel with the highest value. For a spot to be detected, its center pixel must be a local maximum and above a certain threshold, given in multiples of the width of the noise level. This blurring improves the centering of the ROI for the spot, which is then forwarded to the next processing step. For the given combination of microscope and CCD camera, a 7×7 ROI was chosen.

3. **Spot separation:** The ROI may also contain the signal of more than one spot. A signal of the main spot in the center of the ROI may be disturbed by a second spot close-by that leaks into the ROI with its tail. If the disturbance is not too large, the signal of the second spot can be removed well by removing all pixels beyond a local minimum when radially scanning from the center to the borders of the ROI. Spot separation does not improve the signal of the spot, but it helps the following feature extraction. Therefore, the confidence of the fit must be calculated without spot separation.

4. **Zero suppression:** Noise left over in the ROI from the previous step reduces the quality of the fit, especially when it increases the value of pixels at the boundary of the ROI for the center-of-mass calculation in the Gaussian estimator. We therefore suppress all values smaller than two times the width of the Poisson noise by subtracting all values in the ROI with $2\sigma_B = 2\sqrt{B_n}$ and then setting all negative values to zero.

5. **Feature extraction:** The Gaussian estimator is used to determine the properties of the spot. For the center $\vec{\mu}$, the ROI with zero suppression is used. All other properties are determined with zero suppression disabled.

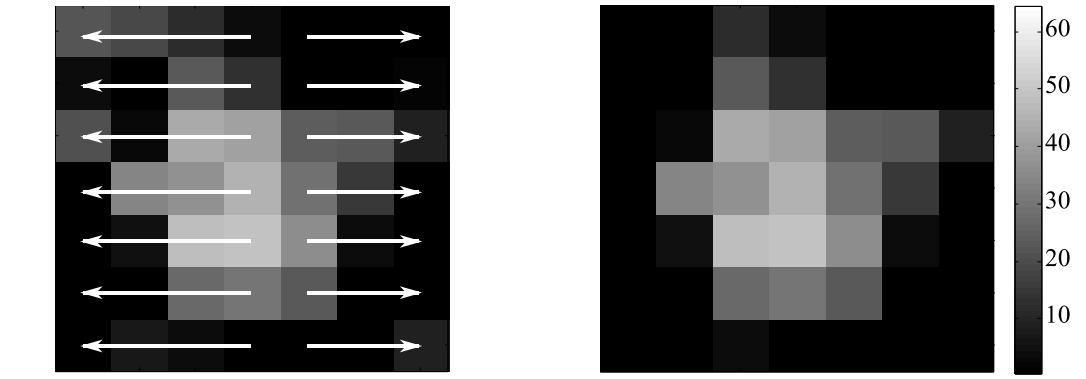
6. **Super-resolution image generation:** The drawing of the final image is left to the software on the CPU. It can still be accelerated by calculating the profile of a location blurred by its confidence only once for a given width casted to an integer. After the first calculation, its bitmap is cached in a hash map. For further spots with the same confidence the bitmap can be retrieved again and be drawn on the image with a saturated add. The discretization of the confidence is chosen fine enough to be invisible to the human eye.

Since Matlab is an interpreted computer language, its runtime is susceptible to the degree of library function versus interpreted code found in the program. The libraries for core functions such as matrix operations are implemented natively and have been heavily optimized, but the gained speed can be easily lost in the scripted part [136]. To ensure a fair comparison between software on CPUs and hardware acceleration with FPGAs, the algorithm presented above was also implemented in the C++ programming language.

Data flow

The algorithm reads the image stack frame by frame. For the stages for background elimination and spot finding, the unit of processing is the frame, accessed pixel by pixel in x and y direction. Background elimination contains a loop in the data flow, since the background map for the current frame depends on the data of the previous background map (Eq. 5.41). Therefore, an entire background map has to be buffered and read back later with a delay equal to the number of pixels in one frame.

For spot finding, the data flow branches into two copies of the stream. One stream is blurred and used for spot finding as described above, looking for a local maximum above a threshold. The other stream is unmodified and, if a spot has been found, used to create a 7×7 pixel ROI with the



(a) a second spot from top left touches the ROI (b) after separation

Figure 5.9: The access pattern of the spot separator is from center to left and right and from top to bottom. Every pixel value is read only once. Values that are past a local minimum in are set to zero. The process is then repeated in vertical direction and in horizontal direction again. [10]

spot in its center. Here, the unit of processing changes from entire frames to small ROIs. Because ROIs can overlap for close-by spots, pixel values may need to be duplicated, and its ROIs must be buffered into a queue for further processing.

The following steps from spot separation to feature extraction operate on ROIs, until finally a location is produced with the other features of the spot attached. Zero suppression is a point function and therefore easily integrated. For spot separation, the ROI must be scanned from the inside to the outside to find local minima and set all pixel values behind to zero. This can be done by scanning the ROI horizontally, vertically and horizontally again. For each direction, every pixel has to be read and potentially set to zero once. The access pattern is shown in Fig. 5.9 and differs from the usual sequential pattern from left to right and from top to bottom. However, since the latency is still restricted to a ROI and only the access order within the ROI is changed, it can be embedded into the same data flow without the need to break the pipeline apart.

The final step on hardware, feature extraction, relies on accumulators to calculate the total intensity Q, the location $\vec{\mu}$, the width (σ_x, σ_y) and the location confidence $(\Delta\mu_x, \Delta\mu_y)$. All accumulators sum over each pixel exactly once, and its commutative property allows an arbitrary order. After each pixel value has been examined, the properties are submitted to the output as a long vector. Overall, the data flow of the entire algorithm concentrates the data in two steps. It starts with a stack of frames where first ROIs with spots inside are filtered out, and finally only seven values are produced for each location.

Dimensioning of the Hardware

The knowledge about the data flow already gives us a description close to the actual implementation of the hardware. The last bits of missing high-level design are the type of parallelism to use and the data types for the multiple variables.

Parallelism The operators in the data flow are applied to two different entities: first entire frames until the spots have been found, and later ROIs cut out from the frames for feature extraction. This requires us to split the design into two statically scheduled pipelines that run a different number of clock cycles. The first pipeline consumes one pixel of a frame per clock cycle, and the total number of clock cycles is therefore equal to the number of pixels in the stack of frames. The second pipeline operates on 7×7 pixel ROIs, also with one pixel per clock cycles. Since ROIs can overlap, the number of pixels in ROIs could be larger than the total number of pixels for pathological recordings.

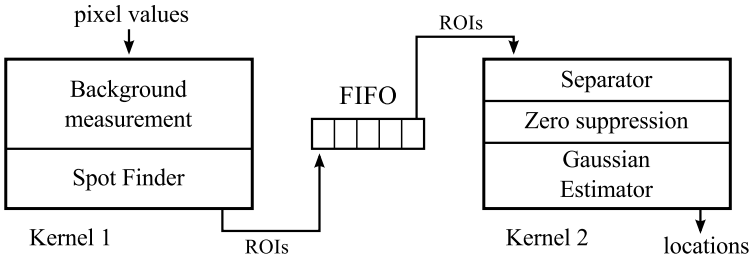


Figure 5.10: The pipeline design. Each pixel from the camera is fed through the first kernel, where the background gets measured and subtracted. When a spot is found, its ROI is sent to the second kernel. Here, the separator removes values from nearby spots that touch the ROI. The estimator finally calculates the position data. Both kernels are decoupled by a FIFO. [10]

In practice, however, a single frame contains only few spots and the second pipeline is expected to run only for a fraction of the time of the first one.

In Fig. 5.10 a sketch of both pipelines can be seen. In the language of the MaxCompiler library, these are called kernels. The first kernel contains all frame operations and the second kernel extracts the features from each ROI. The FIFO between both kernels stores the pixel values of each ROI per entry, along with its properties such as its x and y position in the frame, the level of the background that has already been deduced and the frame number. The FIFO implements the flexible interconnect to allow both kernels to run with a different number of active clock cycles.

The parallelism found here is pipeline (MISD) parallelism. All operations are performed in parallel, but the pipelines only have a single instance each. The total expected throughput is one consumed pixel value per clock cycles. For an estimated minimal clock frequency of 100 MHz, a single frame with 256×256 pixels would compute within 0.7 ms, which is much faster than the camera can produce them. Typical recording times per frame are between 100 and 10 ms, which is one or two orders of magnitude slower than the proposed hardware design. Multi-pipe (SIMD) parallelism for additional acceleration could be done by instantiating the first pipeline several times and splitting each frame among them, but is beyond the requirements of the application.

Numerics The intensities are typically recorded with an accuracy of 12 bits, and the cameras used for microscopy in the co-operating research groups of Prof. Christoph Cremer and Prof. Michael Hausmann followed this rule. For further processing, the data is stored in frames with 16 bit per pixel values, making it immediately compatible with general purpose computing. The input of the system is therefore an unsigned 16-bit integer with only the bottom 12 bits containing information. The chosen number encodings can be seen in Table 5.1. They follow from the input encoding and the steps of the algorithm.

For background elimination the pixel values are smoothed exponentially over time and divided by the factor N in the process. With $N = 8$, the resulting numbers will have 3 fractional bits. To also allow for $N = 16$, the number of fractional bits was set to 4. Since the background map will have a similar range as the input, leading bits cannot be omitted. The opportunity to stay with fixed-point numbers is expected to keep the resource footprint small for background elimination and spot findings. The square root to estimate the width of the noise cannot be represented with a fixed number of fractional bits exactly for most inputs. The error it may introduce to the locations will need to be measured later. An unsigned 16-bit integer is sufficient for 256×256 pixel frames to address a pixel in the background map. The pixel values passed in a ROI are obtained by subtracting the background are therefore share the same encoding.

Spot separation and zero suppression are operations that remove pixels or subtract pixel values by numbers of the same encoding and do not require a cast. For the accumulators, such as $\sum_{x,y} xq_{x,y}$

for the x location and similar for width and confidence, non-negative values are multiplied and summed up 49 times for a 7×7 ROI. The number of integer bits has therefore to be increased to 20 bits, taking into account the 12 bit of information found for each pixel and 3 bits for the encoding of x. To obtain the final location the accumulated values have to be divided by the total intensity Q. For this, a floating-point encoding with single precision was chosen. A location requires 8 bits for the pixel location and at most 10 bits for the sub-pixel position, the 23-bit fractional part for the significand is over-sized by at least 5 bits. However, the standardized format is also used for the output of the pipeline and simplifies communication with the host.

5.1.5 Implementation

The algorithm presented in the previous section was implemented on an FPGA board in a MaxWorkstation built by Maxeler Technologies. The workstation houses the CPU motherboard with the FPGA board on top, connected via PCIe 8x. The CPU embedded in the MaxWorkstation is an Intel Core i5-750 processor with four physical cores at 2.66 GHz and 4 GB of DDR3 RAM.

For the description of the dataflow graph and the generation of the intermediate VHDL representation MaxCompiler 2011.3 was used. The bitfile containing the configuration for the FPGA was synthesized with the Xilinx Integrated Software Environment (ISE) 13.3 for a Xilinx Virtex-5 LX330T FPGA. On the FPGA board, 12 GB of RAM are available for access from the FPGA, but were not used for this application.

Host code

The application consists of a software and a hardware part. The software controls the hardware and acts as an interface for the data streams. It provides the imagery to the FPGA and stores the location it receives on disk. The hardware part is encoded in a bitfile that defines the configuration of the FPGA. It is pushed to the FPGA by the software during initialization.

The software program, also known as host code because it runs on the CPU of the host system, was written in C++ on Linux and executes the parts of the algorithm that can hardly be accelerated with an FPGA. After it configured the FPGA with the bitfile, the host code sets the parameters of the algorithm such as the total number of images, the threshold for the spot finder and the total number of cycles the pipeline should run. For localization microscopy, it converts the image data from TIFF or KDF-encoded recordings into a three-dimensional array of 16-bit pixel values and streams them continuously to the FPGA. On the receiving side, the locations are visualized in a super-resolution image or can be stored in a Comma-Separated Values (CSV) file for further processing.

use case	number encoding
input pixel values	unsigned 16-bit integer encoding
pixel location	unsigned 16-bit integer encoding
background map	signed fixed-point encoding with 16 integer and 4 fractional bits
ROI pixel values	signed fixed-point encoding with 16 integer and 4 fractional bits
accumulators	signed fixed-point encoding with 20 integer and 4 fractional bits
division by Q	floating-point encoding with single precision
output location and features	floating-point encoding with single precision

Table 5.1: Numerical encoding used for the localization algorithms on hardware. Except for the last step all operations can be done with fixed-point encodings, leading to a smaller resource footprint than floating point (see also Fig. 4.12 on p. 61).

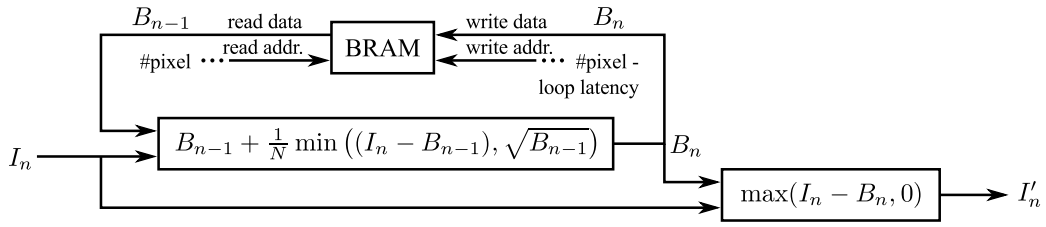


Figure 5.11: Hardware implementation of the background removal. The design of the statically scheduled pipeline consists of a BRAM buffer that holds the background map, and the pipelined implementation of the formulas that estimate and subtract the background pixel by pixel.

In a pipelined design data is sent to and received from the hardware simultaneously. The host code therefore has to perform two I/O operations at the same time. After the hardware was initialized, two POSIX threads are created, one for each direction of the data flow. The data is send from and received to a buffer in the host memory through calls to a C library offered by Maxeler Technologies. Internally, the contents of the buffers are sent to and received from the hardware through DMA transfers on the PCIe bus.

Before the implementation of the hardware was started, the functionality was tested with a Matlab program that closely resembles the hardware design for bugs and accuracy of the calculation. The hardware part consists of the compute intense tasks in two statically scheduled pipelines as shown in Fig. 5.10, and its implementation will be described below.

Background removal

To remove the background a map of its level has to be estimated and subtracted from each frame. The pipeline for this process can be seen in Fig. 5.11. The current background map is stored in BRAM and hold exactly one frame. Hence, the BRAM acts as a ring buffer for the background map.

The read and write address are generated by counters that are incremented with each clock cycle and wrap around if they hit the number of pixels per frame. The part of the pipeline that calculates the background B_n forms a loop with the BRAM. Its latency equals the offset between the address generators for read and write access. With the square root being the most compute intense operation, the latency of the loop is 48 clock cycles.

Spot detection

After the background has been subtracted, a blurred copy of the resulting image is made and every pixel is tested whether it is a local maximum and if its value exceeds the threshold. The source code for the detection of a maximum is shown in Listing 5.1. It builds a pipeline that accepts one pixel value per clock cycle and produces a stream of one-bit Boolean values that indicate whether a maximum was found at the current pixel position.

To determine if a value is maximal in its environment, a 7×7 pixel ROI is created from the input stream in the first set of nested for-loops. Negative stream offsets are allowed by MaxCompiler and are resolved when the design is globally scheduled at the beginning of a build. In the second set of nested for-loops, every value in the ROI is compared to the central value. If all other values are less or equal than the central value, the function returns true. Note the usage of an intermediate result `intermResult` for the comparison between the loops that decreases the latency of the pipeline. It forces the comparator part of the pipeline to form a tree with an intermediate level instead of building an unbalanced tree with only one long branch. A similar approach was chosen for the blurring with a 3×3 pixel matrix.

```
KArrayType<HWVar> roiType = new KArrayType<HWVar>(pixelFracValueType, roiSize);

HWVar isLocalMax(HWVar pixel) {
  // Build a ROI around the current pixel
  KArray<HWVar> roi = roiType.newInstance(this);
  for(int y = 0; y < roiEdgeLength; y++) {
    for(int x = 0; x < roiEdgeLength; x++) {
      roi.connect(y * roiEdgeLength + x,
             stream.offset(pixel, x - roiRadius + (y - roiRadius) * imgWidth));
    }
  }

  // Does "pixel" have the maximum value in the ROI?
  HWVar localMax = constant.var(true);
    for(int y = roiRadius - 1; y <= roiRadius + 1; y++) {
      HWVar intermResult = constant.var(true);

      for(int x = roiRadius - 1; x <= roiRadius + 1; x++) {
        int roiIndex = y * roiEdgeLength + x;
        if(y != roiRadius || x != roiRadius) {
          intermResult = intermResult & (roi[roiIndex] <= pixel);
        }
      }

      localMax = localMax & intermResult;
    }

  return localMax;
}
```

Listing 5.1: Function in MaxJ that builds a pipeline for local maximum detection.

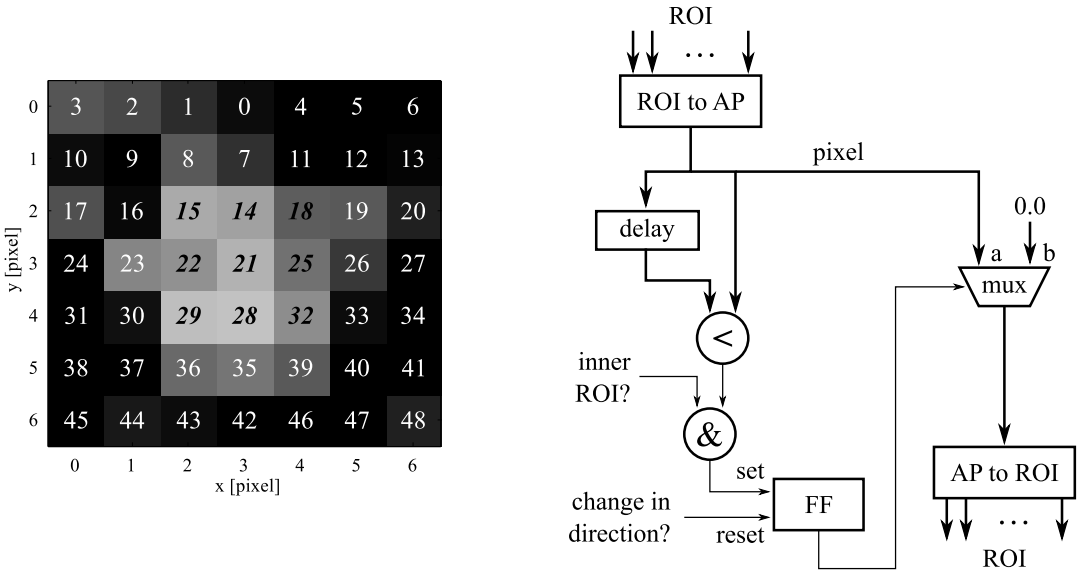
When a spot is found, its ROI is sent to the output as a whole, where it will get enqueued into the FIFO for processing by the second kernel. For a 7×7 pixel ROI and 20 bits per pixel, the width of the word is 980 bits wide. Additionally, the image number, the position of the ROI and the background level are submitted.

When the image stack has been analyzed, the last line of pixels in the last frame is used to generate empty ROIs with the frame number set to a negative number. These ROIs are needed due to limitations of the MaxCompiler software libraries that require the amount of data for a DMA transfer to be known in advance. The empty ROIs pad the last transfer and ensure that the last DMA buffer that still contains some valid locations is flushed and does not cause the final transfer to the host to stall. The invalid locations are identified in the host code by the negative frame number and are discarded.

Spot separation

For the separation of close-by spots the hardware needs to scan the ROI from the center to the outside horizontally, vertically, and horizontally again. Pixel values behind a local minimum are likely to belong to a nearby spot and need to be discarded. Since the Airy disk is flat at the center, noise may trigger such a decision by accident. The inner 3×3 pixels are therefore protected and left unaltered in any case.

The order of access for the horizontal scan is shown in Fig. 5.12a. Given a position (x,y) and a ROI with radius r_{ROI} and width $w_{\mathrm{ROI}} = 2r_{\mathrm{ROI}} + 1$, it can be pre-computed before hardware synthesis in MaxJ with the following formula.



(a) horizontal ROI access order

(b) core of the separator design

Figure 5.12: ROI hardware separator. Signals from close-by ROIs are removed by scanning the ROI from the center to the outside and setting all values to zero after a local minimum. The values of the inner 3×3 pixels are protected against modification.

$$\text{access order} = y \times w_{\text{ROI}} + \begin{cases} r_{\text{ROI}} - x, & \text{if } x \leq r_{\text{ROI}} \\ x & \text{otherwise} \end{cases} \tag{5.42}$$

The design for one direction is shown in Fig. 5.12. First, the pixels in the ROI have to be re-ordered in the direction of the access pattern (AP). This operation is done by selecting the pixels of the buffered ROI in the order given in Eq. 5.42 with a multiplexer (ROI to AP). The current pixel value is then compared with the previous one that has been delayed to arrive in time. If the current pixel value is greater than the previous one, a local minimum has been found. If the pixel is also not an inner one, a flip-flop is set that causes all following pixel values to be set to zero until the direction of the scan is changed. Last, the ROI is buffered again for the following logic.

For the re-ordering, a counter is needed that tracks the number of the pixel in the ROI. For the decision whether the direction changed or if the pixel is an inner pixel, a second counter tracks the x position. For the vertical scan, the hardware was adjusted accordingly.

Feature extraction

A set of accumulators is used when scanning the ROI pixel by pixel for the determination of the total intensity Q, the location $\vec{\mu}$, the width (σ_x, σ_y) and the location confidence $(\Delta\mu_x, \Delta\mu_y)$. For the width the formulas were re-written to not depend on the location for each iteration, such that the accumulators for width and location can work in parallel. The final form for σ_x (Eq. 5.46) does not contain $\vec{\mu}$ in the scope of the accumulator ($\sum$), and hence $\vec{\mu}$ can be calculated independently and is subtracted only at the end. The equations for μ_y and σ_y are obtained by substituting all x for y.

$$Q = \sum_{i,j} q_{i,j} \qquad \mu_x = \frac{1}{Q} \sum_{i,j} q_{i,j} x_i \tag{5.43}$$

$$\sigma_x^2 = \sum_{i,j} \frac{q_{i,j}}{Q} (x_i - \mu_x)^2 \tag{5.44}$$

$$= \frac{1}{Q} \sum_{i,j} x_i^2 - \frac{2\mu_x}{Q} \sum_{i,j} q_{i,j} x_i + \mu_x^2 \tag{5.45}$$

$$= \frac{1}{Q} \sum_{i,j} q_{i,j} x_i^2 - \mu_x^2 \tag{5.46}$$

The same process was done for the calculation of the confidence. Because zero suppression effectively shrinks the ROI, only summands for pixels with a non-zero intensity after background subtraction are added up.

$$\Delta\mu_x^2 = \frac{1}{12Q} + \sum_{i,j} \left(\frac{x_i - \mu_x}{Q} \right)^2 (q_{i,j} + B) \tag{5.47}$$

$$= \frac{1}{12Q} + \frac{\sigma_x^2}{Q} + \frac{B}{Q^2} \sum_{i,j} (x_i - \mu_x)^2 \tag{5.48}$$

$$= \frac{1}{12Q} + \frac{\sigma_x^2}{Q} + \frac{B}{Q^2} \left(\sum_{i,j} x_i^2 - 2\mu_x \sum_{i,j} x_i + \mu_x^2 \sum_{i,j} 1 \right) \tag{5.49}$$

All features except for the confidence are calculated on intensities before spot separation. Hence, two sets of accumulators are instantiated, one set for $(\Delta\mu_x, \Delta\mu_y)$, and one set for the remaining features.

The outputs of the accumulators are connected according to the formulas above and operate in units of pixels. After all pixels of a ROI have been calculated, the root of the squared widths and confidences is taken and the result is sent to the host computer through its PCIe interface.

The results of the feature extraction are checked for plausibility in the host code. Otherwise, photon noise that was confused with a spot by the spot finder could end up as a wrong result. The limits on the features for valid spots require the spot to have a total charge $Q > 0$ and a width $\sigma_x, \sigma_y > 0$. The localization accuracy must be better than 0.7 pixels in both directions to ensure a fit with sub-pixel precision. Finally, the separator was allowed to only remove 30% of the total charge, as our simulations showed this to be the limit for spot separation with sufficient accuracy [10].

Visualization

The command line version of the host code produces a list of locations, encoded as CSV, for further analysis. To allow the operator an immediate visual feedback, a Graphical User Interface (GUI) was implemented. The previous host code was packed into a software library to be called from the GUI. The GUI itself was written with the platform-independent Qt toolkit [137]. A screen-shot is presented in Fig. 5.13.

With the super-resolution image having a resolution increased by a factor of ten in both directions, the image generation took more time at first than the accelerated analysis. Each location had to be drawn as a Gaussian distribution with the fit confidence as the width. This imposed a high load on the floating-point unit of the CPU. To mitigate the bottleneck the Gaussian profiles

where computed only once for each fit confidence, stored in a cache and drawn on top of the plot if the same confidence occurred again. The confidence was rounded to an integer in the coordinates of the super-resolution plot, the introduced aberration by rounding remained invisible to the human eye. For the cache a C++ map (`std::map`) was used to look up the Gaussian distribution with the confidence as the key.

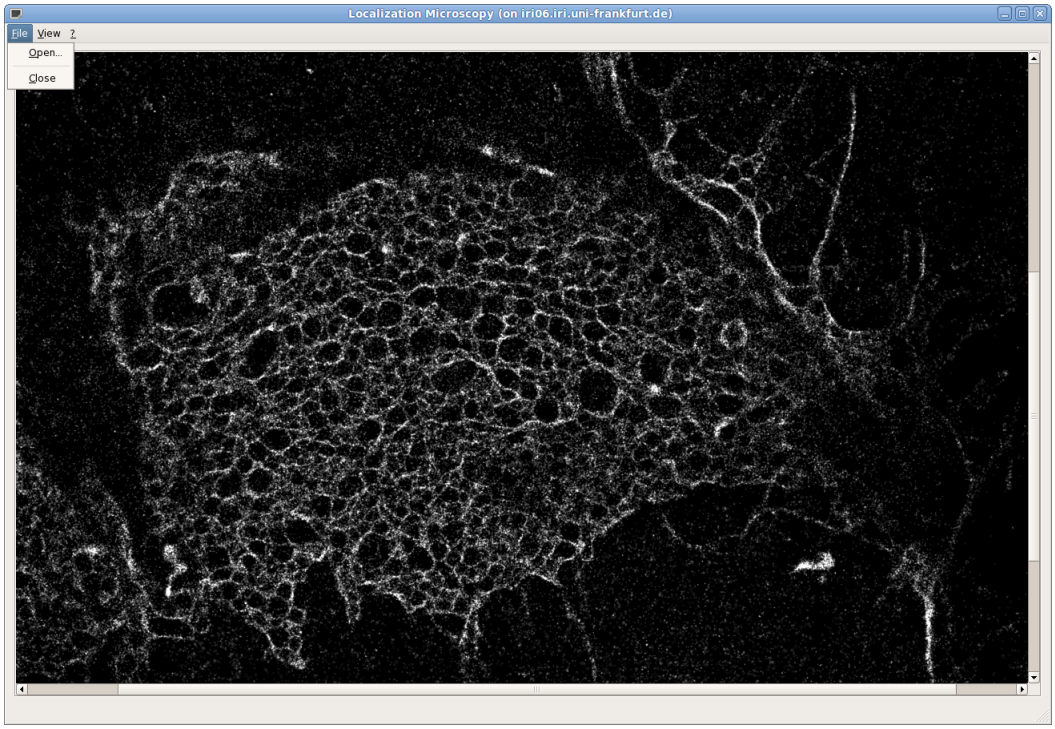


Figure 5.13: Screenshot of the Qt GUI. The program calls the host code as a library function to configure the FPGA and to stream the image stack. Afterwards, it renders a super-resolution image from the location results.

5.2 3D Electron Tomography

The second of two applications that was accelerated is image reconstruction for computed tomography. A proof-of-concept was implemented that shows how computed tomography can benefit from a dataflow design on an FPGA and that it performs better than an implementation on a state-of-the-art graphics card. 3D electron tomography was chosen for the implementation, but the design can also work with other types of radiation.

Computed tomography (CT) aims to reconstruct a density distribution of a sample from a set of projections. It is widely used in medicine and biology to obtain 3D images after the sample was X-rayed from multiple directions and the projections were recorded with a 2D image sensor. Although X-rays are the most common, other types of radiation can be used such as visible light for semitransparent objects or – as for our application – electron rays in an electron microscope.

An overview of the image recording process is shown in Fig. 5.14 for electron tomography. Inside of an electron microscope, the volume of the sample is illuminated from one side with electrons (e^-) and a projection is cast on the image detector. After the projection image has been read out, the sample is tilted by a projection angle φ and the next projection is produced, until the sample was illuminated from about 50 directions. For electron tomography, the ray is often not perfectly perpendicular to the tilt axis and will feature a non-zero declination angle α. Depending on the microscope optics, it may also be slightly twisted between sample and electron detector.

The volume is mathematically discretized into cubic voxels for the reconstruction of its density distribution. Due to the mentioned aberrations from a perfectly aligned electron ray, a layer of these voxels will be projected on multiple lines of the detector, even if voxels and detector pixels have the same edge length. Therefore, the problem of reconstructing the density distribution of the sample cannot be divided into a stack of 2D reconstructions from individual lines of detector pixels as it is usually done for X-ray tomography. Reconstruction for electron tomography is harder to parallelize and its solution already includes a solution for other types of CT.

Examples of the capabilities of computer tomography can be seen in Fig 5.15. With electron rays the resolution of the reconstructed 3D volume approaches 5 nanometers. Electron tomography provides insights into microscopic samples such as biological cells and their organelles and has become an important imaging method for research.

5.2.1 Reconstruction Algorithms

When an electron beam travels through the sample its intensity is weakened by the density of the material. The decrease is quantified by the detector and is proportional to the integrated density along the path of the beam. If the density of each voxel v_i was known, the intensity decrement p_i on the detector would hold equation 5.50 for each pixel. In the future we will always assume that p_i is the difference between the unobstructed intensity and the measured intensity for detector pixel i. The sum runs over all voxels, the weighting factor w_{in} indicates whether and how much voxel n was hit by the ray ending in pixel p_i. For most combinations of rays and voxels the weighting factor will be zero because the ray missed the voxel.

$$0 = p_i - \sum_{n=1}^{N} w_{in} v_n \qquad (\forall i) \tag{5.50}$$

The formula is a discretized version of the Radon transformation. It transforms a scalar function defined in a multi-dimensional number space to its projections from every direction and can be exactly inverted. In practice however, only a limited number of projections can be obtained from an electron microscope. For these projections the system of linear equations is under-determined,

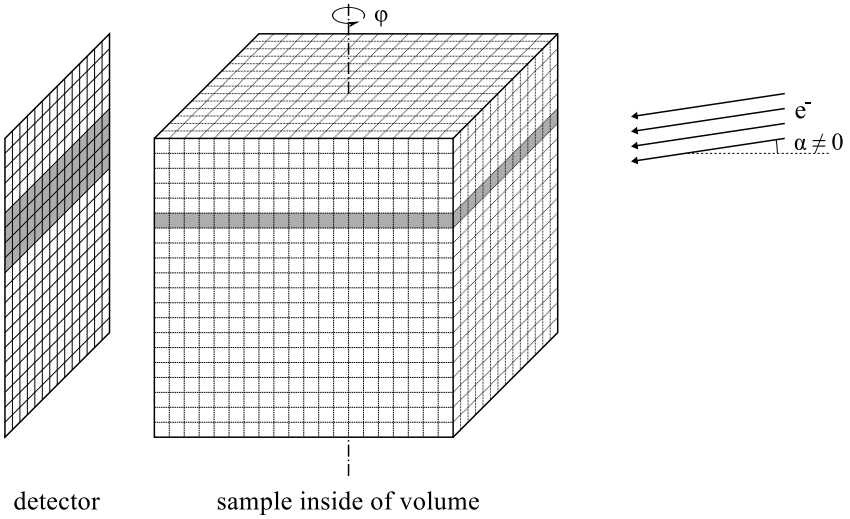


Figure 5.14: Overview of an electron tomography recording. The sample inside the discretized volume is projected from multiple angles φ_k to a CCD camera in the electron microscope. A layer of voxels is projected on a band of detector pixels for non-zero declination angles $\alpha \leq 10°$ or twisted beams. [11]

because the volume has many more voxels with unknown densities than there are signals available from the detector pixels for projection angles φ. As a consequence, the distribution of densities in the sample cannot be calculated analytically from Eq. 5.50 alone.

The Algebraic Reconstruction Technique (ART) starts with an empty volume of voxels and refines the voxel densities iteratively to match the projections. It was invented in 1970 by Gordon et al. [138]. The technique is based on previous works presented in 1937 by Stefan Kaczmarz [139]. The algorithm consists of three steps that are repeated for each voxel in the reconstructed volume, each pixel on the detector, and each projection. It starts with an empty volume.

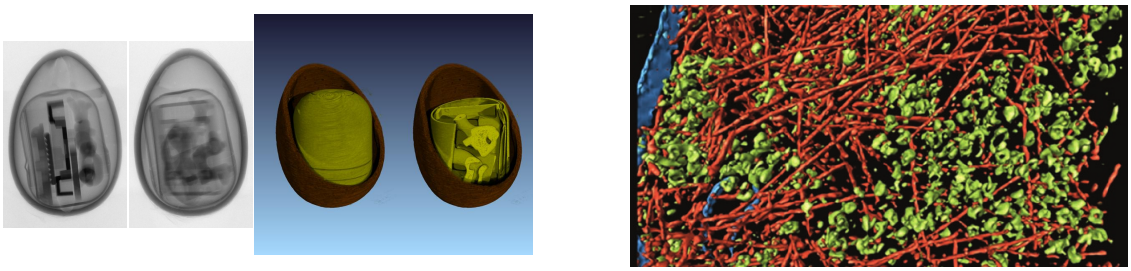
1. A virtual projection of the volume is calculated for the current pixel. The ray that ends in this pixel is partially absorbed by the voxels of the current iteration. This step is called the (virtual) forward projection.

2. The residue for the pixel is calculated by subtracting the pixel value from the measurement from the virtual one.

3. The voxel densities in the virtual volume along the ray's path is adjusted to match the measured pixel value. The voxel densities are increased by the equally spreaded residue. This step is called the back projection.

By repeating the process for every pixel i and every measured projection k, the calculated densities in the voxels are expected to approach the real values. The iteration formula is given in Eq. 5.51. In the nominator the difference between measured and calculated forward projection is given for a pixel value p_i, and the difference is applied back to the voxels. The relaxation parameter λ is chosen within the range $0 < \lambda \leq 1$. The more accurate the calculation, the closer λ can approach 1.

$$v_j^{(k+1,i)} = v_j^{(k,i)} + \lambda w_{ij} \frac{p_i - \sum_{n=1}^{N} w_{in} v_n^{(k,i)}}{\sum_{n=1}^{N} w_{in}^2} \tag{5.51}$$



(a) Surprise egg from X-ray images [140]

(b) Eukaryotic cell, electron tomography [141]

Figure 5.15: Examples of 3D reconstruction from 2D projections. Tomography may be known best from computer tomography with X-rays, an example with a surprise egg is shown in (a). Electron tomography uses an electron microscope and has a resolution of about 5 nm. It can be used to reconstruct the architecture of biological cells (b). Here, the actin network of the cell was colored red afterwards.

ART is computationally expensive. Every ray that ends at a detector pixel requires a full iteration. The Simultaneous Algebraic Reconstruction Technique (SART) was invented in 1984 by A. Andersen and A. Kak [5] to accelerate ART. It computes an entire virtual forward projection for all pixels before it is compared to the measured projection P_{φ_k} at angle φ_k and then projects the difference image back into the volume at once (Eq. 5.52).

$$v_j^{(k+1)} = v_j^{(k)} + \lambda \frac{\sum_{p_i \in P_{\varphi_k}} \left(\frac{p_i - \sum_{n=1}^{N} w_{in} v_n^{(k)}}{\sum_{n=1}^{N} w_{in}} \right) w_{ij}}{\sum_{p_i \in P_{\varphi_k}} w_{ij}} \tag{5.52}$$

As before the difference between virtual forward projection and measured projection is calculated in the inner fraction, while the outer parts describe the back projection for all pixels for tilt angle φ. Note that the factor w_{ij} is zero for most combinations of ray i and voxel j, since most voxels do not shadow the given pixel. For an implementation on a computer, both forward and back projection are calculated by following a given ray through the volume and by integrating the voxel densities along its path, skipping the calculation of most zero w_{ij}. Alternatively, the volume can be read linearly voxel by voxel and the densities are accumulated for the shadowed pixels during projection.

The rays for SART do not have to be parallel, they may also all stem from the same point or can have a different configuration. For electron tomography we will assume parallel beams.

5.2.2 State of the Art

The first implementations of computer tomography were two-dimensional reconstructions and consisted of back projections only. A sample was imaged with X-rays on a one-dimensional sensor for multiple angles. The sample was then reconstructed by accumulating the back projections on top of each other. Since the virtual forward projection is omitted, all back projections are applied with the same magnitude. For a point-like density distribution in the sample, the reconstruction shows a spot that is blurred with a starred shape (Fig. 5.16). The star has two spikes for each direction of projection.

The technique of *filtered back projection* was invented to mitigate the blurring effect caused by the summed back projections. A high-pass filter is applied on the reconstructed image and sharpens the reconstruction. The projection slice theorem [142] allows the application of the filter on the one-dimensional recordings for the same effect, and greatly reduces the computational power

needed. The filtered back projection works especially well if many projections can be recorded. This is usually the case for X-ray tomography, but not for electron tomography. Filtered back projection can be easily computed in parallel with one thread for each projection. For 2D-images with 1000×1000 pixels and more than 1000 projections, the 2D image can be reconstructed in the order of a second on state-of-the-art devices. [143]

For tomography with a lesser number of measured projections available, ART and SART proved to reconstruct with a higher accuracy. The total number of projections in electron tomography must be balanced between reconstruction quality that benefits from an increase of projections and radiation damage of the sample that must be kept low. The need for processing power is further tightened by improvements of the resolution of the CCD sensors in the microscope. The SPIDER and WEB software packages have been developed in 1996 to incorporate multi-core processing into reconstruction for transmission electron tomography [144]. Later in 2004 the XMIPP software package further accelerated reconstruction by cluster computing for UNIX-like systems with full data format compatibility [145].

Further acceleration was achieved with graphics cards after they started to support floating-point arithmetic natively. Daniel C. Díez et al. showed in 2006 that SART and related techniques can be sped up by a factor of sixty to eighty with commercially available GPUs [7] compared to the performance of a single CPU. A machine with a single graphics card could therefore substitute an entire medium-range computing cluster. With further optimizations of the reconstruction algorithm an extra factor of about an order of magnitude could be achieved in 2010 on similar GPU hardware for transmission electron tomography with parallel beams by Wei Xu et al. [146] .

Only filtered back projection has been implemented on reconfigurable hardware so far. Miriam Leeser et al. ported the algorithm for parallel beams of X-rays in 2005 [6] on a Xilinx Virtex 2000E FPGA and achieved a speed-up of 100 for 2D reconstruction. SART encompasses the calculations of forward and backward projections and was not implemented on reconfigurable hardware yet.

5.2.3 Analysis of the Algorithm

In order to implement the algorithm as a system of pipelines we must find a dataflow description that fits the constraints of the FPGA and its periphery. We started with an implementation on the GPU given by the research group of Prof. Frangakis, an enhancement of the algorithm presented by Daniel C. Díez et al. [7] that was optimized for the architecture of modern graphics cards. It calculates forward and back projections by following each ray through the volume. The density of the volume is sampled along the ray in intervals of one third of the voxel edge length for integration. The volume is kept in the memory of the graphics card and therefore was restricted to about 2 GB at the time of development. For the forward projections, the voxel densities are integrated along the path of the ray. After comparison with the measured projections, the residue is projected back by

(a) Original (b) Reconstruction

Figure 5.16: Reconstruction with back projection only. A point-like density distribution was reconstructed by projecting back its (few) measured projections. The circular object in the middle is imaged with a star-shaped artifact and one line for each projection in the corresponding direction.

spreading it equally on each voxel along the path. On the FPGA, the voxel densities cannot be kept in the few megabytes available with BRAM storage, and therefore the volume has to be stored in the on-board DRAM.

The location of the volume storage in the DRAM leads us immediately to consider the access speed of the voxel densities. The transfer frequency of DRAM is limited to less than 143 MHz for random access and can only be read faster in a sequential order (Sec. 4.5). If we aimed for a hardware pipeline that processes one voxel per clock cycle, the pipeline would stall below 143 MHz, which is already at the low end of the capabilities of a modern FPGA. In case the number of FPGA resources allow for the design to be multi-piped, the limitation of random DRAM access becomes a bottleneck that prevents further acceleration.

Fig. 5.17 sketches in which order the voxels have to be accessed in the graphics card implementation. Following a ray through the volume, the arising access pattern is pseudo-random instead of linear. The direction of the rays changes with each change of recorded projection and can therefore not be globally re-arranged in the direction of the rays. Re-arrangement between projections, however, would introduce pseudo-random access again.

Modifications

The solution to the access limitation is to re-write the algorithm to process the volume voxel by voxel instead of ray after ray. For each voxel, the rays that are likely to hit it have to be found and tested for intersection. The virtual projection can be built by accumulating the pixel values for each ray separately, until the entire volume was read. In summary, the random access to the volume is traded for random access in the storage for the virtual projection. Because the projection is two-dimensional, its size only grows with the power of two of the volume edge length and fits into the BRAM storage, while the volume grows with the power of three and must be kept in DRAM. For the projection in BRAM, random access is equally fast as sequential access.

A given voxel can shadow up to four detector pixels for the common case that pixels and voxels share the same edge length (Fig. 5.17b). The corresponding four rays are found by projecting the center of the voxel onto the detector, and selecting the four closest pixels. The centers of the pixels then give the end points of rays to be tested for intersection. If ray and voxel intersect, the intersection length will be given by the distance between the point where the ray enters the voxel and the point where it leaves the voxel again. A voxel may only be hit by up to four rays, but a ray hits every voxel on its path and is cumulatively weakened by every voxel with a non-zero density. The length of the ray-voxel intersections has to be computed for both forward and back projection and must be repeated for every voxel for every projection.

The four detector pixels shadowed by a voxel are found mathematically by projecting the center of the voxel onto the detector. The direction of the electron rays is known from the experiment, such is the position of the current voxel and the orientation of the detector. Using an affine projection, the voxel can be transformed into the coordinate system of the detector and projected onto it. An affine transformation contains a linear matrix operation such as a rotation, and a translation by a fixed amount. The formula of the projection is shown in Eq. 5.53 with $\vec{v}$ as the position of the voxel, M as a 2×4 projection matrix and $\vec{p}$ as the position on the detector. It requires six multiplications and four additions in total. Here, the first three rows of M describe the linear transformation of $\vec{v}$ and the last row contains the translation vector.

$$\begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} \tag{5.53}$$

By using units of the detector pixels and rounding the x- and y-coordinate of $\vec{p}$ up ($\lceil\cdot\rceil$) and down ($\lfloor\cdot\rfloor$) to the nearest integer, we obtain the coordinates of the four closest detector pixels $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$ and $\vec{p}_4$. The coordinates of these pixels are then equal to the endpoints of the four rays that need to be tested for intersection with the current voxel (Eq. 5.54). Please observe that the components of the vectors differ only by one unit in a direction, for example $\vec{p}_2 = \vec{p}_1 + \vec{e}_y$.

$$\vec{p}_1 = \begin{pmatrix} \lfloor p_x \rfloor \\ \lfloor p_y \rfloor \end{pmatrix} \qquad \vec{p}_2 = \begin{pmatrix} \lfloor p_x \rfloor \\ \lceil p_y \rceil \end{pmatrix} \qquad \vec{p}_3 = \begin{pmatrix} \lceil p_x \rceil \\ \lfloor p_y \rfloor \end{pmatrix} \qquad \vec{p}_4 = \begin{pmatrix} \lceil p_x \rceil \\ \lceil p_y \rceil \end{pmatrix} \tag{5.54}$$

After the endpoints of the four rays have been determined, they need to be transformed into the coordinate system aligned with the volume. This requires another affine transformation with a 3×3 matrix on each end point before the rays can be tested for intersection (Eq. 5.55).

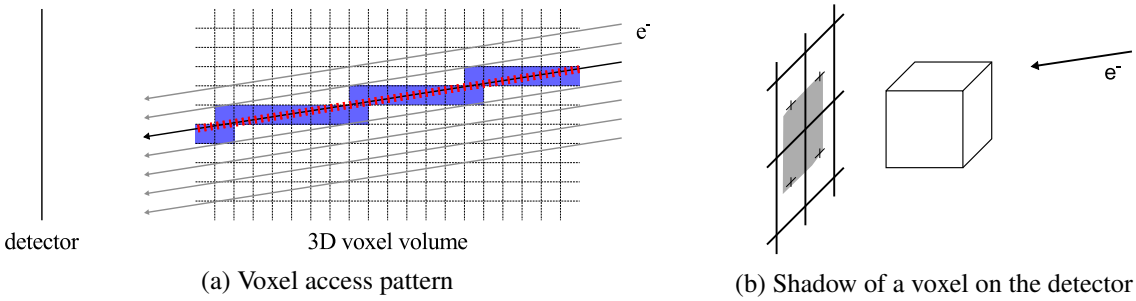
$$\vec{p}'_i = \begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix}_i = \begin{pmatrix} N_{11} & N_{12} & N_{13} \\ N_{21} & N_{22} & N_{23} \\ N_{31} & N_{32} & N_{33} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}_i \qquad i \in \{1,2,3,4\} \tag{5.55}$$

One of these transformations requires six additions and six multiplications. For all four pixel positions $\vec{p}_i$, however, the cost does not rise by a factor of four. The x- and y-coordinates of $\vec{p}_i$ have been rounded to integer values in units of pixels before and differ only by one. After the transformation for position $\vec{p}_1$ with both components rounded down has been computed, the other vectors are obtained more efficiently by adding either the matrix columns (N_{11}, N_{21}, N_{31}), (N_{12}, N_{22}, N_{32}) or both on top of $\vec{p}'_0$ (Eq. 5.56). Hence, no further multiplications are introduced and a total of 15 additions are needed.

$$\vec{p}'_2 = \vec{p}'_1 + \begin{pmatrix} N_{12} \\ N_{22} \\ N_{32} \end{pmatrix} \qquad \vec{p}'_3 = \vec{p}'_1 + \begin{pmatrix} N_{11} \\ N_{21} \\ N_{31} \end{pmatrix} \qquad \vec{p}'_4 = \vec{p}'_1 + \begin{pmatrix} N_{12} \\ N_{22} \\ N_{32} \end{pmatrix} + \begin{pmatrix} N_{11} \\ N_{21} \\ N_{31} \end{pmatrix} \tag{5.56}$$

The matrices M and N depend on the geometry of the microscope and can be computed on the host computer. They only change when a new iteration starts after a projection has been completed.

The graphics card implementation samples the volume along the ray in intervals of one third of the voxel edge length. For the FPGA implementation, which operates voxel by voxel, we cannot use this method. The voxel is already known and the rays have to be found instead. To obtain the



(a) Voxel access pattern

(b) Shadow of a voxel on the detector

Figure 5.17: Pseudo-random access pattern of voxels along a ray of electrons (e^-). The GPU version of the algorithms follows ray after ray and samples the voxel density three times per edge length (red markers). This creates a pseudo-random access pattern in DRAM (blue). For maximum throughput DRAM should be read sequentially voxel by voxel in bursts of data instead. A voxel at position $\vec{v}$ casts a shadow on up to four detector pixels with positions $\vec{p}_i$.

intersection lengths of the four rays and the voxel, we use four instances of *Smit's algorithm* for ray-box intersection [147, 148]. Here, the box is the voxel in the coordinate system aligned with its edges. The algorithm was adapted from the original description to handle rays that are not defined by the origin, as it is normally the case, but by their endpoint. Its description is given in Eqs. 5.57 to 5.63 and illustrated in Fig. 5.18.

$$\vec{t}_{\text{entry}} = (\vec{v}_{\text{min}} - \vec{p}')/\vec{r} \tag{5.57}$$
$$\vec{t}_{\text{exit}} = (\vec{v}_{\text{max}} - \vec{r})/\vec{r} \tag{5.58}$$
$$\vec{t}_{\text{min}} = \vec{\min}(\vec{t}_{\text{entry}}, \vec{t}_{\text{exit}}) \tag{5.59}$$
$$\vec{t}_{\text{max}} = \vec{\max}(\vec{t}_{\text{entry}}, \vec{t}_{\text{exit}}) \tag{5.60}$$
$$t_{\text{in}} = \max(t_{\text{min,x}}, \max(t_{\text{min,y}}, t_{\text{min,z}})) \tag{5.61}$$
$$t_{\text{out}} = \min(t_{\text{max,x}}, \min(t_{\text{max,y}}, t_{\text{max,z}})) \tag{5.62}$$
$$l = \max(t_{\text{out}} - t_{\text{in}}, 0) \tag{5.63}$$

The position and dimensions of the voxel are given by its opposing corners $\vec{v}_{\text{min}}$ and $\vec{v}_{\text{max}}$, where $\vec{v}_{\text{min}}$ points to the corner with the numerically smallest vector components. The ray is given by the endpoint $\vec{p}'$ on the detector and its normalized direction $\vec{r}$. The quotient $1/\vec{r}$ and the three-dimensional minimum function $\vec{\min}()$ and maximum function $\vec{\max}()$ are calculated component by component.

The ray-voxel intersection length l is computed by first calculating the intersection of the ray with the six layers that border the voxel. The positions of the respective six intersection points are stored in $\vec{t}_{\text{min}}$ and $\vec{t}_{\text{max}}$ as multiples of the direction $\vec{r}$, starting from the endpoint $\vec{p}'$. t_{min} and t_{max} also contain the information where the ray hits the voxel, but must be sorted first in Eqs. 5.59 and 5.60 to not confuse entry and exit. The position of the entry of the ray into the voxel is then given by $p' + t_{\text{in}}\vec{r}$, and similar for the exit with t_{out}. Since $\vec{r}$ is normalized to a length of one, the intersection length is the difference of t_{out} minus t_{in}. If the ray misses the voxel, the difference will be negative. In this case, it is set to zero in Eq. 5.63.

Note that every variable is assigned only once. This makes the computation immediately transferable to a synchronous pipeline in dataflow computing.

Data Flow

We can immediately infer a circular data flow from the iterative nature of the algorithm. The currently reconstructed volume is projected onto a virtual projector, the measured projection is

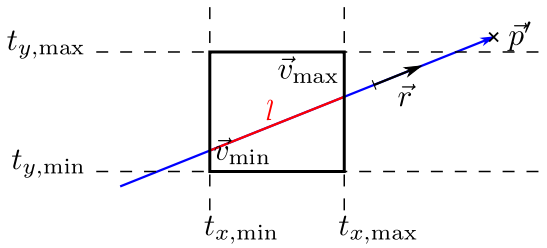


Figure 5.18: Ray-voxel intersection. The densities of the voxels are integrated along the path of the ray. To calculate the intersection length l (red), all coordinates are first transformed into the coordinate system aligned with the voxel (black box). Afterwards, the intersections t of the ray with the layers of the voxel are determined as multiples of $\vec{r}$, sorted, and subtracted (Eq. 5.57 to 5.63). The drawing is reduced to two dimensions for clarity.

compared with the virtual projection, and the difference is projected back into the reconstructed volume. Then, the next iteration continues. The high-level dataflow can be seen in Fig. 5.19. It also gives a first hint for the implementation.

At runtime, the design starts with an empty volume in the DRAM. It is read linearly voxel by voxel and fed into logical block A, where the four rays that may hit the current voxel are determined. A hit consists of the intersection of ray and voxel, and its length corresponds with the weights w_{ij} in Eq. 5.52. In block B these hits are multiplied with the voxel density and accumulated for each pixel they shadow on the virtual detector. Here, the virtual projection is gradually generated line by line. When a detector line has been fully computed, it is compared with the measured projection from the host computer. The residues are stored in block C for lookup and are required for the back projection. The last element of the circular dataflow is again a test for the ray-voxel intersection in block D. This time, however, the residues are read from block C and applied onto the reconstructed voxel density from the beginning of the iteration. An iteration for one projection is done when all voxels have been processed and the next projection can be started. In the last iteration the voxel values are sent to the host computer and constitute the final reconstruction. Block A and B form the logic for the forward projection, while block C and D apply the back projection onto the volume.

The dataflow allows for all stages to operate in parallel most of the time. The only delay occurs if block B has not yet accumulated the shadows of all voxels that belong to the first line. In this case block C must wait until the line is completely computed and can be compared with the measured projection. The higher the declination angle, the more layers of voxels will contribute to a line on the detector and the longer the back projection in block C and D will have to wait (see also Fig. 5.14 on page 98). A small declination angle close to zero is therefore desired for maximum performance. We will see later in the implementation section how to decouple forward and back projection to account for their non-trivial synchronization.

All operations in the dataflow operate on the stream of voxel values, each forming one data item to process in the pipeline. For a 3D volume with edge length L, the number of operations on voxels is of order $\mathcal{O}(L^3)$, while the number of operations on pixels in a projection is only of order $\mathcal{O}(L^2)$. Hence, with L up to 1000, all operations that run on voxels are the most time-intensive and should be ported to the FPGA accelerator, while operations on the values of the detector pixels may stay on the CPU.

Dimensioning of the Hardware

The knowledge about the data flow already gives us a description close to the actual implementation of the hardware. The last parts missing from the high-level design are the type of parallelism to use and the data types for the multiple variables.

Parallelism The dataflow graph shown in Fig. 5.19 suggests that we can process all data with pipeline-parallelism, continuously streaming the reconstructed voxel densities from and to the volume storage in the DRAM. For this, the individual algorithmic blocks must fit into the same streaming model of one data item per clock cycle. We will start with a single pipeline that processes one voxel per clock cycle (MISD) and later extend the design with multi-pipeline parallelism (MIMD).

For ray-voxel intersection, we have already seen that the algorithm adapted from computer graphics is of linear nature without branchings or loops. It can therefore be expected to be implemented with a pipeline that accepts one voxel and ray per clock cycle and emits the length of the ray-voxel intersection. The matrix multiplications needed before are also small enough to be implemented into a pipeline directly without the need for a loop over the vector indexes. These matrix multiplications encompass the projection of the voxel onto the detector that are needed to

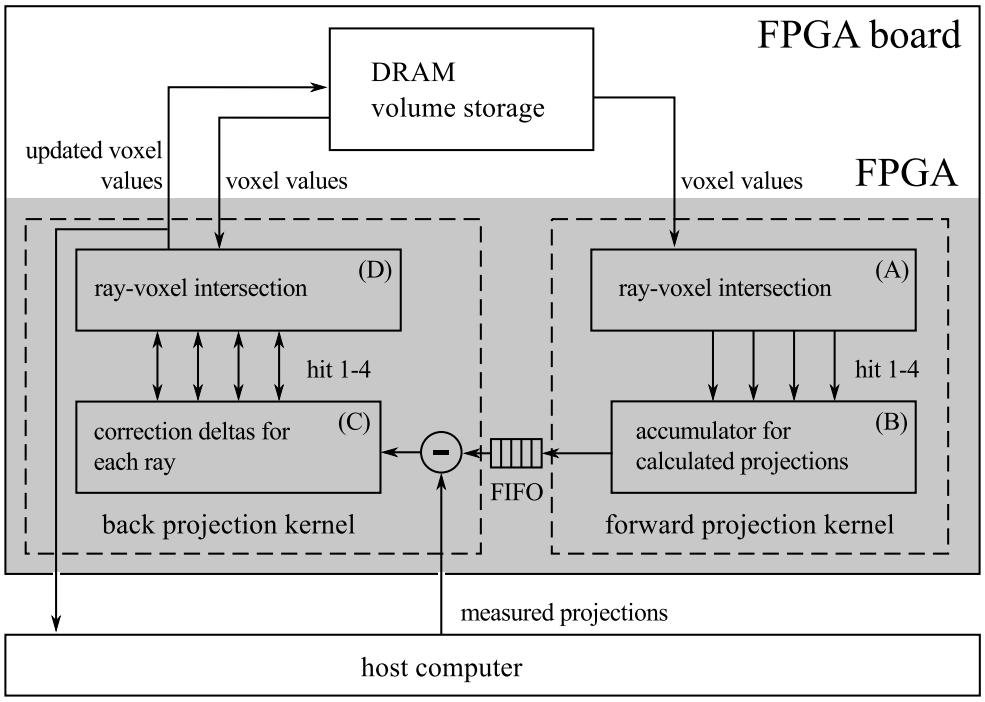


Figure 5.19: High-level dataflow between hardware components. The iterations of the reconstructed volume are kept in DRAM. Its voxel are piped through the FPGA, where the ray-voxel intersections (A) are computed to accumulate the forward projection (B). After the forward projection has been obtained, it is compared with the measured projection and the difference (C) is projected back into the volume (D). [11]

find the four rays to possibly hit the voxel, and the transformation from the coordinate system of the detector to the coordinate system of the voxel.

The virtual projection is accumulated in BRAM. Here, the voxel densities have to be integrated along the path of the rays through the volume. After the ray-voxel intersection was computed, we need to add the intersection lengths multiplied by the voxel density on top of the previous value of the affected virtual detector pixels to build up the forward projection. The pixel values are kept at a given storage location in dual-port BRAM, where they can be read and written at the same clock cycle. However, accumulating values in BRAM is complicated by the fact that BRAM introduces a latency between reading a value and writing it back. To increase a pixel value it has to be read first, then incremented and finally written back. For pipelined access this introduces a read-after-write conflict that leads to errors when the same storage location is incremented continuously for several clock cycles. Fortunately, the access pattern on the virtual detector in the DRAM allows us to avoid this problem with few extra hardware resources: The voxels are linearly read from the volume storage in BRAM, and after the affine transformations have been applied the access pattern to the pixels in the BRAM is still largely monotonic. The details can be found in the implementation subsection 5.2.4.

After the virtual projection was completely accumulated and subtracted from the measured projection, the residue storage is written only once per projection. After this, access remains read-only until the next projection, avoiding any read-after-write conflicts and preserving data consistency. The residues are read during back projection when the ray-voxel intersections are computed again and the residues are applied to the volume.

The voxels in the volume are read twice per iteration from the DRAM, first during forward projection and a second time when the density values are updated during back projection. Both

times the same ray-voxel intersection lengths are computed. Instead of computing the intersection twice, it would be computationally less intense to store them during forward projection and merely retrieve them during back projection. However, due to the time offset between forward and back projection, the amount of storage needed for the intersection length would require us to store them in on-board DRAM. Read and write access to this intersection buffer would then more than double the total memory access load. A bandwidth of 2.4 GB/s is already required with about 4 bytes for the voxel density, two reads and one write access per clock cycle for forward and back projection, and a target clock frequency above 150 MHz. The remaining DRAM bandwidth was instead kept free for multi-piping the design. The cost for this decision is an increase in resource usage for flip-flops, look-up tables and DSPs.

Numerics The dimensions of the number types in the entire design is largely given by the accuracy of the detector in the electron microscope and the size of the volume. The CCD sensor produces a 16-bit integer value for each pixel. The accuracy of a calculated voxel density is then determined by examining the back projection where the densities are reconstructed. Here, the error of ± 1 for the residue is spread equally to all voxels on the path of each ray. The length of a ray is at maximum $\sqrt{2} \times 1024$ for a volume with $2^{10} = 1024$ voxels in each direction and a diagonal ray through the volume assuming a low declination angle. Hence, the accuracy of a voxel density is at most about 2^{-13} with a 2-bit safety margin.

The positions of the (virtual) detector pixels and the reconstructed voxels can be addressed with integer indexes. For an edge length of 1024, a 10-bit integer is sufficient. For the affine projections, the determination of the accuracy is more difficult: The ray-box intersection is sensible to small errors, as the intersection of a ray that barely touches a voxel must be precisely calculated to not produce false negatives or positives. Especially rays where one component of the vector direction vanishes and the direction becomes parallel to one side of the voxel volume depend on a precise calculation of their endpoint, as a small change in its position makes the difference between a hit with a maximum intersection length equal or greater than one voxel edge length, and a miss with zero interaction [148]. In this case, the intersection length changes from a continuous function of the endpoint position to a discrete one.

To find the right accuracy for fixed-point number encoding, a simulation was carried out that compared the accuracy of the ray-box intersection length using a fixed-point number encoding with the intersection length obtained with a floating-point number encoding. The ray endpoint was randomly chosen with a distance of up to 2048 voxel, the voxel position was also randomly set and the ray direction was set with a random distribution such that it would hit the voxel with a chance of about 50 %. The simulation avoided the case where the ray direction would contain a component with an absolute value of less than 0.0005. In the unlikely event that a vector component of the direction happens to be smaller than that, it will be set to 0.0005. The magnitude of the errors introduced hereby will be given in the results section.

The numeric encoding chosen for the remaining parts of the design is given in Table 5.2. Most encodings are fixed-point encodings with a total number of 32 bits. When used for inputs or outputs, the host code will convert them between floating-point for the CPU and packed 32-bit integers for the PCIe bus. The only notable exceptions from this rule is the core algorithm for ray-voxel intersection, which is computed with a 50-bit fixed-point encoding with 30 fractional bits. The results of the analysis that lead to this encoding is shown in section 6.2. Furthermore, the density the of volume is read out during the final iteration encoded as floating-point numbers with single precision.

The use of fixed-point numbers limits the resource usage. The problem of reconstruction features a limited range for the numbers involved, mainly because all operations are related to the limited range of the CCD sensor. The only exception is the ray-voxel intersection that requires the

use case	number encoding
voxel density	fixed-point encoding with 13 integer, 19 fractional bits
voxel density read-out	floating-point encoding with single precision
transformation matrix	fixed-point encoding with 12 integer, 20 fractional bits
ray-box intersection	fixed-point encoding with 20 integer, 30 fractional bits
intersection length	fixed-point encoding with 12 integer, 20 fractional bits
physical detector pixel	unsigned 32-bit integer
virtual detector pixel, residue	signed fixed-point encoding with 16 integer, 12 fractional bits

Table 5.2: Numerical encoding used for the algorithms for electron tomography on hardware. All internal operations can be executed with signed fixed-point encodings, leading to a smaller resource footprint than floating point.

inverse of ray-direction to be computed, with the potential of producing arbitrarily large numbers. Since the inverse is pre-computed on the CPU for each iteration, we can calculate the inverse components of the ray direction on the CPU with double precision and convert them to fixed-point numbers afterwards, limiting the maximum error to a small and well-known magnitude that is smaller than the precision of the microscope.

5.2.4 Implementation

MaxCompiler 2011.3 was used for the description of the dataflow graph and the generation of the intermediate VHDL representation. The bitfile containing the configuration for the FPGA was then synthesized with the Xilinx Integrated Software Environment (ISE) 13.3 for a Xilinx Virtex-6 SX475T FPGA. On the FPGA board, 24 GB of RAM are available, equally distributed on six SODIMMS.

The discussion of the data flow in the previous subsection already gave us a high-level description of the hardware design. The functionality is divided into two domains, the first one for forward projection and the second one for back projection. For the implementation these domains must be mapped to one or more pipelines in hardware. Backward projection can only start after the first lines of the virtual projection have been completed. Due to a non-zero declination angle of up to 10°, the calculation of the first line of the virtual detection may require the traversal of multiple layers in the volume. Hence, the statically scheduled pipeline of the design has to be split between forward and back projection and must be decoupled with an (elastic) FIFO in between. As a result, the design is made of two kernels in the MaxCompiler source code that are dynamically scheduled by the manager. This design decision is already reflected in Fig. 5.19 on page 105.

Scheduling

The scheduling between kernels, DRAM and the host CPU is done automatically by the manager and the Linux kernel module of MaxCompiler. The kernels run when all enabled kernel inputs are non-empty and the outputs do not stall. Within the kernel pipelines, however, the orchestration of the dataflow is done manually. Contrary to the application for localization microscopy, we cannot resort to the principle of one data item for every clock cycle. The BRAM implementation of the projection accumulator in the forward projection kernel cannot transfer data to the back projection kernel at the same time. A parallel read and write access is needed to increment a value in the virtual detector, and both ports of the BRAM are therefore already occupied. The forward projection must be stopped to transfer the values (read access) and to zero the storage for future use (write access) to transfer the pixels of the virtual detector,

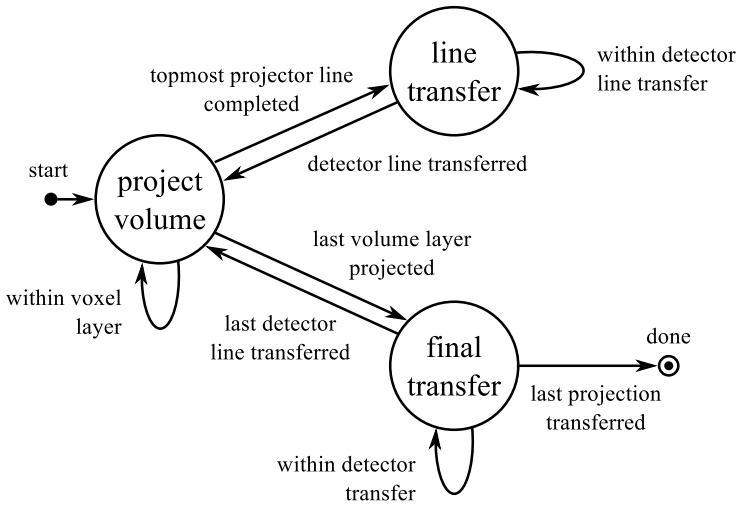


Figure 5.20: Forward projection FSM. The first state projects a layer of the volume onto the virtual detector ("project volume"). When done, the virtual projection is halted and one line of the virtual detector is transferred to the kernel for back projection ("line transfer"). When the entire volume was projected, the remaining lines of the detector are transferred ("final transfer") and the next projection is started.

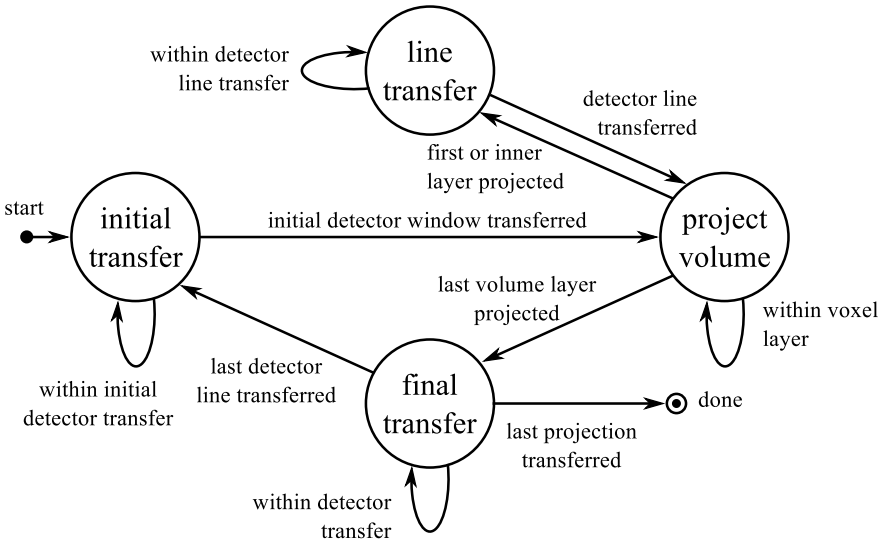


Figure 5.21: Back projection FSM. The first state blocks until the residues needed for the first voxel layer have been received ("initial transfer"). When done, the residues are projected back into the layers of the volume ("project volume"). After a layer was projected back, projection is halted and the next line of residues is received ("line transfer"). At the end of a projection the possibly remaining residues are consumed by the kernel ("final transfer")

The virtual projection has far less pixels than the volume has voxels. For a volume with 1024^3 voxels and a detector with 1024^2 pixels, the forward projection has to be halted for transfer for less than 1 ‰. The specific implementation of the BRAM accumulator allows to transfer multiple pixel values per clock cycle and reduces the down time of the forward projection even further.

The position of the window of virtual pixels and residues that have to be kept in BRAM slides monotonic with every new layer of voxels. When a layer of voxels has been processed, the shadow of the next layer advances downwards by at most one line of pixels on the detector. It is therefore sufficient to dedicate transfer time for just one detector line after a voxel layer was completed.

The control logic was implemented as a Finite State Machine (FSM). An FSM deviates from the dataflow model, but simplifies the formulation of the logic. It is integrated in the statically scheduled pipeline of the forward projection kernel and transitions its state every clock cycle when the kernel is running. The state transition graph is shown in Fig. 5.20. It consist of one state where the forward projection takes places, and two states for the transfer of the previously computed virtual detector pixels.

The volume is projected voxel by voxel onto the virtual detector during the initial state “project volume”. The state is kept until an entire layer of the volume was traversed, and the index of the voxel is increased to track the position of the voxels. After one layer was projected, the FSM transitions into the state “line transfer”. Here, the projection is paused to allow the transmission of a detector line to the back projection kernel. The transmission will only take place if the BRAM accumulator indicates that the calculation of the line was completed. Afterwards, the FSM will return to the projection state.

At the end of a projection, when all voxels of the volume have been traversed, the remaining lines of pixels in the virtual detector have to be transferred to the back projection kernel. This is done in the state “final transfer”. After the last line has been transferred, the state returns to “project volume” for the next iteration unless the projection was the last one. In the latter case, the FSM switches into the “done” state, where it remains until the FPGA is reset.

The transition graph of the FSM for the back projection kernel is shown in Fig. 5.21. Before back projection can start, all residues needed for the first layer of voxels must be available. Depending on the declination angle of the electron microscope, the shadow of the first voxel layer may fill the entire BRAM. During the first “initial projection”, the forward projection of the first layer is received and the residues are calculated accordingly by subtracting the real from the virtual projection. The amount of data needed is calculated by the host CPU and called the “initial detector window”. As soon as the data was transferred, back projection can start for the first layer of voxels in the state “project volume”. When the back projection was applied for the layer and if needed, the FSM halts back projection and performs the transfer for the next line of residues in the state “line transfer”. After the last layer has been projected, the state “final transfer” takes care of consuming any detector pixels from the forward projection kernel that have not been transferred yet between kernels and could otherwise stall the forward projection kernel. When done, the next back projection is ready to start and the FSM continues with the initial transfer for it. Finally, after the last iteration was projected back, the FSM stops and stays in the “done” state until the FPGA is reset.

The FSMs for the forward and back projection kernels control all other elements of the hardware. They keep track of the current voxel to be projected forward or back, as well as the pixel transfer between kernels. They also control the sliding windows where the forward projection is accumulated within, and the residues are obtained from for back projection. Each FSM needs to match the behavior of the other one in order to prevent the connecting FIFO from underflows or overflows whenever possible.

When a projection was finished, the FSMs also enable the inputs of both kernels that receive the properties of the next projection for one clock cycle. These are the matrices for the affine

transformations and the inverse direction of the electron rays. The data is pre-computed on the host CPU and sent to both kernels via PCIe.

External DRAM

The hardware design accesses the volume in the DRAM storage through three channels. The first kernel reads the voxel densities linearly to calculate the forward projection. The second kernel updates the voxel densities during back projection and needs to first read each voxel and then write the new value. The CPU does not have read or write access. Instead, the forward projection kernel treats all (uninitialized) densities as zero during the first iteration, and the back projection kernel sends the resulting voxels to the host CPU during the last iteration through an extra output using the PCIe bus.

All accesses to the DRAM are linearly and allow for burst access. The three address generators that produce the access pattern for the DRAM controllers can be instantiated in the Java class that describes the manager. Here, the DRAM and the kernels are instantiated and connected with each other and the PCIe bus. Thanks to the DRAM burst mode the lowest available DRAM frequency of 303 MHz could be chosen to ease the requirements for timing closure during hardware synthesis.

The linear access pattern compromises a range in the address space and the number of blocks to be accessed. These values are set during runtime by the host code on the CPU before the FPGA is started. The corresponding C function is provided as a library call by the Maxeler RT library. After that, the address generators run on its own on the FPGA. Their speed is controlled by the throughput of the connected kernels.

The size of a read or write burst is 384 bytes because the hardware combines the bursts of six DRAM SO-DIMMS for maximum performance. The width of the inputs and outputs of the kernels that are connected to the DRAM must be an integer divisor of the burst length to save hardware resources. The storage format was chosen to be a fixed-point encoding with 32 bits, which came closest above the theoretical precision.

Ray-Box Intersection

The implementation of the algorithm for ray-box intersection is very close to the description given in Eqs. 5.57 to 5.63, which is already in a format suitable for dataflow computing. The respective accuracies for the fixed-point numbers are listed in Table 5.2 in the previous subsection.

In very rare cases a voxel may be wrongfully missed by all four rays due to rounding errors. This can only happen if the rays are parallel to one of the voxel's sides and the voxel happens to be exactly in between two pairs of rays. The error introduced for the calculation of the forward projection can be neglected, since a ray traverses through many voxels. For the back projection, this miss can introduce high frequency noise in the volume. The partial pipeline for ray-box intersection therefore checks for a vanishing intersection length for all four rays. In this case, a rounding error caused all four rays to miss the voxel, and the individual intersection lengths are all set to 1/4.

During development it became apparent that this part of both kernel pipelines is responsible most of the time when the design did not reach timing closure in hardware synthesis. In this part of the pipeline the encodings for fixed-point numbers contain the largest number of bits and make the multipliers expensive in terms of hardware resources and timing constraints. Overall and as shown later in the results chapter, most DSPs in the design are required for this part of the algorithm. The MaxJ description was carefully tuned with extra pipeline registers to re-time the connecting lines between the DSPs. To reach the highest clock frequency for both kernels every pipeline stage had to be extended with one extra register. The pipeline stage before the multiplication with the inverse ray direction in Eq. 5.57 and 5.58 had to be re-timed with even two extra layers of pipeline registers.

Projection accumulator

After calculating the intersection with four rays for each voxel, the forward projection kernel multiplies them with the voxel density and accumulates the resulting shadow where it is casted on the virtual detector. After a line has received all shadows, its values are transferred to the back projection kernel, where the line of pixels is compared to the real projection obtained from the microscope. The traversal of the volume is chosen such that first the voxel within a layer are visited before the next layer is accessed (Fig. 5.14 on page 98). Hence, a layer of voxels shadows multiple lines of the detector, but not the entire detector, allowing us to keep only the affected detector lines in BRAM on the FPGA.

For a volume with 1024^3 voxels, the shadow of a layer on the detector fills a band with a width of up to $1024\sqrt{2}\sin(\alpha)$ pixels. For a maximum beam declination angle $\alpha \leq 10°$, this means that we have to keep at least 252 detector lines or about 1 MB in the BRAM. Storing the entire virtual detector would simplify the hardware, but is not an option: the storage needed would amount to 3.6 MB, which is more than half the entire BRAM available on a Xilinx Virtex-6 SX475T FPGA [149], and the same amount would be needed again to store the residues in the back projection kernel. Therefore, only the currently shadowed lines are buffered.

The design of the projection accumulator consists of three layers of hardware. The top-most layer manages the sliding band of shadowed pixels when the volume is traversed voxel by voxel. It also contains the logic to transfer every completely calculated line of the detector to the back projection kernel. The middle layer reads and writes to this storage and adds the received voxel shadows on top of the previously stored pixel value. Finally, the bottom-most layer provides the storage needed to accumulate the shadows of the volume.

The top-most layer receives the individual shadows from the four rays that can hit the current voxel. The pixels where these rays end are neighbors and form a 2×2 square on the virtual detector. The memory was therefore spit into four parts, one part for each combination of odd and even index for detector column and line (Fig. 5.17b on page 102). Each part then updates one pixel per clock cycle and acts as its own accumulator. The storage location is determined by using the line number modulo the number of buffer lines. This yields a ring buffer that just contains the detector lines that are shadowed at the moment by the current volume layer, plus some extra lines for the transfer. After a layer of the volume was traversed, it is checked whether a pixel line was left unmodified and is ready to leave the ring buffer during the following “line transfer” state of the forward projection FSM. The line is then zeroed and can be used again for a different line index.

The middle layer contains the accumulation logic and is instantiated four times per voxel, one time for every ray. The accumulation can be compared with filling a histogram. For a series of increments at random histogram buckets, the design challenge of such an accumulator is known as“scattered add” [150, 151]. It exhibits the following problems for dual-port BRAM:

- Limited resources for adders: With 1024^2 detector pixels, resources on the FPGA do not allow to consume one fixed-point accumulator for every detector pixels. Hence, only one fixed-point accumulator is used and the pixel values are stored in BRAM and retrieved from it during accumulation.

- Memory conflicts: An increment consists of a read access, followed by a write access to the same address for the updated value. BRAM has a non-zero latency and the value could have been updated in the clock cycle before, but the update could have not reached the BRAM yet. This read-after-write conflict can lead to lost updates.

- Clock speed: For an entire megabyte of BRAM, the read and write latency increases and requires either more clock cycles or a slowed-down clock frequency.

Fortunately, we can avoid all three problems after examining the access pattern on the virtual detector. The three-dimensional volume is traversed by incrementing the x index first to address the current voxel. When the x index wraps, the y index is incremented by one, until a voxel layer was visited. Only then is the z index incremented.

When only the x index of the current voxel in the volume is increased, the addresses of the four shadowed virtual detector pixels change monotonic because voxel and pixel positions are connected by an affine projection. In this case, the location of the detector value that has to be incremented by the shadow must be either the same as in the previous cycle, or the location advances and will not return to a previous one. It is therefore sufficient to only handle the case that a pixel is incremented consecutively. Otherwise, we can load from the BRAM, increment and write back without having to mind storage conflicts.

When the x index of the current pixel wraps, the same insight does not immediately hold true and a storage conflict could happen, since the wrap caused the pixel location to leap and to not move monotonically. The situation is sketched in Fig. 5.22a for a very small volume with an edge length of only 14 voxels. The x index wraps between time step 13 and 14. The wrap causes not the same, but the previous odd pixel to be hit again. The odd pixel that was hit in time step 6 the last time is hit again in time step 14, disturbing the monotonic pixel access pattern. Also, at time step 21, the same pixel is incremented that was already incremented in time step 13.

For the previous pixel to be hit again in the worst case, the rays must be almost parallel to the direction of the x index. We therefore can at least afford a latency of $21 - 13 = 8$ clock cycles for the BRAM to store an update safely. The allowed BRAM latency increases for volumes with larger edge length. The time t between repeated hits is given in Eq. 5.65 and was derived from Fig. 5.22a. All units are multiples of voxels for l_x and clock cycles for t.

$$t = \left\lfloor l_x - \sqrt{\frac{l_x^2}{4} - 1} \right\rfloor \tag{5.64}$$

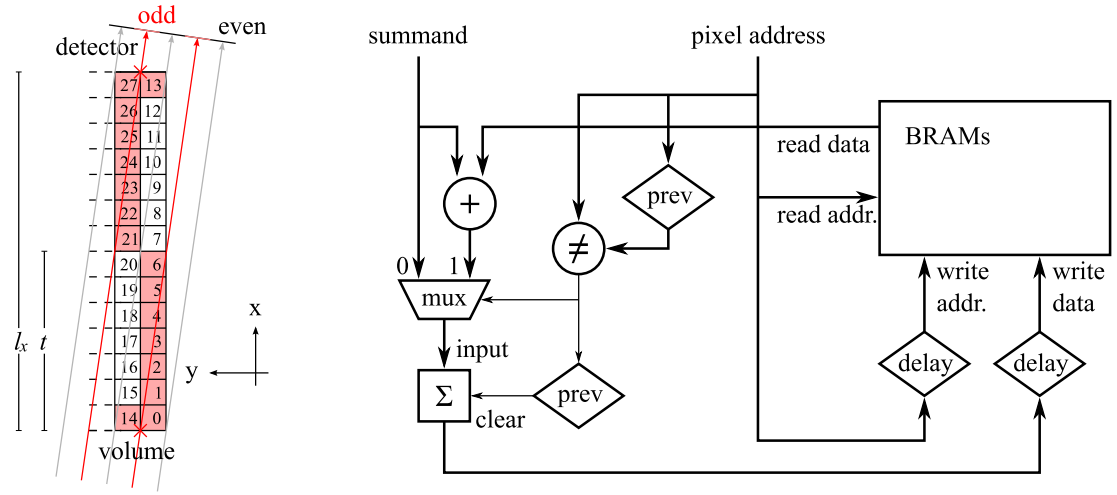
$$\simeq \frac{l_x}{2} \tag{5.65}$$

The time between repeated, non-monotonic hits t is about half the edge length in x direction. In practice the shortest edge of the volume is at least 200 voxels wide, hence the hardware has a comfortable constraint of 100 clock cycles for a round trip between write to and read from BRAM. The implementation requires only 13 clock cycles. We are left with a small modification of the projection accumulator to ignore increments by zero, because the same odd ray that intersects the current voxel at time step 21 for the first time with a non-zero intersection length will already be sent to the accumulator with a zero intersection length since time step 18.

When the y index wraps, a voxel layer was completed and the forward projection FSM switches into a transfer state and projection pauses. When the forward projection continues again all updates have been safely written to the BRAM.

The logic for the middle layer of the projection accumulator before static scheduling is shown in Fig. 5.22b. The pixel address is compared with the previous one ($\neq$ node) and the value at the address is retrieved from the BRAM. If the address has changed, the value from the BRAM is added with the summand (i. e. the shadow of the current voxel) and loaded into the previously zeroed fixed-point accumulator ($\sum$). If the address has stayed the same, only the summand is sent to the fixed-point accumulator. The result of the accumulation is sent to the BRAM. The delay is required to allow the loop in the design to be scheduled and depends on the latency of the BRAM and the other logic in the loop's path. In the final hardware design its latency is 13 clock cycles.

The logic for the BRAM accumulator contains logic that was not shown for increased clarity. It contains additional multiplexers to allow the read-out and zeroing of detector lines for line transfers



(a) Detector access pattern

(b) BRAM accumulator design before scheduling

Figure 5.22: The projection accumulator. The individual shadows of the voxels are accumulated on the virtual detector during forward projection. The access pattern on the detector is sufficiently monotonic to allow the latency of the BRAM to be mostly ignored, avoiding the general histogram problem (a). The hardware design only needs to check for sequences of accumulations to the same BRAM address. The multiplexers for the line transfers and the logic to ignore accumulations with a zero summand were left out for clarity (b).

to the back projection kernel. Also, the logic that ignores summands that are equal zero is not shown. This design is instantiated four times per voxel, one time for each ray.

The memory is assembled from many individual BRAM blocks on the lowest level of the projection accumulator. Xilinx ISE allows us to combine multiple BRAM slices on the FPGA into one big BRAM. This technique is helpful, but not sufficient for memory blocks that approach a quarter of a megabyte. The interconnects between the slices decrease the maximum achievable clock frequency drastically. The memory was therefore assembled manually in MaxJ and layers of pipeline registers were added to limit path lengths and fanout. The optimal size for a BRAM block in terms of clock frequency was found to be close to the hardware size of a BRAM at 4 kB with two stages of extra pipeline registers before and after its ports.

The final accumulator design is able to accumulate four values per clock cycle and integrates seamlessly with the remaining pipeline stages of the forward projection kernel. It is controlled by the FSM of the kernel and notifies the FSM when the last detector line of an iteration was transferred to the back projection kernel.

Residues Storage

The backward kernel receives the virtual projection from the forward projection kernel as well as the real projection and the total intersecting length of each ray from the host. After virtual and real projection were compared, the resulting residues need to be stored for lookup during back projection. Much like the projection accumulator, only a band of pixel lines is kept in memory. Also, the storage is spit into multiple parts to allow simultaneous read access for four residues per clock cycle and voxel.

The logic for the residue storage shares most components from the projection accumulator. For example, the storage part is implemented the same way with the same memory size by combining multiple BRAMs with extra pipeline registers. Only its core, the actual accumulator logic, is missing, since a residue is only written once during the transfer cycles of the back projection FSM.

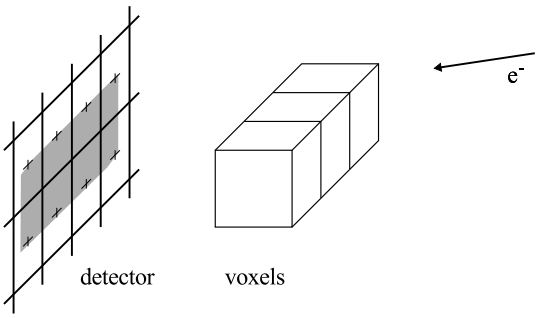


Figure 5.23: Shadow of a three voxels on the detector. Three voxels in a line can shadow up to 2×4 voxels. To simultaneously update all affected pixels, the BRAM of the projection accumulator was partitioned into 8 parts. The part is selected by calculating $(x \bmod 4, y \bmod 2)$ in detector co-ordinates with bit slicing, the remaining bits are used as the address within the parts.

If a line of residues was not accessed during the back projection of the last layer of voxels, it is marked for removal and will be substituted by a new line during the following transfer.

Together with the residues storage, the FSM ensures that all pixel values from the forward projection kernel are consumed. Otherwise, the entire design could block when the output of the forward projection kernel stalls.

Multi-piping

A first implementation of the hardware processed one voxel per clock cycle. The design usage of this design was less than one third for all types of FPGA resources. To improve the efficiency, the pipelines in the kernels were multiplied. Flynn's taxonomy [26] of the architecture is thereby converted from a one-pipeline design with *multiple instruction, single data stream* towards a multi-piped design with *multiple instruction, multiple data* streams.

The resource usage of the single-piped design allowed us to estimate that the Xilinx Virtex-6 SX475T FPGA would be able to support a triple pipelining. Many components could just be multiplied by three, such as the ray-voxel intersection, or the width of the DRAM interface. Some components stayed the same, with modified parameters, such as the FSMs in both kernels.

An increased effort was needed for the projection accumulator and the residue storage. Both act as non-trivial reductions in the data stream. Voxels that were hit by the same ray at forward projection can now hit up to three voxels during the same clock cycle, and their shadow at the ray's endpoint on the detector must be added simultaneously instead of consecutively. Also, three voxels in a line can now shadow up to 2×4 pixels instead of 2×2 pixels (Fig. 5.23).

The modification of the design therefore consisted of two parts. First, the partitioning of the memory for the projection accumulator and for the residue storage was changed. Before, it consisted of four parts, where each part was chosen whether the row and line address is odd or even. Now, the modulo by four is calculated of the line address and selects the correct memory part together with the modulo by two of the row address. This allows us to perform eight simultaneous read and write accesses to the BRAM memory with the given access pattern. On the hardware level, we compute $(x_{\mathrm{BRAM}}, y_{\mathrm{BRAM}}) = (x \bmod 4, y \bmod 2)$ by slicing the last bits and use the remaining bits as the address within the parts.

The projection accumulator expects the shadows of the voxels to be already sorted and reduced with respect to the 2×4 memory partitioning. If a ray hits more than one of the three voxels, the shadows for the detector pixel at the endpoint have to added before. Pixels with the same pair of $(x_{\mathrm{BRAM}}, y_{\mathrm{BRAM}})$ are sorted in a short sorting network and shadows are added if more than one of the three voxels is hit by the same ray on the way. The same sorting logic without the adders is also

used for the residue storage in the back projection kernel.

The multi-piping logic requires the x side of the volume to be a multiple of three. In the worst case and for a volume with 1024^3 voxels, the volume must be padded with extra voxels that increase the voxel count by about 2 ‰. The overall speed, however, is still accelerated by about a factor of three.

Chapter 6

Results

Both applications, localization microscopy and electron tomography, could be successfully accelerated while maintaining correctness and accuracy. In this chapter the results of porting the algorithms to reconfigurable hardware are presented in detail.

6.1 Localization Microscopy

For localization microscopy, the most important benchmarks are the accuracy of the localization and and the number of spots found. Both influence the resolution of the final localization image directly. A high fit accuracy is needed to improve the precision of the Abbe diffraction limit, and a high number of locations is necessary to allow the human eye to recognize the structures as seen with a conventional light microscope. While an improvement in speed was the primary motivation for this thesis, we need to ensure first that the accuracy of the results was preserved.

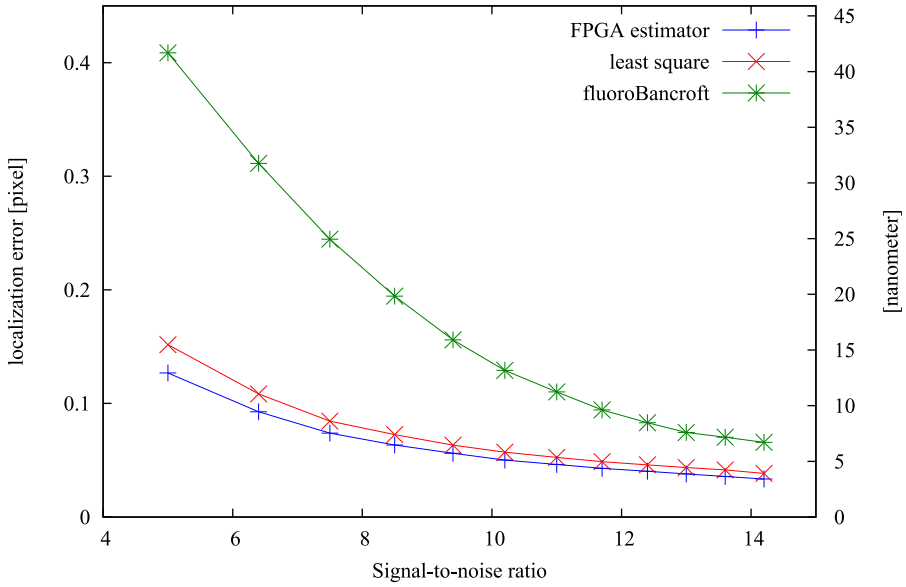
6.1.1 Accuracy

The fit errors presented here were obtained by performing a Monte-Carlo simulation. Synthetic imagery has the advantage that the true centers of the spots are known. They can be compared to the results of the fit and lead to a precise quantization of the fit error. The average localization error $\overline{\Delta\mu_{x,y}}$ is the mean error standard deviation and given by Eq. 6.1.

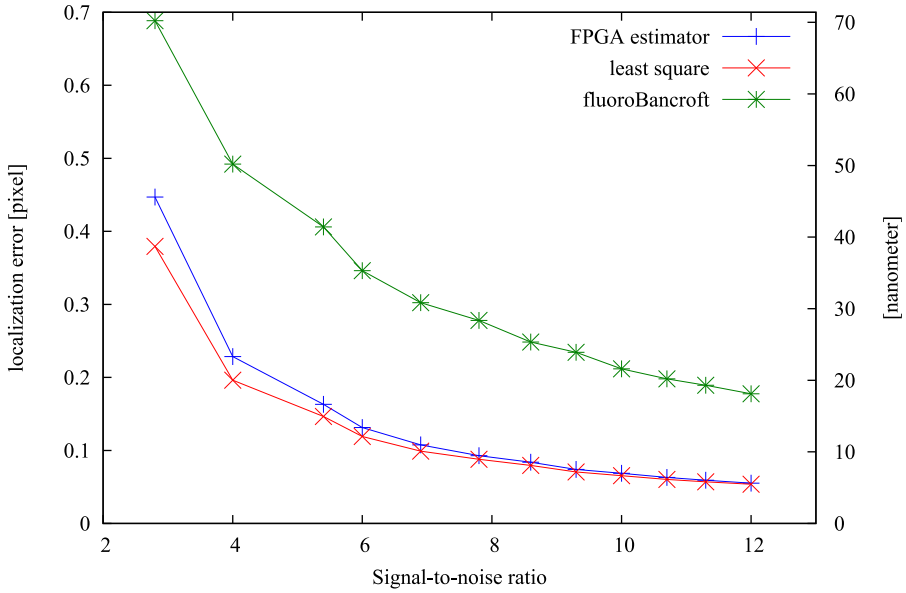
$$\overline{\Delta\mu} = \sqrt{\frac{1}{N}\sum_i (\mu_{i,\text{fit}} - \mu_{i,\text{true}})^2} \tag{6.1}$$

The result of these simulations with synthetic signals, background and Poisson noise can be seen in Fig. 6.1. The average fit error is plotted for the FPGA implementation and the previously used iterative least-square fit. For comparison the fluoroBancroft algorithm [130] was also included. It provides a non-iterative alternative to implementations that are based on the center-of-mass. For each data point 2000 spots were simulated.

The fit error is given with respect to the signal-to-noise ratio (SNR). It is defined as in Kubitscheck et al. [152] and given in Eq. 6.2. It depends on the background level B, the peak intensity $q_{\max}$ of the spot without background and the Poisson noise level at the peak $\sigma_{q_{\max}}$. The localization error is given in units of pixels and units of nanometers for a common pixel area of 102 nm × 102 nm.



(a) low background $B = 10$, $\sigma_{x,y} = 1.4$ px, 1 px is 102 nm



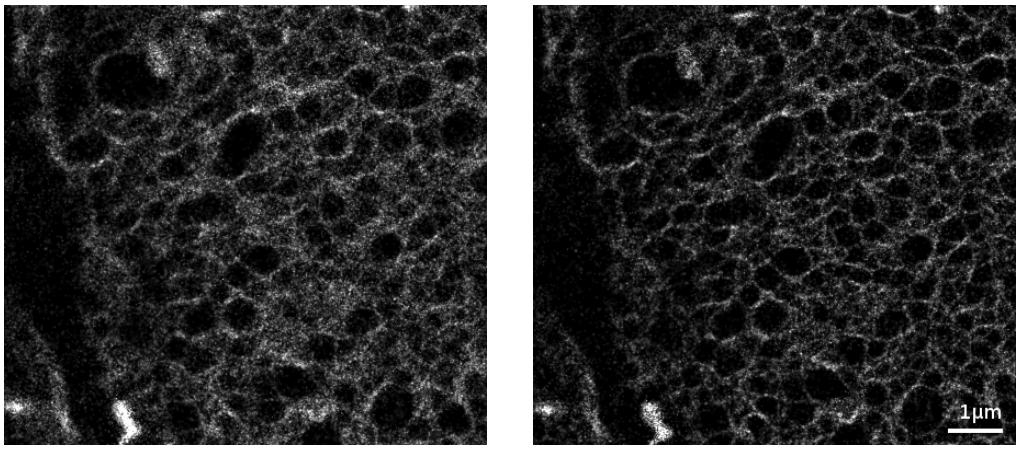
(b) high background $B = 100$, $\sigma_{x,y} = 1.4$ px, 1 px is 102 nm

Figure 6.1: Monte-Carlo simulation of the localization accuracy. The algorithm on hardware (FPGA estimator) is about as accurate as the numerical fits within a 5% margin. For a low background (B = 10) it even outperforms the numerical fit. The ROI was chosen 7×7 pixels. [10]

$$\mathrm{SNR} = \frac{q_{\max}}{\sqrt{B + \sigma^2_{q_{\max}}}} \qquad \sigma^2_{q_{\max}} = B + q_{\max} \tag{6.2}$$

$$= \frac{q_{\max}}{\sqrt{2B + q_{\max}}} \tag{6.3}$$

The charts show that the accuracy of the FPGA implementation is close to the previous least-square implementation. For a low background level with $B = 10$ photons per pixel the FPGA implementation consistently outperforms the least-square fit for all SNRs by about 5% (Fig. 6.1a).



(a) without spot separator (b) with spot separator

Figure 6.2: Improvement in resolution owed to the signal separator. For image stacks where spots regularly touch each other the spot separator sharpens the resulting localization image. The mesh structure in the tight junction is less visible without separation.

The gap narrows for higher SNRs. A background of $B = 10$ can typically be found in images where few parts of the biological sample happen to be distributed outside the focus plane of the microscope.

For a higher background $B = 100$ the least-square fit performs slightly better (Fig. 6.1b), especially for a low SNR. The FPGA implementation closes the gap in accuracy for spots with a higher intensity and both fits asymptotically reach an accuracy of 5 nm or 1/20 of a pixel.

During development the effects of zero suppression and spot separation on the least-square fit were also evaluated. These steps improved the quality of the FPGA implementation. For the least-square fit, however, the removal of values from the periphery introduced discontinuities to the signal of the underlying Airy disk and interfered with the convergence of the fit iterations.

The effect of the spot separator can be seen in Fig. 6.2. The sample shows tight junctions that form a mesh structure between cells [153]. The recording contained a large number of spots that often touched each other. With the spot separator, some areas of the mesh are more clearly visible, especially at the right border of the image. The need for spot separation was raised by the end-users in the research group of Prof. Cremer in Heidelberg who observed a clustering of signals along the lines of the mesh that would have been deemed unlikely if only synthetic imagery had been available. We found that the estimated fit error $\Delta\mu_{x,y}$ matches the true fit error well if we allow the separator to only remove less than 30% of the total intensity in a ROI and otherwise discard the signal as inseparable.

The localization microscopy challenge at the IEEE International Symposium on Biomedical Imaging 2013 [8] provided an opportunity to compete with other research groups in terms of localization accuracy. The true positions were only known to the organizers of the challenge, who also conducted the analysis. Since the algorithm needed to run on standard hardware, Manfred Kirchgessner created a graphical user interface with the Qt toolkit for the C++ version after he had finished his diploma thesis [48]. The recording was read from the provided TIFF image files and the program produced a localization image and a list of all locations found.

The challenge contained two data sets with synthetic low-density and high-density imagery. For both sets our algorithm performed accurate and consistently scored in the upper third of the board. For the high-density image stack “HD3” it achieved the best accuracy among the 15 contestants as shown in Fig. 6.3. The tendency seen in the chart indicates that the average localization error increases with the number of points found. In the FPGA implementation the average accuracy can

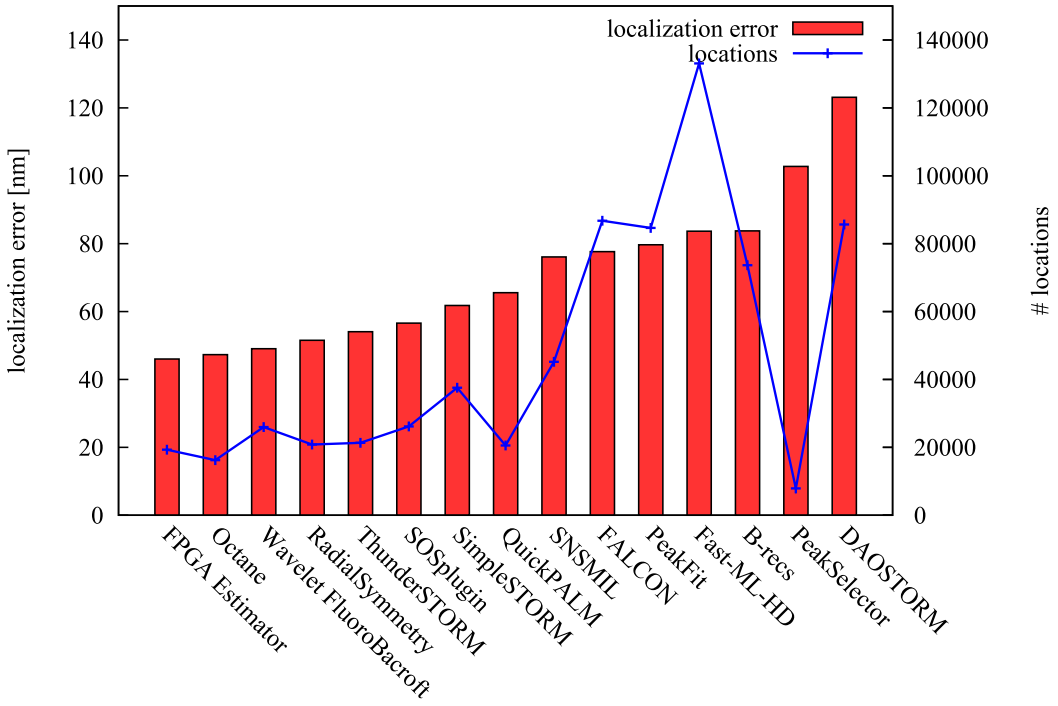


Figure 6.3: Results of the high-density contest at the ISBI Localization Microscopy Challenge [8]. The diagram shows a trend that accuracy degrades with a higher location count. For the sample HD3 our algorithm "FPGA estimator" reached the best accuracy. More locations could have been obtained by increasing the spot threshold, most probably also increasing the localization error.

be traded for a larger number of locations by increasing the threshold for the spot finder.

The accuracy of an algorithm can be also examined with real-world data if the shape of the data is known. The chart in Fig. 6.4 displays the profile of a biological structure that forms a line that is thinner than the optical resolution limit. The FPGA implementation shows a profile thinner by 12% than the original iterative least-square algorithm, indicating a smaller deviation from the line. This is largely because the FPGA implementation drops to zero faster in the chart than the least-square fit and therefore produces a higher contrast.

To rule out errors and problems in the algorithm that may not show up during evaluation, but in practice, subsequent software releases for Matlab on CPUs were created that produce the same results as the FPGA implementation. These releases were then given to other research groups. It was well-received due to the already considerable speed-up it brought to these users and provided valuable feed-back in return. As a result, the algorithm was not changed, but checks were introduced that warn the user if, for example, the brightness of the light source varied and would have interfered with the exponential smoothing in the part of the algorithm that removes the background. During evaluation, the intensity of the laser was initially assumed to be constant, but real-world usage showed that it cannot be relied on when the user operates a complex microscopy setup.

6.1.2 Throughput

After the algorithm was re-designed from a least-square fit towards a Gaussian estimator, the original Matlab program was changed accordingly. In the new implementation we chose matrix operations over nested loops wherever possible since Matlab as an otherwise interpreted language is known to take advantage of them [136]. The modification of the software alone in the same environment yielded an acceleration factor of more than 100. On a Intel i5 450 computer with

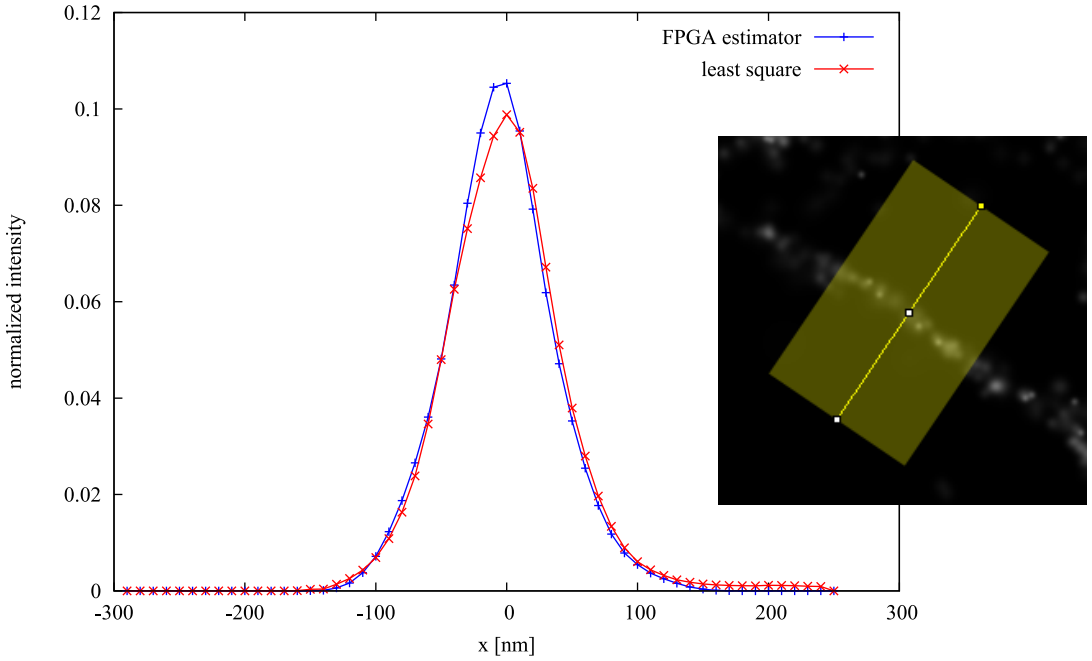


Figure 6.4: Line profile with real-world data. The line is formed by a molecular structure of a cell and is considerably thinner than the diffraction limit of light microscopy. The yellow area indicates the part of the localization image that was analyzed parallel to the yellow line. The results from the Gaussian estimator have a sharper width $\sigma_{\text{estim}} = 41.7\,\text{nm}$ than the least-square method ($\sigma_{\text{lsq}} = 47.5\,\text{nm}$). [10]

Matlab 7.10.0 the analysis of a 256×256 pixel image dropped from 7.41 s to 73.9 ms. The input frames could therefore be processed with a throughput of 0.89 Mpx/s. The time used for feature extraction consumed 68% of the computing time and scales with the number of spots found.

The algorithm was also implemented in C++ close to the new Matlab code. Matlab can be susceptible to delay code executing in its interpreter. It was found, however, that the C++ version was slower by up to a factor of two, and could not compete with the matrix operations in Matlab that are optimized for CPU vector instructions.

The dataflow implementation was carried out on a MAX2 board from Maxeler Technologies with a Xilinx Virtex-5 LX330T FPGA, where the design achieved a clock frequency of 200 MHz. The board is hosted in a workstation with a Intel Core i5 CPU 750 at 2.67 GHz and 4 GB of RAM. The design was later also synthesized on a MAX3 board for a Xilinx Virtex-6 SX475T inside of a workstation with an Intel Core i7 CPU 870 at 2.93 GHz and 16 GB of RAM.

The runtime of the application can be seen in Fig. 6.5 for both systems. Each data point was measured ten times to obtain the mean runtime and its variance. As expected, the runtime rises linearly with the number of input frames. The MAX2 system, however, runs considerable slower with 69.5 MPx/s and has a bigger runtime variance than the MAX3 system. The theoretical maximum throughput is 200 Mps/s, with one pixel consumed per clock cycle. Compared to the already accelerated Matlab implementation, the MAX2 system achieves an additional acceleration factor of 78, and the MAX3 system speeds up the analysis by 185. In total, an acceleration of 18 500 was achieved when compared to the original Matlab version.

The first kernel consumes one input pixel per clock cycle, and the feature extraction kernel requires $7 \times 7 = 49$ clock cycles to process the ROI of a spot. For a recording with 2000 frames with 280×320 each and a total of about 300 000 detected spots, the kernel for feature extraction was measured to be occupied for 7.7% of the total time the analysis was running.

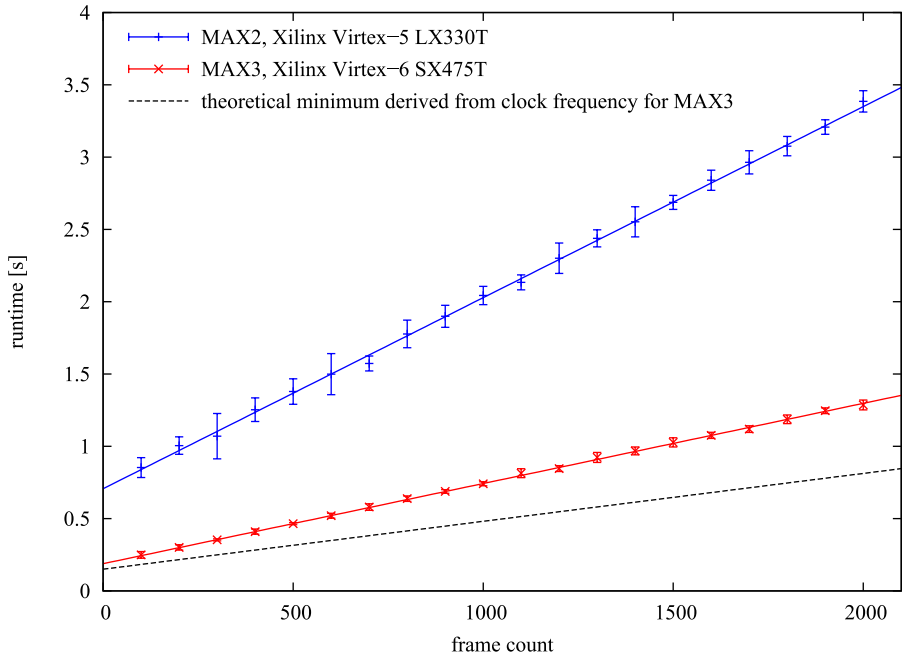


Figure 6.5: The runtime rises linearly with the number of frames with size 288 px × 320 px. The MAX2 board achieved a throughput of only 69.59 ± 0.47 Mpx/s, while the implementation on the MAX3 board runs more streamlined and reaches $167,20 \pm 0.59$ Mpx/s. The maximum theoretical throughput derived from the clock frequency is 200 Mpx/s.

6.1.3 Resource Usage

The resource usage for the FPGA implementation on the MAX2 board is shown in Table 6.1. The FPGA is occupied by about 1/3 for FFs and BRAMs, and less than 1/4 for LUTs and DSPs. The pipeline stages for background measurement have the biggest footprint on the BRAM where the background map is stored. Signal separation mainly consists of FFs to implement the comparators and the access pattern from the center towards the border in a ROI. The feature extraction step finally implements the formulas of the Gaussian estimator and consumes the majority of flip-flops for the accumulators and the following floating-point division. All other resources are used for logic to synchronize the kernels and the PCIe interface shared with the host CPU.

On the MAX3 resources usage was similar (Table 6.2). The main difference was the consumption of 56 DSPs. All other resources were consumed less and stayed within a 10% margin.

After scheduling, the pipeline of the kernel for spot detection consisted of 298 MaxJ nodes and had a latency of 1791 clock cycles. For the kernel for feature extraction a pipeline with 639 MaxJ nodes and a latency of 280 clock cycles was generated.

The source code for the application consists of 1 517 lines of MaxJ for the hardware description and 1 970 lines of C++ for the host code. For the evaluation of the new algorithm 3 886 lines of Matlab code were written. The hardware description took about three month to write down after the algorithm was understood and after the new fit method was implemented in Matlab and C++.

6.1.4 Discussion

The acceleration of the analysis for localization microscopy is a result of a combined algorithm and hardware acceleration approach. Most of the time was spent on re-writing the algorithms while carefully maintaining the same level of accuracy. When the algorithm was brought into a design suitable for pipelining the following implementation is straightforward with the MaxCompiler

component	LUTs	FFs	BRAMs	DSPs
background measurement and subtraction	1 179	1 308	51	0
spot finding	211	198	0	2
spot separation	1 401	3 303	0	0
feature extraction	23 146	33 817	0	42
total resource usage	42 044	63 881	108	44
total resources available	207 360	207 360	324	192
total resource usage ratio	20%	31%	33%	23%

Table 6.1: Resource usage on a Xilinx Virtex-5 LX330T FPGA (MAX2) for localization microscopy. The clock frequency is 200 MHz. The resources for synchronization and I/O are included in the total number [10].

component	LUTs	FFs	BRAMs	DSPs
background measurement and subtraction	1 301	1 343	51	0
spot finding	414	333	0	2
spot separation	1 257	3 369	0	0
feature extraction	26 969	38 302	0	54
total resource usage	41 336	62 897	96	56
total resources available	297 600	595 200	1064	2016
total resource usage ratio	14%	11%	9%	3%

Table 6.2: Resource usage on a Xilinx Virtex-6 SX475T FPGA (MAX3) for localization microscopy. The clock frequency is 200 MHz. With regard to the Virtex-5 LX330T 12 additional DSPs were used in total, but fewer LUTs, FFs and BRAMs.

libraries.

The results show that the Gaussian estimator with the chosen background removal, zero suppression and spot separation is as accurate as the compute-intensive least-square fit in Matlab. This is suprising, given the simplicity of the final algorithm. No iterations are needed and all features of a spot can be calculated within a single scan of the ROI. This made a pipelined implementation feasible that takes one pixel per clock cycle as an input. An implementation of the iterative least square fit would require the calculation of the fit function (Eq. 5.22) and its derivatives with respect to the fit parameters an unknown number of times until a convergence criterion is reached. The calculation of the exponential function would have led to an increased resource usage on the FPGA and a lesser throughput. [10]

The change in the algorithm accounted for an acceleration factor of more than 100, and the dataflow implementation in hardware multiplied an additional acceleration factor of 185 on top of it. For the user of the FPGA implementation this results in an acceleration of 18 500. As we will see for the application for electron tomography a direct connection of the microscope with the FPGA board is likely to reach the theoretical maximum of an acceleration factor of 22 500. Especially with the MAX2 board a big fraction of the maximum speed is lost due to I/O.

Compared to the iterative MaLiang method on GPUs the speed-up is 2.5 for the MAX3 implementation [9]. The difference is rooted in the change of the algorithm, since the FPGA resources were only occupied by less than 1/3. Otherwise, with a similar algorithm, the FPGA implementation should have shown an inferior performance in terms of raw hardware performance.

The throughput of the algorithms allows the processing of raw data in real time for current microscope setups that are usually below 10 Mpx/s, which is more than one order of magnitude

slower than the FPGA. For most applications of microscopy, the speed of the accelerated Matlab implementation is already fast enough and a relief for the operators. Since its development our method has gained traction and is now in use in the biomedical research groups of Prof. Dr. Michael Hausmann (Heidelberg University), Prof. Dr. Christoph Cremer (IMB Mainz) and Dr. Rainer Kaufmann (University of Oxford). Along with the ISBI localization microscopy challenge we can therefore be sure that all major flaws of the algorithms would have been caught and that the usability of the algorithm meets the standards of the research groups involved. A portal has been set up by the author to co-ordinate future improvements and to improve the interface with already existing tools (Fig. 6.6).

The resource usage indicates that the algorithm would also fit on smaller and cheaper FPGAs such as the Xilinx Artix-7 A105. FPGAs of this size already handle the read-out of some cameras and could be used to perform the analysis for localization microscopy within the housing of the camera in the future.

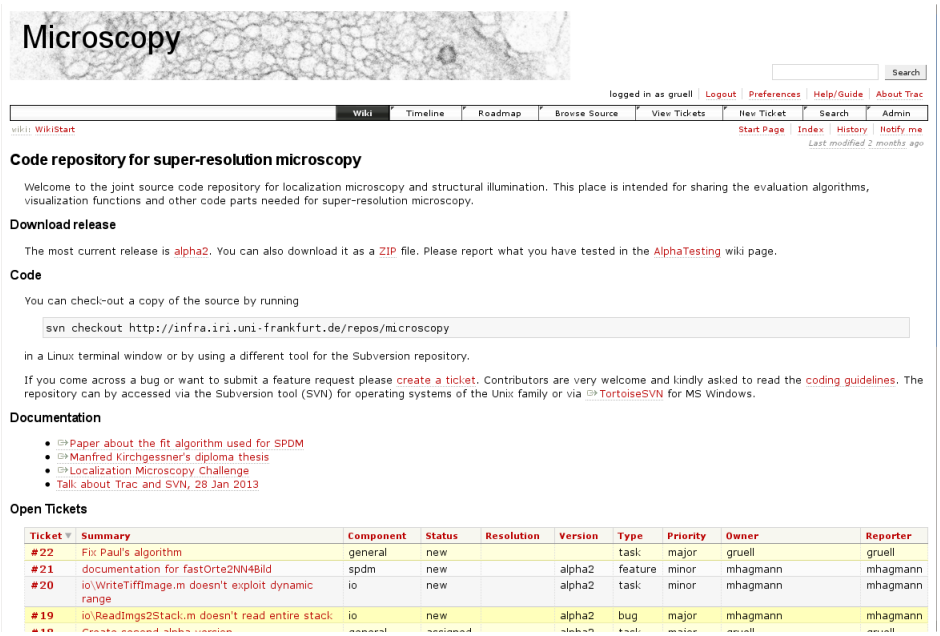


Figure 6.6: Collaboration portal for super-resolution microscopy. The development of the accelerated Matlab program let to the establishment of a bug tracker and wiki, where academic users from Heidelberg, Mainz, Oxford and Frankfurt collect their algorithms (http://infra.iri.uni-frankfurt.de/trac/microscopy).

6.2 Electron Tomography

In this section the results for the application for electron tomography are presented. After forward projection, comparison and backward projection were all implemented on the FPGA we examine the application in terms of accuracy, speed and resource usage.

6.2.1 Accuracy

The precision of the fixed-point data encodings was derived theoretically for most types from the resolution of the CCD camera pixels and the dimensions of the volume and its voxels (Table. 5.2). The accumulators for the forward projection, the comparator for the residues and the residue storage therefore compute exactly without error.

The matrix multiplications for the affine transformations between the coordinate systems of the volume and the microscope inevitably cause a loss in precision. Also, the ray-voxel intersection is susceptible to a non-continuous change of the intersection length when a ray is parallel to one of the edges of a voxel. An arbitrarily small error in the voxel position can then make a difference between a full hit and a miss.

Fig. 6.7 shows the error of the logic for ray-box intersection. The end point of the ray as well as the position of the voxel were randomly chosen 10 000 times within the possible boundaries of the application. Then, the ray direction was taken from a random distribution such that the voxel was hit in about 50% of all cases by the single ray, an upper limit to the observed hit rate in the application with four rays per voxel. The error was determined by calculating the difference between the ray-voxel intersection length in Java with double precision and the bit-accurate simulation with the MaxCompiler library.

For a voxel with an edge length equal one, the standard deviation of the error is $\sigma_{\text{ray-voxel}} = 1.6 \cdot 10^{-7}$ with a bias of $\mu_{\text{ray-voxel}} = -3.8 \cdot 10^{-10}$. All errors stayed within $\pm 1.1 \cdot 10^{-6}$. The average error $\sigma_{\text{ray-voxel}}$ is far smaller than the error introduced by the CCD camera. Here, the relative error of a 16-bit integer is at least $\sigma_{\text{CCD}} \geq 2^{-16} = 1.5 \cdot 10^{-5}$. The relative error from the CCD is therefore at least two orders of magnitude larger than the error introduced by fixed-point arithmetic. A shadow is later calculated in the forward projection by multiplying the voxel density with the intersection length, and the relative errors of the factors are added quadratically. This allows us to ignore $\sigma_{\text{ray-voxel}}$ for the following (exact) projection accumulation and comparison with the real projection. The same applies for the back projection.

6.2.2 Throughput

For both the forward and the back projection kernel a clock frequency of 170 MHz was achieved. The remaining parts of the design, such as the manager, the DRAM controllers and the PCIe controller were set to run at a slower clock frequency of 120 MHz. The different clock speeds allowed to reach timing closure for all parts of the design with a maximum speed for the kernels. The data paths of the slower parts were widened by a factor of two to compensate for the decreased clock frequency. The design was synthesized for the Max3 board with a Xilinx Virtex-6 SX475T FPGA. [11]

The maximum throughput can be derived immediately from the parallelism within the kernels. Both kernels consume three pixels per clock cycle when they are not within a wait state or transfer data between the forward projection accumulator and the residue storage. The result for two different beam declination angles α is shown in Fig. 6.8. The runtime of the application was measured as a function of the number of projections. For each projection count, the runtime was measured 10 times and the mean and the variance were plotted. A projection consisted of 1024×1024 values from the detector pixels, and the volume was partitioned in 1024^3 voxels.

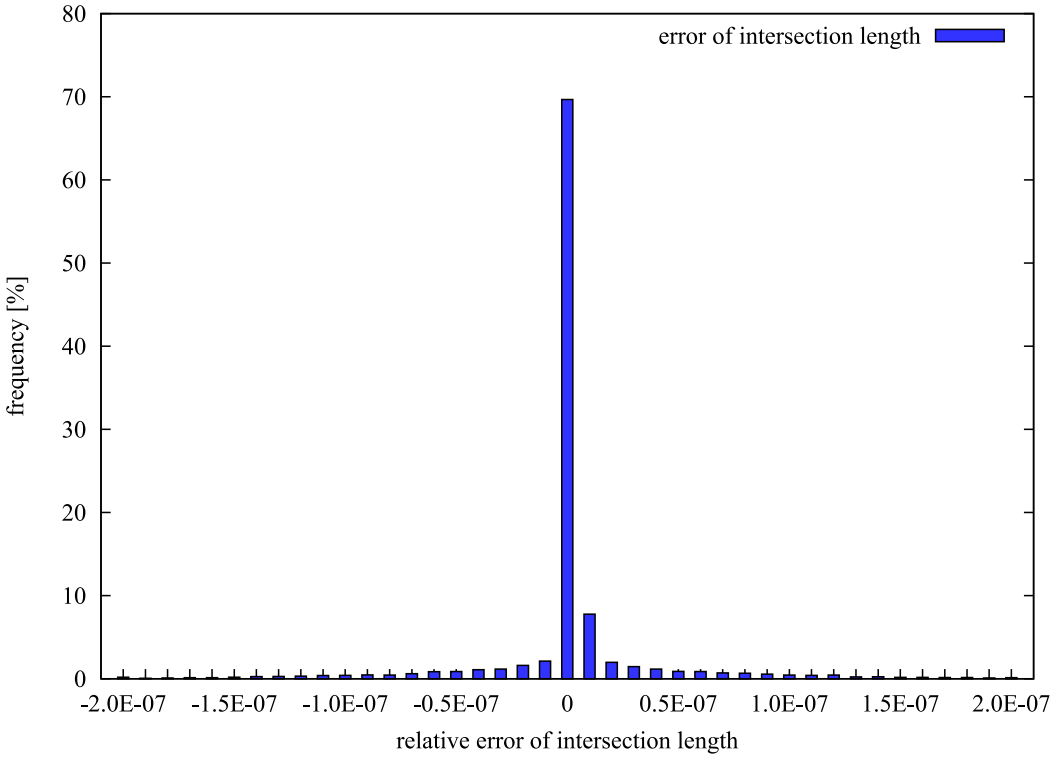


Figure 6.7: Error of the ray-box intersection length introduced by fixed-point arithmetic. The relative error of the ray-voxel intersection has a long tail . Its average is $\sigma_{\text{ray-voxel}} = 1.6 \cdot 10^{-7}$, which is two orders of magnitude smaller than the relative error introduced by the camera ($\sigma_{\text{CCD}} \geq 2^{-16} = 1.5 \cdot 10^{-5}$). The accumulators and the comparison logic for the residues calculate exactly.

The throughput is then derived from the slope of the trend line. With a clock frequency of 170 MHz the maximum throughput would be 510 megavoxel/s in theory. For a declination angle of $\alpha_1 = 1°$, a throughput of 490 megavoxels/s was achieved, which translates to an efficiency of 96% when compared to the theoretical maximum. For $\alpha_2 = 10°$, the throughput decreases to 425 megavoxels/s, or 83% of the theoretical maximum. The variance of the runtime is low, smaller than 0.5% of the mean for each projection count.

The decrease in throughput correlates with an increased number of projection lines that have to be buffered at the start of a projection and before the back projection kernel can begin to update voxel densities. Similarly, projector lines that have not been transfered yet will block the forward projection kernel at the end of a projection until they are consumed by the back projection kernel. The amount of virtual detector lines where both kernels overlap is only 4 lines for α_1, but 250 lines for α_2 as the declination angle.

The combined (constant) setup and cleanup time is 19 s in both cases. The measurement did not include the time needed at the end of a reconstruction to write the results to a hard disk. The time needed to store a projection with a fixed number of voxels would add to the constant setup and cleanup time and introduce additional jitter.

6.2.3 Resource Usage

The hardware design was implemented on the MAX3 board, which was big enough to hold all components of the algorithm with three parallel voxel pipelines. Table 6.3 list the resources which were consumed on the embedded Xilinx Virtex-6 SX475T FPGA. The row for ray-tracing describes the resources required for the affine transformations and calculation for the ray-voxel intersection.

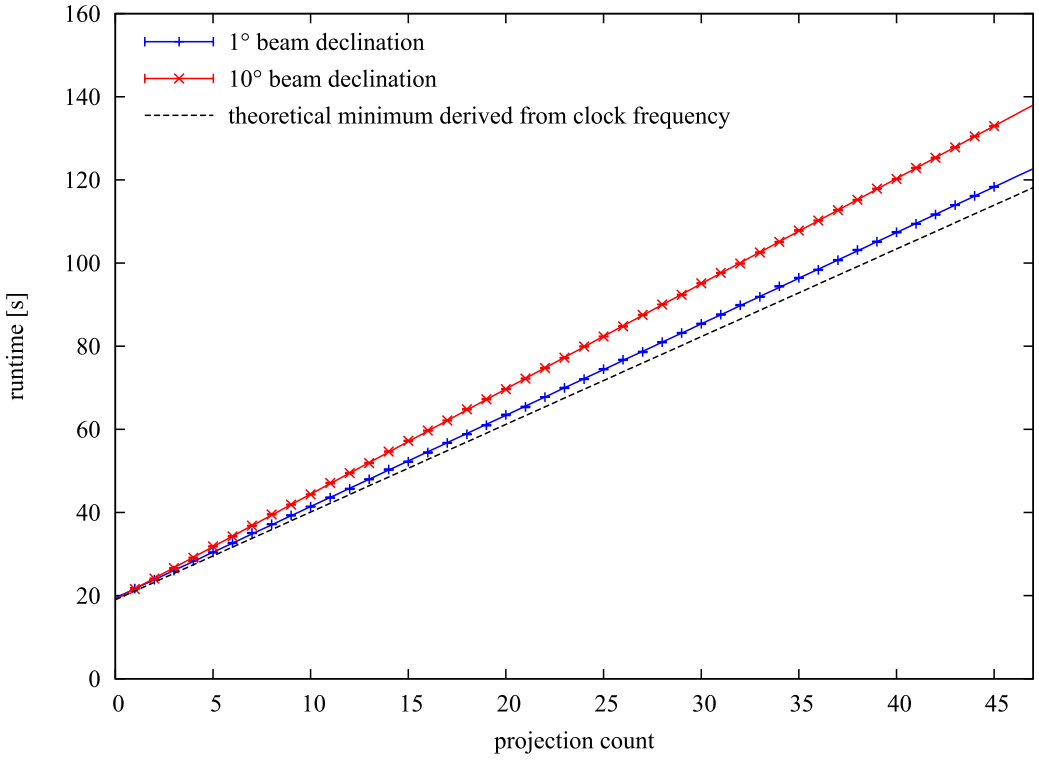


Figure 6.8: Runtime of the hardware implementation for electron tomography. The runtime is shown for the reconstruction of a volume with $1k^3$ voxels from projections with $1k^2$ pixels. The measured throughput is 490 megavoxels/s for a beam declination of $\alpha_1 = 1°$, which is close to the theoretical maximum of 510 megavoxels/s. The throughput is 425 megavoxels/s for an extreme declination angle of $\alpha_2 = 10°$. [11]

The logic is present in both the forward and back projection kernel. It consumes most of the LUTs, FFs and DSPs. The projection accumulator in the forward projection kernel and the residue storage in the back projection kernel mainly consist of BRAM, where the active part of the projection is buffered.

The kernels for forward and back projection consume about the same amount of resources within a 10% margin. The logic needed for the synthesized DRAM and PCIe controllers as well as the synchronization manager is expensive in terms of resource usage and consumes more look-up tables than both kernels combined. Apart from the DSP usage, the FPGA is occupied by about 50%. The number of available DSPs constitutes the limiting resource of the design. With more DSPs on the chip, the design could be extended to more than three voxel pipelines.

The numbers shown in Table 6.3 differ slightly from the previous publication [11] due to bug-fixes and optimizations that were introduced after submission. Most resources were saved when the control logic for forward and back projection was moved from a convoluted dataflow description towards a simpler and clearer description with state machines. An initial implementation with FSM did not reach 170 MHz for the kernels, and was only reached again after the introduction of extra pipeline stages. These stages were added at the interface between the FSMs and the pipeline parts they control.

The source code of the e-tomography design consists of 7 752 lines of MaxJ for the hardware description and 1 569 lines of C++ for the host code. From the MaxJ description the MaxCompiler library generated 364 936 lines of VHDL code. The longest path in the pipelines has a length of 97 stages in the kernel for forward projection and 109 stages for back projection. It took the author approximately eight months to implement and test the design.

component	LUTs	FFs	BRAMs	DSPs
ray tracing (per kernel)	22 220	66 939	0	720
projection accumulator	5 038	15 994	192	0
residue storage and update	3 985	13 728	192	0
forward projection kernel	28 345	85 461	192	720
back projection kernel	30 789	89 295	192	728
IO / DRAM / synchronization	72 936	90 942	181	0
total resource usage	132 070	265 698	565	1 448
total resources available	297 600	595 200	1 064	2 016
total resource usage ratio	44.4%	44.6%	53.1%	71.8%

Table 6.3: Resource usage on a Xilinx Virtex-6 SX475T FPGA (MAX3) for electron tomography. Ray-tracing consumes the DSPs, the BRAMs are used for projection and residue storage. The table contains bug fixed and optimizations introduced past the previous publication [11].

6.2.4 Discussion

The algorithm for electron tomography was successfully ported to reconfigurable hardware by re-organizing the access order of the data towards a linear pattern. The new data flow then gave us the opportunity to spend the same amount of time for each voxel for calculating the ray-voxel intersection instead of pseudo-random sampling along a ray. This lead to a pipeline design where both kernels could be statically scheduled.

The results show that we maintain accuracy where we cannot compute with otherwise exact fixed-point operations. The error introduced at the calculation of the ray-voxel intersection is two orders of magnitude bellow the error caused by the microscopy setup and therefore negligible. When the setup changes, the design can be parametrized with more integer or fractional bits for the fixed-point data types, depending on the constraints of the microscope.

The speed of the design slightly depends on the beam declination angle α. For a small angle, that is for a electron microscope with well-calibrated optics, it closely approaches the maximum throughput derived from the clock frequency of the kernels. For a larger declination angle α, the number of transfer cycles increases. These cycles are a necessity that pauses the computation of the projections, and therefore slow down the overall throughput. About 1/8 of all cycles are transfer cycles for the upper limit of $\alpha = 10°$.

The most recent GPU design by Wei Xu et al. reaches more than 1 gigavoxel/s for some problem sizes [146]. Compared to this, the FPGA design is slower. However, the GPU design relies on $\alpha = 0°$ and can therefore compute 3D reconstructions from a stack of 2D reconstructions. The required support for non-zero declination angles has complicated the design for 3D ray-box intersection and projection handling.

The GPU design that we received as a starting point from the research group of Achilleas Frangakis was developed on two NVidia Tesla C1060 working in parallel. These can trace rays from any direction and processes 196 megavoxels per second. The FPGA design therefore runs faster and accelerates reconstruction by a factor of 2.5 for a small, but common declination angle. Compared to a single graphics card, the application is accelerated by a factor of 5. Due to the entirely different architectures the reason for the acceleration must stay unclear. The FPGA benefits from the linear access pattern on the volume storage in DRAM, and the continuous operation on numbers encoded as fixed point instead of floating point.

Chapter 7

Conclusion

In the introduction of this thesis three statements were claimed to be true for the acceleration of biomedical image processing and reconstruction with FPGAs (see page 2). The results in the previous chapter now give us the opportunity to defend these for both applications that were chosen as benchmarks: localization microscopy and electron tomography. The following statements are made and will be discussed in more detail in the sections below.

1. Effective portability from an imperative program to a dataflow description.
2. Efficient development with high-level languages.
3. Acceleration of both sample applications by a significant factor.

7.1 Portability

Both applications had their core algorithms written in the style of imperative control flow. It was not initially apparent whether the algorithms could be re-written to follow a dataflow description. Before, smaller algorithms for image processing where implemented successfully on reconfigurable hardware, such as pixel functions and convolutions on raster images, but have stayed below the threshold of a fully-ported scientific high-performance application. Larger applications, such as SART for computer tomography, were only partially implemented and have become suitable for reconfigurable hardware only recently after chip sizes have sufficiently increased. In this thesis, the image processing applications for localization microscopy and electron tomography (SART) were implemented with all compute kernels on reconfigurable hardware for the first time.

The port of both applications was conducted by focusing on the smallest unit of visual data, the pixel, or for 3D electron tomography, the voxel. The analysis of this data flow then allowed the exploration of the solution space compatible with a pipeline architecture, and the selection of a solution that adheres to the constraints of the FPGA hardware and the periphery on its circuit board. For an efficient mapping a solution that consists of a system of long and statically scheduled pipelines was found. These pipelines also contained few branches that would be seldom used for meaningful result.

Other applications that process rasterized imagery can likely be ported, too. The implementation of the algorithms for localization microscopy and electron tomography show an inroad into the problem domain. The general approach was described in chapter 4, “Acceleration of Imperative Code with Dataflow Computing”. The following chapter described both example applications. Finally, the maintenance of the same functionality, including the accuracy of the computation, was shown in chapter 6, “Results”.

7.2 High-level Development

The increased size of the code base made the development of an implementation for a single developer unlikely with commonly used low-level description languages, such as VHDL and Verilog. The integration of a PCIe interface, for example, is estimated to already take half a year or more for a Ph. D. student. The time needed for implementation of the sample applications was only three months for localization microscopy and eight months for electron microscopy. The remaining time could be spent for high-level design and evaluation for accuracy, throughput and resource usage.

The chosen MaxJ compiler library from Maxeler Technologies enabled the delivery of both sample applications within three years of this Ph. D. study. Contrary to other high-level languages that aim to translate imperative control-flow descriptions to hardware, MaxJ follows the dataflow model to abstract from the low-level details of the hardware. This means that the hard work of translating control flow to data flow remains at the programmer, and MaxJ can be understood more as a set of macros that generate hardware pipelines than a programming language where the underlying hardware can be of little interest to the developer. It is, however, the opinion of the author that the approach of MaxJ makes the resource usage much more predictable from the source code when compared to control-flow abstractions.

The ratio of hand-written lines of code of code versus the amount of VHDL generated by the MaxCompiler library approaches 50 for the electron tomography application and gives a hint about the efficiency that can be gained through high-level tools. A comparison of the resource usage of hand-written VHDL and hand-written MaxJ was carried out by group member Heiko Engel [154]. It shows that the resource usage of MaxJ has a different signature in terms of look-up-tables, flip-flops and DSPs, but keeps the overall resource usage of the FPGA close to the VHDL implementation. Combining these results, MaxJ can be recommended as a tool to improve the efficiency of a hardware programmer for applications where hardware from Maxeler is an option.

7.3 Acceleration

The solution space was large enough to select on details such as the access pattern for the sample applications. The feature extraction kernel for localization microscopy allowed linear access of the signal in the region of interest, and the application for electron tomography could be re-written to also access the 3D volume linearly voxel by voxel. The pipelines could then be scheduled statically to process one data item per clock cycle in general. The following benefits allow a dataflow pipeline to run faster than a CPU program despite of slower clock frequencies and the infrastructure overhead on the chip needed for reconfiguration.

- **Register transfers:** General purpose hardware has only a limited number of registers, and a compute core performs transfers between these registers one at a time. Techniques such as hyper-threading and multi-cores cannot ease this constraint to the same extent as a statically scheduled pipeline that performs hundreds of register transfers between pipeline stages at every clock cycle. For image processing the pipeline is a natural way of processing pixel or voxel values for a wide range of algorithms, ranging from convolutions with a stencil to feature extraction and image reconstruction through back projections.

- **Control logic:** An application that can be written as a statically scheduled pipeline follows the data flow and contains few control logic. The space required for the cache hierarchy, out-of-order execution, branch prediction and virtual memory on general purpose hardware on a CPU is freed and compensates for the overhead required for reconfiguration on an FPGA. For image processing both example applications benefited from pipelines that consume one

pixel or voxel per clock cycles and transport these through the stages before the corresponding intermediate values get accumulated or otherwise reduced for the end result. The dataflow pipeline can be seen as a member of the map-reduce family of algorithms.

- **Custom data operations:** FPGAs support numeric data types of custom range and precision. General purpose hardware can only support a limited number of types, such as integers with a width that must be a power of two between 8 bit and 128 bit. The arithmetic logic in a CPU must then follow a design that produces correct results for the worst case, independent of the error semantics. On the FPGA we are free to choose any encoding between these formats, depending on the intrinsic error of the input data. Image processing is especially well-suited for custom data operations because image data is recorded as integer pixel values. The following pipeline stages can then often be implemented with custom fixed-point hardware. Fixed-point operators consume less resources than floating-point operators on an FPGA. Again, the hereby saved resources become available for computing.

- **Larger FPGAs:** Finally, the availability of larger chips pushes the development of image processing applications on FPGAs. The applications for electron tomography was only partially implemented on reconfigurable hardware before. An increase of chip resources that out-paces Moore's law allowed the transition of small applications that perform few convolutions to full-sized implementations for high-performance computing. While this enables portability in the first place, the increase in chip size makes the previous points to come into effect.

The results are execution times that are only fractions of what they were before. The analysis for localization microscopy was accelerated by a factor of 18 500, where a factor of 185 was due to hardware acceleration and the remainder due to changes in the algorithm. For electron tomography, the application was accelerated by a factor of five compared to an implementation that was heavily optimized to run on a NVidia Tesla C1060 graphics card.

It has already been shown that data flow computing on reconfigurable hardware accelerates a wide range of applications by orders of magnitude compared to execution in CPUs [155]. With this work it was shown that biomedical image processing and reconstruction can benefit from this approach in the same range.

7.4 Outlook

The acceleration of analysis for localization microscopy is expected to further advance optical light microscopy. The resolution of future CCD sensors with single-photon efficiency will increase further with time, and fluorophores that switch states at higher frequencies are in development. This will lead to bigger image frames and higher frame rates from the camera. The developed FPGA accelerator will keep up data processing speeds with the increased data rates and gives way to the next generation of localization microscopy. Images of life cells could then be resolved in real-time with sub-second latencies, a time scale where many biological processes take place and localization microscopy becomes feasible to monitor in vivo manipulation.

For electron tomography the acceleration factor of five is about the same factor that the FPGA board was more expensive than the graphics card. Further research and the observation of market prices will decide which hardware to use for future experiments. Here, better sensors with higher resolutions will allow bigger sample sizes, and the FPGA is in a unique position to keep the entire 3D density distribution that will grow with $O\left(l^3\right)$ in its on-board memory.

FPGAs have evolved from small chips previously only used for "glue logic" to ASIC replacements and accelerators of fully-featured applications. FPGAs are still less known and have

remained a tool for special domain experts. With the acceleration potential shown in this thesis for applications that would have only fit on a CPU system few years ago, and the advent of high-level dataflow description languages that increase productivity while maintaining execution speeds, the distribution of FPGA accelerators is expected to further increase in popularity in the scientific community concerned with high-performance computing.

Bibliography

[1] Donald G. Bailey. *Design for Embedded Image Processing on FPGAs*. John Wiley & Sons, 2011.

[2] Maxeler Technologies. MaxCompiler white paper, February 2011. http://www.maxeler.com/content/software/.

[3] E. Betzig, G. H. Patterson, R. Sougart, O. W. Lindwasser, S. Olenych, J. S. Bonifacino, M. W. Davidson, J. Lippincott-Schwartz, and H. F. Hess. Imaging intracellular fluorescent proteins at nanometer resolution. *Science Express*, 313:1642–1645, 2006.

[4] P. Lemmer, M. Gunkel, D. Baddeley, R. Kaufmann, A. Urich, Y. Weiland, J. Reymann, P. Müller, M. Hausmann, and C. Cremer. SPDM: Light microscopy with single-molecule resolution at the nanoscale. *Applied Physics B: Lasers and Optics*, 93:1–12, 2008.

[5] A.H. Andersen and A.C. Kak. Simultaneous algebraic reconstruction technique (SART): A superior implementation of the ART algorithm. *Ultrasonic Imaging*, 6(1):81 – 94, 1984.

[6] Miriam Leeser, Srdjan Coric, Eric Miller, Haiqian Yu, and Marc Trepanier. Parallel-beam backprojection: An FPGA implementation optimized for medical imaging. *J. VLSI Signal Process. Syst.*, 39(3):295–311, March 2005.

[7] Daniel Castaño Díez, Hannes Mueller, and Achilleas S. Frangakis. Implementation and performance evaluation of reconstruction algorithms on graphics processors. *Journal of Structural Biology*, 157(1):288 – 295, 2006.

[8] Daniel Sage, Hagai Kirshner, Thomas Pengo, Nico Stuurman, Junghong Min, and Suliana Manley. Localization microscopy challenge workshop. In *IEEE International Symposium on Biomedical Imaging 2013*, April 2013.

[9] Tingwei Quan, Pengcheng Li, Fan Long, Shaoqun Zeng, Qingming Luo, Per Niklas Hedde, Gerd Ulrich Nienhaus, and Zhen-Li Huang. Ultra-fast, high-precision image analysis for localization-based super resolution microscopy. *Optics Express*, 18(11):11867–11876, 2010.

[10] Frederik Grüll, Manfred Kirchgessner, Rainer Kaufmann, Michael Hausmann, and Udo Kebschull. Accelerating image analysis for localization microscopy with FPGAs. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 1–5, 2011.

[11] Frederik Grüll, Michael Kunz, Michael Hausmann, and Udo Kebschull. An implementation of 3D electron tomography on FPGAs. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–5, 2012.

[12] Christophe Bobda. *Introduction to Reconfigurable Computing*. Springer, 2007.

[13] *Virtex-6 FPGA Configurable Logic Block – User Guide*, February 2012. Xilinx Inc.

[14] *Spartan and Spartan-XL FPGA Families Data Sheet*, June 2008. Xilinx Inc.

[15] Ian Carlos Kuon. *Measuring and navigating the gap between FPGAs and ASICs*. PhD thesis, University of Toronto, 2008.

[16] *Virtex-6 FPGA Memory Ressources – User Guide*, September 2013. Xilinx Inc.

[17] IEEE standard VHDL language reference manual. *IEEE Std 1076-1987*, 1988.

[18] Ken Chapman. *Get your Priorities Right – Make your Design Up to 50% Smaller*, October 2007. Xilinx Inc.

[19] *Xilinx Synthesis Technology (XST) User Guide*, November 2009. Xilinx Inc.

[20] Frederik Grüll. *The Parallel Object Language – Development and Implementation of an Object Oriented Language for Partial Dynamic Reconfiguration on FPGAs*. May 2009. Diploma thesis, University of Heidelberg.

[21] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, June 2011.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter 1.5 Trends in Power and Energy in Integrated Circuits, pages 21–26. Elsevier, 2012.

[23] Jeff Rupley. AMD's "Jaguar": A next generation low power x86 core. In *Hot Chips 24*, August 2012.

[24] David A. Patterson and Carlo H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[25] Jeff Dean. Keynote: Numbers everyone should know. In *Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

[26] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, 1972.

[27] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2: Instruction Set Reference, A-Z. June 2013.

[28] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70, March 1996.

[29] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[30] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[31] John von Neumann. First draft of a report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993.

[32] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[33] Ian Watson and John Gurd. A prototype data flow computer with token labelling. In *Managing Requirements Knowledge, International Workshop on*, page 623. IEEE Computer Society, 1899.

[34] Mihai Budiu. *Spatial Computation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2003.

[35] Bernd Jähne. *Digital Image Processing*, chapter 4 Neighborhood Operations, pages 105–134. Springer, 2014.

[36] Rafael C. Gonzales and Richard E. Woods. *Digital Image Processing*, chapter 3 Intensity Transformations and Spatial Filtering, pages 104–198. Pearson Education, 2008.

[37] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 4:114–117, April 1969.

[38] International Technology Roadmap for Semiconductors. 2012 update, 2012.

[39] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[40] Top FPGA companies for 2013. *SourceTech411*, April 2013.

[41] Inc. Xilinx. 7 series FPGAs overview, February 2014.

[42] *LogiCORE IP Floating-Point Operator v5.0*, March 2011. Xilinx Inc.

[43] Maxeler Technolgies Ltd., London. website: http://www.maxeler.com.

[44] Convey Computer. Convey white paper, hybrid-core: The "big data" computing architecture, June 2012. http://www.conveycomputer.com.

[45] Convey Computer. Convey computer's implementation of pacbiotoca algorithm speeds dna sequence assembly, delivering up to fifteen times acceleration, December 2013. Press release.

[46] Convey Computer. New convey mx demonstrates leading power/performance on graph 500 benchmark, November 2013. Press release.

[47] Silicon Software GmbH. Silicon Software runtime 5 documentation. website: http://www.siliconsoftware.de.

[48] Manfred Kirchgessner. *FPGA-based hardware acceleration of localization microscopy*. October 2011. Diploma thesis, University of Heidelberg.

[49] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[50] Khronos Group. OpenCL: The open standard for parallel programming of heterogeneous systems. website: http://www.khronos.org/opencl//.

[51] Nvidia Corp. Nvidia Tesla GPU accelerators datasheet. website: http://www.nvidia.com.

[52] Nvidia Corp. CUDA toolkit v5.5 documentation. website: http://docs.nvidia.com/cuda/.

[53] Hans Meuer, Erich Strohmaier, and Horst Simon. Top500 supercomputer sites november 2013. website: http://www.top500.org.

[54] AMD Inc. Fusion whitepaper: AMD fusion family of APUs - enabling a superior, immersive PC experience, May 2010. http://amd.com.

[55] Intel Corp. White paper: Intel Xeon Phi coprocessor, developer's quick start guide, May 2010. http://amd.com.

[56] Jirí Dokulil, Enes Bajrovic, Siegfried Benkner, Sabri Pllana, Martin Sandrieser, and Beverly Bachmayer. Efficient hybrid execution of C++ applications using Intel(R) Xeon Phi(TM) coprocessor. *CoRR*, abs/1211.5530, 2012.

[57] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving Intel Xeon Phi. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 137–148, New York, NY, USA, 2014. ACM.

[58] Daniel D. Gajski and Robert H. Kuhn. Guest editors' introduction: New vlsi tools. *Computer*, 16(12):11–14, Dec 1983.

[59] TIOBE Software BV. TIOBE programming community index, April 2014. http://www.tiobe.com.

[60] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Commun. ACM*, 56(4):56–63, April 2013.

[61] *Handel-C language reference manual*, 2007. Agility Design Solutions Inc., http://www.agilityds.com.

[62] Inc. Xilinx. Vivado design suite user guide: High-level synthesis, April 2014.

[63] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[64] *ROCCC 2.0 User's Manual - Revision 0.6*, February 2011.

[65] Betul Buyukkurt, John Cortes, Jason Villarreal, and Walid A. Najjar. Impact of high-level transformations within the roccc framework. *ACM Trans. Archit. Code Optim.*, 7(4):17:1–17:36, December 2010.

[66] The OpenSPL Consortium. Openspl: Revealing the power of spatial computing, December 2013. http://www.openspl.org.

[67] Shan Shan Huang, Amir Hormati, David Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP 2008*, Paphos, Cyprus, July 2008.

[68] Eclipse foundation. The Eclipse project. https://www.eclipse.org.

[69] *Silicon Software VisualApplets Documentation*, February 2010. Silicon Software.

[70] Donald E. Knuth. *The Art of Computer Programming*, volume 1, Fundamental Algorithms, chapter 2.2 Linear lists, pages 238–307. Addison-Wesley, 2005.

[71] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, April 1960.

[72] Makoto Amamiya, Ryuzo Hasegawa, Osamu Nakamura, and Hirohide Mikami. A list-processing-oriented data flow machine architecture. In *Proceedings of the June 7-10, 1982, national computer conference*, AFIPS '82, pages 143–151, New York, NY, USA, 1982. ACM.

[73] Makoto Amamiya, Ryuzo Hasegawa, and Hirohide Mikami. List processing with a data flow machine. In Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa, editors, *RIMS Symposium on Software Science and Engineering*, volume 147 of *Lecture Notes in Computer Science*, pages 165–190. Springer, 1982.

[74] Haskell Data.List library, version 4.6.0.1. http://www.haskell.org/ghc/docs/7.6.2/html/libraries/base-4.6.0.1/Data-List.html.

[75] American National Standards Institute, editor. *Programming languages – C++*, chapter 25. Algorithms library, pages 543–570. ISO/IEC 14882, 2003.

[76] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, chapter Survey of sparse matrix storage formats, pages 57–60. Society for Industrial and Applied Mathematics, 1987.

[77] Donald E. Knuth. *The Art of Computer Programming*, volume 4, fascicle 3, chapter 7.2.1.2 Generating all permutations, Algorithm L, pages 5–6. Addison-Wesley, 2005.

[78] J. T. Butler and T. Sasao. Hardware index to permutation converter. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 431–436, 2012.

[79] Michael Sipser. *Introduction to the Theory of Computation*, chapter 1. Regular languages. PWS Publishing, 1996.

[80] Donald E. Knuth. *The Art of Computer Programming*, volume 3, Sorting and Searching, chapter 5.3 Optimum sorting, pages 180–247. Addison-Wesley, 2005.

[81] Dirk Koch and Jim Torresen. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[82] Anjit Sekhar Chaudhuri, Peter Y. K. Cheung, and Wayne Luk. A reconfigurable data-localised array for morphological algorithms. In *Field-Programmable Logic and Applications*, volume 1304 of *Lecture Notes in Computer Science*, pages 344–353. Springer Berlin Heidelberg, 1997.

[83] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[84] VisualVM, all-in-one Java troubleshooting tool. http://visualvm.java.net.

[85] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3B: System programming guide, part 2, chapter 18: Performance monitoring, pages 127–215. June 2013.

[86] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[87] John C. Mitchell. *Concepts in Programming Languages*, chapter Part 2, Procedures, types, temory tanagement, and control, pages 93–228. Cambridge University Press, 2003.

[88] The Go programming language. http://www.golang.org/ref.

[89] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[90] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, May 1966.

[91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[92] Marleen Adé, Rudy Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 64–69, New York, NY, USA, 1997. ACM.

[93] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[94] *Intel C++ Compiler XE 13.1 User and Reference Guide*, 2013.

[95] Song Sun and Joseph Zambreno. A floating-point accumulator for FPGA-based high performance computing applications. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, December 2009.

[96] Maxeler Technologies. Acceleration tutorial, loops and pipelining. Version 2012.2.

[97] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[98] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, FPGA '03, pages 185–194, New York, NY, USA, 2003. ACM.

[99] Oracle. *Java Platform, Standard Edition 6, API Specification*. java.util.Random.

[100] Donald E. Knuth. *The Art of Computer Programming*, volume 2, Seminumerical Algorithms, chapter 3.2.1 The linear congruential method, pages 9–10. Addison-Wesley, 1969.

[101] Donald E. Knuth. *The Art of Computer Programming*, volume 2, Seminumerical Algorithms, chapter 4.3.3 How fast can we multiply?, pages 258–280. Addison-Wesley, 1969.

[102] The Institute of Electrical and Inc. Electronics Engineers. IEEE standard for floating-point arithmetic 754-2008, August 2008.

[103] N.G. Kingsbury and P.J.W. Rayner. Digital filtering using logarithmic arithmetic. *Electronics Letters*, 7:56–58, January 1971.

[104] Norbert Pucker Cristian B. Lang. *Mathematische Methoden in der Physik*, chapter 21.2 Wahrscheinlichkeitsrechnung und Statistik, page 629. Elsevier GmbH, 2005.

[105] Robert A. Gowen and Alan Smith. Square root data compression. *Review of Scientific Instruments*, 74(8):3853–3861, 2003.

[106] A. Savakis and M. Piorun. Benchmarking and hardware implementation of JPEG-LS. In *Image Processing. Proceedings. 2002 International Conference on*, volume 2, pages II 949–952 vol.2, 2002.

[107] I. Lehtoranta, E. Salminen, A. Kulmala, M. Hannikainen, and T.D. Hamalainen. A parallel MPEG-4 encoder for FPGA based multiprocessor SoC. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 380–385, 2005.

[108] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. *SIGPLAN Not.*, 35(5):108–120, May 2000.

[109] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual*, volume 2: System Programming, chapter 2: x86 and AMD64 Architecture Differences. September 2012.

[110] JEDEC solid state technology association. *DDR3 SDRAM Standard*, chapter 3.4.2.1: Functional Description – Burst Length, Type and Order. July 2012.

[111] JEDEC solid state technology association. *DDR3 SDRAM Standard*, chapter 12.3: Electrical Characteristicx & AC Timing for DDR3-800 to DDR3-2133 – Standard Speed Bins. July 2012.

[112] R. Kaufmann, P. Müller, G. Hildenbrand, M. Hausmann, and C. Cremer. Analysis of her2/neu membrane protein clusters in different types of breast cancer cells using localization microscopy. *Journal of Microscopy*, 242(1):46–54, 2011.

[113] Ernst Abbe. The relation of aperture and power in the microscope. *Journal of the Royal Microscopical Society*, 2(3):300–309, 1882.

[114] Louis Broglie. The reinterpretation of wave mechanics. *Foundations of Physics*, 1(1):5–15, 1970.

[115] Stefan W. Hell and Jan Wichmann. Breaking the diffraction resolution limit by stimulated emission: stimulated-emission-depletion fluorescence microscopy. *Opt. Lett.*, 19(11):780–782, Jun 1994.

[116] M. J. Rust, M. Bates, and X. Zhuang. Sub-diffraction-limit imaging by stochastic optical reconstruction microscopy (STORM). *Nature Methods*, 3:793–796, 2006.

[117] S. T. Hess, T. P. K. Girirajan, and M. D. Mason. Ultra-high resolution imaging by fluorescence photoactivation localization microscopy. *Biophysical Journal*, 91:4258–4272, 2006.

[118] George Biddell Airy. On the diffraction of an object-glass with circular aperture. *Transactions of the Cambridge Philosophical Society*, 5:283, 1835.

[119] Norbert Pucker Cristian B. Lang. *Mathematische Methoden in der Physik*, chapter 17.2 Die Besselsche Differenzialgleichung, page 496. Elsevier GmbH, 2005.

[120] Andor Technology. *Sensitivity of CCD cameras, Key factors to consider*. White paper.

[121] Seamus J. Holden, Stephan Uphoff, and Achillefs N. Kapanidis. DAOSTORM: an algorithm for high-density super-resolution microscopy. *Nature Methods*, 8(4):279, 2011.

[122] Susan Cox, Edward Rosten, James Monypenny, Tijana Jovanovic-Talisman, Dylan T Burnette, Jennifer Lippincott-Schwartz, Gareth E Jones, and Rainer Heintzmann. Bayesian localization microscopy reveals nanoscale podosome dynamics. *Nature Methods*, 9(2):195–200, 2011.

[123] Sigmund Brandt. *Data Analysis*, chapter 9. The Method of Least Aquares. Springer, 1998.

[124] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes – The Art of Scientific Computing*, chapter 15.5 Nonlinear Models, pages 799–806. Cambridge University Press, 2007.

[125] The MathWorks Inc. MATLAB & Simulink R2013a documentation: Solve nonlinear curve-fitting (data-fitting) problems in least-squares sense. http://www.mathworks.de/de/help/optim/ug/lsqcurvefit.html.

[126] Sigmund Brandt. *Data Analysis*, chapter 7. The Method of Maximum Likelihood. Springer, 1998.

[127] Y. Feng, J. Goree, and Bin Liu. Accurate particle position measurement from images. *Review of Scientific Instruments*, 78(5):053704–053704–10, 2007.

[128] National Institutes of Health. ImageJ, image processing and analysis in Java. http://rsb.info.nih.gov.

[129] Russell E. Thompson, Daniel R. Larson, and Watt W. Webb. Precise nanometer localization analysis for individual fluorescent probes. *Biophysical Journal*, 82(5):2775 – 2783, 2002.

[130] S.B. Andersson. Precise localization of fluorescent probes without numerical fitting. In *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on*, pages 252–255, april 2007.

[131] S. Bancroft. An algebraic solution of the GPS equations. *Aerospace and Electronic Systems, IEEE Transactions on*, AES-21(1):56–59, 1985.

[132] Zhaolong Shen and S. B. Andersson. Bias and precision of the fluoroBancroft algorithm for single particle localization in fluorescence microscopy. *Signal Processing, IEEE Transactions on*, 59(8):4041–4046, 2011.

[133] Michael K. Cheezum, William F. Walker, and William H. Guilford. Quantitative comparison of algorithms for tracking single fluorescent particles. *Biophysical Journal*, 81(4):2378 – 2388, 2001.

[134] Dipimage & diplib. http://www.diplib.org.

[135] Sigmund Brandt. *Data Analysis*, chapter 4. Computer Generated Random Numbers, The Monte Carlo Method. Springer, 1998.

[136] The MathWorks Inc. MATLAB & Simulink R2013a documentation: Techniques for improving performance. http://www.mathworks.de/de/help/matlab/matlab_prog/techniques-for-improving-performance.html.

[137] Qt online reference documentation. http://doc.qt.nokia.com/.

[138] Richard Gordon, Robert Bender, and Gabor T. Herman. Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography. *Journal of Theoretical Biology*, 29(3):471 – 481, 1970.

[139] Stefan Kaczmarz. Angenäherte Auflösung von Systemen linearer Gleichungen. *Bulletin International de l'Académie Polonaise des Sciences et des Lettres*, 35:355–357, 1937.

[140] Saarbrücken Fraunhofer Institut für zerstörungsfreies Prüfen.

[141] Ohad Medalia, Igor Weber, Achilleas S. Frangakis, Daniela Nicastro, Günther Gerisch, and Wolfgang Baumeister. Macromolecular architecture in eukaryotic cells visualized by cryoelectron tomography. *Science*, 298(5596):1209–1213, 2002.

[142] R. N. Bracewell. Numerical transforms. *Science*, 248(4956):697–704, 1990.

[143] Gabor T. Herman. *Fundamentals of Computerized Tomography – Image Reconstruction from Projections*, chapter 8 Filtered Backprojection for Parallel Beams, pages 135–157. Springer, 2009.

[144] Joachim Frank, Michael Radermacher, Pawel Penczek, Jun Zhu, Yanhong Li, Mahieddine Ladjadj, and Ardean Leith. SPIDER and WEB: Processing and visualization of images in 3d electron microscopy and related fields. *Journal of Structural Biology*, 116(1):190 – 199, 1996.

[145] C. O. S. Sorzano, R. Marabini, J. Velázquez-Muriel, J. R. Bilbao-Castro, S. H. W. Scheres, J. M. Carazo, and A. Pascual-Montano. XMIPP: a new generation of an open-source image processing package for electron microscopy. *Journal of Structural Biology*, 148(2):194 – 204, 2004.

[146] Wei Xu, Fang Xu, Mel Jones, Bettina Keszthelyi, John Sedat, David Agard, and Klaus Mueller. High-performance iterative electron tomography reconstruction with long-object compensation using graphics processing units (GPUs). *Journal of Structural Biology*, 171(2):142 – 153, 2010.

[147] Brian Smits. Efficiency issues for ray tracing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[148] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*, chapter 12.3.1 Bounding Boxes, pages 279–285. A K Peters, 2009.

[149] *Virtex-6 FPGA Families Product Table*, January 2012. Xilinx Inc.

[150] Jung Ho Ahn, Mattan Erez, and William J. Dally. Scatter-add in data parallel architectures. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 132–142, 2005.

[151] Asadollah Shahbahrami, Jae Young Hur, Ben Juurlink, and Stephan Wong. FPGA implementation of parallel histogram computation. In *2nd HiPEAC Workshop on Reconfigurable Computing, Göteborg, Sweden*, pages 63–72, 2008.

[152] Ulrich Kubitscheck, Oliver Kückmann, Thorsten Kues, and Reiner Peters. Imaging and tracking of single GFP molecules in solution. *Biophysical Journal*, 78(4):2170–2179, 2000.

[153] Rainer Kaufmann, Jörg Piontek, Frederik Grüll, Manfred Kirchgessner, Jan Rossa, Hartwig Wolburg, Ingolf E. Blasig, and Christoph Cremer. Visualization and quantitative analysis of reconstituted tight junctions using localization microscopy. *PLoS ONE*, 7(2):e31128, 02 2012.

[154] Frederik Grüll, Heiko Engel, and Udo Kebschull. Signal processing in physics with high-level dataflow sysnthesis on FPGAs. In *Computing Frontiers, 2013 ACM International Conference on*, 2013.

[155] Oliver Pell and Vitali Averbukh. Maximum performance computing with dataflow engines. *Computing in Science Engineering*, 14(4):98–103, 2012.

Index