Energy- and Cost-Efficient Lattice-QCD Computations Using Graphics Processing Units

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Matthias Bach

aus Darmstadt

Frankfurt 2014

(D 30)

vom Fachbereich 12 der

Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Thorsten Theobald
Gutachter:	Prof. Dr. Volker Lindenstruth Prof. Dr. Owe Philipsen Prof. Dr. Tilo Wettig
Datum der Disputation:	18. Februar 2015

Abstract

Quarks and gluons are the building blocks of all hadronic matter, like protons and neutrons. Their interaction is described by Quantum Chromodynamics (QCD), a theory under test by large scale experiments like the Large Hadron Collider (LHC) at CERN and in the future at the Facility for Antiproton and Ion Research (FAIR) at GSI. However, perturbative methods can only be applied to QCD for high energies. Studies from first principles are possible via a discretization onto an Euclidean space-time grid. This discretization of QCD is called Lattice QCD (LQCD) and is the only *ab-initio* option outside of the high-energy regime.

LQCD is extremely compute and memory intensive. In particular, it is by definition always bandwidth limited. Thus—despite the complexity of LQCD applications—it led to the development of several specialized compute platforms and influenced the development of others. However, in recent years General-Purpose computation on Graphics Processing Units (GPGPU) came up as a new means for parallel computing. Contrary to machines traditionally used for LQCD, graphics processing units (GPUs) are a mass-market product. This promises advantages in both the pace at which higher-performing hardware becomes available and its price.

CL^{2}QCD is an OpenCL based implementation of LQCD using Wilson fermions that was developed within this thesis. It operates on GPUs by all major vendors as well as on central processing units (CPUs). On the AMD Radeon HD 7970 it provides the fastest double-precision $\not{D}$ kernel for a single GPU, achieving 120 GFLOPS. $\not{D}$—the most compute intensive kernel in LQCD simulations—is commonly used to compare LQCD platforms.

This performance is enabled by an in-depth analysis of optimization techniques for bandwidth-limited codes on GPUs. Further, analysis of the communication between GPU and CPU, as well as between multiple GPUs, enables high-performance Krylov space solvers and linear scaling to multiple GPUs within a single system.

LQCD calculations require a sampling of the phase space. The hybrid Monte Carlo (HMC) algorithm performs this. For this task, a single AMD Radeon HD 7970 GPU provides four times the performance of two AMD Opteron 6220 running an optimized reference code. The same advantage is achieved in terms of energy-efficiency.

In terms of normalized total cost of acquisition (TCA), GPU-based clusters match conventional large-scale LQCD systems. Contrary to those, however, they can be scaled up from a single node. Examples of large GPU-based systems are LOEWE-CSC and SANAM. On both, CL^{2}QCD has already been used in production for LQCD studies.

Contents

Chapter 1.

Introduction

Quarks and gluons—which make up hadronic matter like protons and neutrons—interact via the strong force. This force is described by Quantum Chromodynamics (QCD). QCD features asymptotic freedom. This means, the coupling decreases with increasing energy while it is of the order of unity for small energies. Thus, perturbative methods are only applicable for high energies [1, 2]. Studies from first principles are possible via a discretization onto an Euclidean space-time grid. This discretization of QCD is called Lattice QCD (LQCD). It constitutes one of the most compute intensive problems and has heavily influenced the development of computer architectures.

To retrieve physical values from LQCD calculations, the discretization must be removed again. Therefore, the lattice spacing must be decreased to infinitesimal values. However, a lattice of 30^4 points is merely enough for a hypercube of 3 fm spatial extent. Calculations on such a lattice already require a large amount of TFLOPS [3]. Additionally, due to the four-dimensional volume, increasing the resolution by a factor of two already increases the number of lattice sites by a factor of 16. Other parameters—like the pion mass—have an influence on the computational cost, too. Thus, LQCD calculations are usually performed using a pion mass differing from the physical value. To calculate quantities like the hadron masses and decay constants at an accuracy matching that of experimental results, LQCD requires machines with compute capabilities at the multi-PFLOP level [4].

To fulfil the compute requirements of LQCD, dedicated architectures like Array Processing Element (APE) [5], QCD-on-a-chip (QCDOC) [6], and QCD Parallel Computing on the Cell Broadband Engine (QPACE) [7] were developed. In addition, LQCD influenced the development of high-performance systems like the Blue Gene family [8].

The clock speed of central processing units (CPUs) no longer increases. Instead, their core count does. This makes graphics processing units (GPUs)—with their high peak performance and bandwidth—an interesting platform for high-performance computing (HPC), as parallelization is required in any case. As Figure 1.1 shows, in terms of peak performance they have been ahead of CPUs by an order of magnitude over the last years. The TOP500 [9] list of supercomputers was repeatedly led by systems using heterogeneous architectures. In November 2010 and November 2012 these were systems equipped with GPUs. In June 2013 it was a system using Intel Xeon Phis. An example of such a heterogeneous architecture for general-purpose computing is LOEWE-CSC [10]. Located at the Johann Wolfgang Goethe-Universität Frankfurt am Main it provides two 12-core Advanced Micro Devices (AMD) Magny-Cours CPUs and

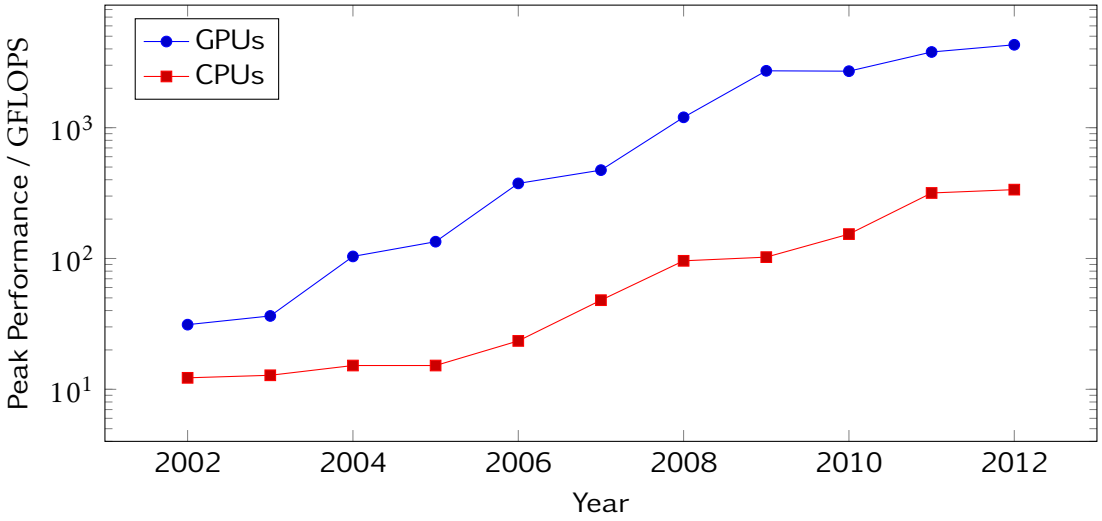

Figure 1.1.: Development of CPU and GPU compute power in comparison [14]. The CPU performance is that of the fastest consumer CPU at that time. The GPU performance is that of the fastest consumer GPU by AMD at that time.

one AMD Radeon HD 5870 GPU in the majority of its nodes. Originally, it was ranked 22nd in the TOP500 list of supercomputers and ranked eighth in the Green500 [11] list of energy-efficient supercomputers. Its energy efficiency is 718 MFLOPS/W [12]. Another example is SANAM. This computer, which was built by a cooperation of the Frankfurt Institute for Advanced Studies (FIAS) and the King Abdulaziz City for Science and Technology (KACST), is equipped with four AMD FirePro S10000 in each node. In fall 2012 it claimed second place in the Green500 [13].

Originating in the high-end computer gaming market, GPUs nowadays offer highest computing capabilities at a very attractive price-per-flop ratio. Current high-end gaming GPUs by AMD and NVIDIA are priced at about 500 €. As GPUs provide a well-suited computing architecture, there is an on-going software and algorithm development in order to utilize GPUs for this kind of computation [2, 15–17]. The pioneer work in the field [18]—using application programming interfaces (APIs) designed for graphics rendering—was in principle platform agnostic. Yet, nearly all later developments in the field were based on NVIDIA CUDA. Therefore, they are limited to hardware by this single GPU vendor.

GPU-accelerated systems offer an interesting solution to the compute requirements of LQCD. They provide a better performance-per-price ratio than CPU systems. Yet, like those—and unlike a Blue Gene system—they can be scaled up from small budgets. The same is true in relation to dedicated systems where initial development and custom hardware produced in small batches drive up the price.

To utilize clusters like LOEWE-CSC—and to get out of the vendor lock—an application which can utilize GPUs but does not rely on NVIDIA CUDA is required. In this thesis I present CL^{2}QCD, an OpenCL based LQCD application. Utilizing OpenCL en-

ables the code to be run on GPUs by AMD and NVIDIA, as well as on classical CPUs. Beyond that, OpenCL is also available on other platforms like the Intel Xeon Phi.

Parts of this work have already been presented in 'LatticeQCD using OpenCL' [19] and 'Lattice QCD based on OpenCL' [20]. Here I also present how the application was optimized to achieve the results in those publications. In addition, I show how the architecture of the application was adjusted to enable the utilization of multiple GPUs.

As mentioned, in terms of peak-performance GPUs offer an excellent price-per-flop ratio. However, they are not designed explicitly for LQCD computations. Dedicated systems and the Blue Gene come with a higher price-tag. Yet, these have been optimized—at least in part—for LQCD computations. Thus, it is not necessarily clear whether dedicated hardware or GPUs provide the best performance-per-price. Therefore, I also present a cost-per-flop comparison of CL^{2}QCD on GPUs in comparison to LQCD applications running on other systems.

Energy-efficiency is another major challenge in HPC. So far, energy consumption has risen with overall system performance. However, as energy-costs rise and only a limited amount of energy can be dissipated, performance-per-watt becomes an increasingly important metric. GPUs have repeatedly placed among the first ten systems of the Green500 list, showing the excellent energy-efficiency of GPUs in the HPL [21] benchmark. However, so far no comparisons have been performed on the energy-efficiency of LQCD computations. Here, I present the energy-efficiency of CL^{2}QCD utilizing GPUs and that of a reference CPU application.

1.1. Lattice QCD

QCD is an SU(N_c) gauge theory consisting of gauge and fermion fields. In nature, the value of N_c equals 3. The strong force, which it describes, differs from the other forces—electromagnetic, weak and gravitation—in important aspects. For long distances the potential between colour-charged particles—quarks and gluons—increases linearly. Therefore, a colour-charged particle cannot be given enough energy to overcome the potential wall. As a result they cannot be observed freely but are always bound into colourless states. This is called *confinement*.

Another distinct feature of QCD is given by its coupling constant, which for low energies is in the order of unity. Therefore, higher-order terms are not suppressed and perturbative methods cannot be used for calculations at the scale of the hadronic world [1]. Only when increasing the involved energies to the order of the Z-Boson mass, the *asymptotic freedom* stemming from the running of the coupling constant enables perturbative methods.

LQCD provides an *ab-initio* approach to QCD by discretization of the Euclidean path integral. In the continuum the partition function is the following:

$$Z = \int \mathcal{D}A_\mu \mathcal{D}\psi \mathcal{D}\overline{\psi}\, e^{-S_{\text{QCD}}}. \tag{1.1}$$

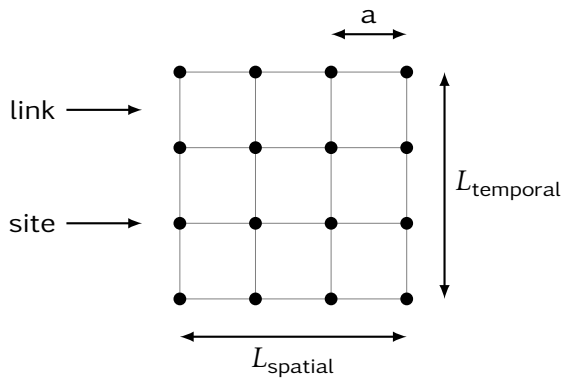

Figure 1.2.: Lattice with sites, links, lattice spacing and dimensions

In this integral S_{QCD} is the QCD action:

$$S_{\text{QCD}} = \int d^4x \left(\frac{1}{4} F_{\mu\nu} F^{\mu\nu} - \overline{\psi} M(A_\mu) \psi \right). \tag{1.2}$$

The fermions are represented by the Grassmann variables ψ and $\overline{\psi}$, M is the Dirac operator and F is the field strength tensor, which depends on the gluons represented by gauge fields A_μ.

Here I give a short introduction into the basic methods of LQCD and introduce the terminology used. For a more comprehensive introduction please see dedicated literature like *Lattice QCD for Novices* [22], *Introduction to Lattice QCD* [1], and *Quarks, Gluons and Lattices* [23].

1.1.1. The Lattice

In LQCD the space-time coordinates are discretized onto a four-dimensional lattice, which leads to the name. The points of the lattice—called *sites*—are separated by a lattice spacing, commonly called a. The connections in between the sites are called *links*. A two-dimensional projection of this is shown in Figure 1.2.

In these lattices the quarks are described by the spinor field. It is described by four components at each site. Of those components, each is a complex three-vector. The gluons are described by the gauge field. However, *link variables* U are used instead of the values A_μ at the lattice sites. The link variables are $SU(3)$ matrices, which are determined by a line integral of A_μ along the link connecting two sites. They completely represent one path in the path integral (Equation 1.1) and are, therefore, also referred to as a *configuration*.

$SU(3)$ matrices are complex 3-by-3 matrices, thus they contain 18 floating point numbers. Naming the columns of an $SU(3)$ matrix $\vec{a}$, $\vec{b}$ and $\vec{c}$, these have to obey $\vec{c} = \vec{a} \times \vec{b}$. This allows to reconstruct the last from the other two. Using this technique—commonly referred to as *Reconstruct 12 (REC12)*—only 12 floating point numbers have

to be stored in memory which conserves memory and bandwidth at the cost of additional arithmetic operations. Additional restrictions imposed by $SU(3)$ allow to eliminate 2 or 4 more floating point numbers. However, this can lead to arithmetic problems during the reconstruction [17].

The choice of lattice size influences the physics that can be observed. The length of the lattice in time direction is inversely proportional to the temperature. Thus, lattices with a large extent in the time direction are closer to the zero-temperature limit and are used for vacuum simulations. Lattices with a small extent in time direction operate in the finite temperature regime and are thus used to study thermal systems.

Despite the discretization in space-time, the path integral would still be an integral over as many integration variables as there are links. As this is infeasible, Monte Carlo methods are used for its evaluation.

It is important to note that the path integral average $\langle A(x) \rangle$ of an observable A is given by an average over paths weighted with $\mathrm{e}^{-S(x)}$. Here S is the action and x is one path:

$$\langle A(x) \rangle = \frac{\int \mathcal{D}x\, A(x)\, \mathrm{e}^{-S(x)}}{\int \mathcal{D}x\, \mathrm{e}^{-S(x)}}. \tag{1.3}$$

If for a large number of generated paths N_{cf}, the probability $P(x^{(i)})$ to obtain any particular path $x^{(i)}$ is equal to the weight for that path in Equation 1.3, then the integral can be estimated by an unweighed sum.

Sets of configurations with such a distribution of configurations can be generated using Metropolis algorithms. One starts with an arbitrary configuration $x^{(0)}$ and from this generates a new path $x^{(1)}$ via the so-called update step. This is continued until the desired amount of configurations has been generated. The key to the proper distribution is the Metropolis step. Each new configuration $x^{(i+1)}$ is only accepted if the change $\triangle S$ of the action fulfils the following inequality in comparison to a random number η.

$$\mathrm{e}^{-\triangle S} > \eta \tag{1.4}$$

The random number η is drawn from a uniform distribution in range 0 to 1. If the inequality is not fulfilled, then the new configuration is discarded and the next update step is again started from configuration $x^{(i)}$.

Successive configurations are highly correlated. Thus, when collecting N_{cf} configurations, there should be a distance of N_{cor} configurations between those kept. Here, N_{cor} must be chosen large enough to overcome the correlation length of the system. Typical values for N_{cor} are in the order of 50 configurations. Those for N_{cf} are in the hundreds. Thus, a couple thousand steps of the hybrid Monte Carlo (HMC) algorithm must be executed.

There are two common ways to choose the initial configuration for the HMC. On is using a so-called *cold* configuration containing only unit matrices. Alternatively, a *hot* configuration, filled with random matrices, can be used. The initial configuration is usually quite atypical. Therefore, the configurations generated first should be discarded. The process of reaching typical configurations is called *thermalization*. It can easily span about 1000 steps.

1.1.2. The Heatbath Algorithm

Often one is interested in so-called pure gauge theory (PGT). This theory describes QCD-like systems without fermions—or those with infinitely heavy ones. In these cases the set of configurations can be generated using the heatbath algorithm [24–26].

The heatbath algorithm follows the concept of the general Metropolis algorithm. Its update step is based on an exact algorithm for $SU(2)$ which updates each link of the lattice according to its neighbours. In $SU(3)$ this algorithm is extended by systematically reducing the $SU(3)$ link to three $SU(2)$ subgroups. Each subgroup is updated and then extended back into $SU(3)$. The product of the results of the subgroups is used to generate the new link. As the algorithm only operates on the gauge field it cannot be used if fermions are involved.

1.1.3. The HMC Algorithm

For calculations including fermions the update step is usually performed using the HMC algorithm [27]. It embeds the system into a fictitious molecular dynamics system with time τ. The algorithm relies on a Markov process to generate the configurations with the required distribution.

The operations performed by the algorithm for each step of the algorithm are visualized in Figure 1.3. The *chain* of configurations follows from the evolution of the system in fictitious time τ. The Hamiltonian of the system is given as follows:

$$H(U,\pi) := \frac{\pi^2}{2} + S(U). \tag{1.5}$$

Each update step consists of multiple molecular-dynamics steps. In each of these steps, the system is evolved according to the derived equations of motion with a force F for a time τ_0.

$$\dot{\pi} = -\partial S/\partial U \equiv F \tag{1.6}$$

$$\dot{U} = \pi \tag{1.7}$$

Schemes like leapfrog or second order minimal (2MN) [28] can be used to integrate these. The conjugate momenta $\pi(t)$, with which the evolution is started, are chosen at random from a Gaussian distribution of unit variance around zero. The same happens for the fermion field ϕ, which is held constant during the integration. The acceptance probability is calculated slightly different than noted before, it is given via the following:

$$P_A = \min(1, \mathrm{e}^{\delta H}). \tag{1.8}$$

Thus, if energy is conserved, the new configuration is always accepted as $\delta H = 0$ in that case.

The number of molecular-dynamics steps and the integration time τ_0 can be tuned to vary the acceptance rate ϵ. This rate is defined by the ratio of accepted configurations over the number of acceptance tests performed. If the ratio is too low, the simulation

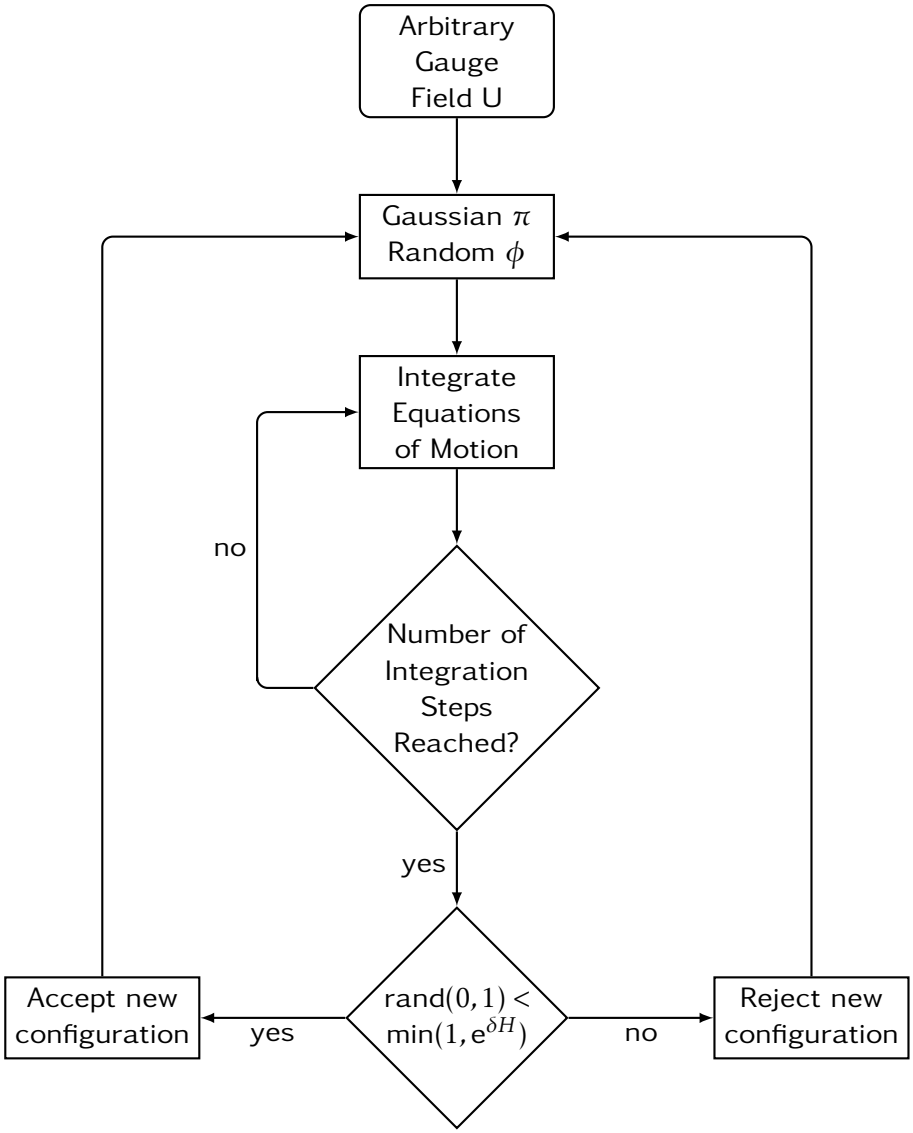

Figure 1.3.: Flow-chart of the HMC algorithm.

takes too long; however, if ϵ approaches unity, the phase space might be sampled insufficiently. Thus, the acceptance rate is typically tuned to 60 % to 80 %. This can be done by varying the number of steps used for the integration.

1.1.4. Inversions

Including the gauge fields U and the fermionic fields ψ into Equation 1.3 leads to:

$$\langle A \rangle = \frac{1}{Z} \int \mathcal{D}U \, \mathcal{D}\psi \, \mathcal{D}\overline{\psi} \, A \, \mathrm{e}^{-S(U,\psi,\overline{\psi})}. \tag{1.9}$$

The fermion fields can be integrated out exactly, yielding the determinant of the fermion matrix D.

$$\langle A \rangle = \frac{1}{Z} \int \mathcal{D}U \, A \det(D(U)) \, \mathrm{e}^{-S_{\text{gauge}}(U)} \tag{1.10}$$

This leaves only an integral over all gauge configurations. The fermion determinant, however, is expensive to calculate. Therefore, it is commonly replaced by an integral over bosonic pseudo-fermion fields ϕ_R and ϕ_I.

$$\det(D(U)) \sim \int \mathcal{D}\phi_R^\dagger \, \mathcal{D}\phi_I \, e^{-\phi_R^\dagger D^{-1}(U) \, \phi_I} \tag{1.11}$$

This leads to the following effective action:

$$S_{\text{eff}}(U, \phi) = S_{\text{gauge}}(U) + \phi_R^\dagger D^{-1}(U) \, \phi_I. \tag{1.12}$$

The dimension N of the matrix D is given by the lattice volume $V_{4\text{d}} = N_x \cdot N_y \cdot N_z \cdot N_t$, the Dirac indices N_D, and the number of colours N_c:

$$N = V_{4\text{d}} \cdot N_D \cdot N_c. \tag{1.13}$$

The actual inversion of the matrix is impractical, but the evaluation of the joined expression $D^{-1} \, \phi$ is equal to solving a system of linear equations:

$$\psi = D^{-1} \, \phi \Rightarrow D\psi = \phi. \tag{1.14}$$

A common approach to solving these equations are Krylov space solvers like conjugate gradients (CG) and the biconjugate gradient stabilized method (BiCGSTAB) [29]. These iterate to the value of ψ by starting with a trial solution ψ_0 and repeatedly evaluating $D \, \psi$ to refine it until the result has reached the desired precision. Their convergence rate depends inversely on the mass of the simulated quarks and the total volume of the lattice.

The convergence rate can be improved by using *even-odd preconditioning* [30]. It takes advantage of the structure of the fermion matrix. This allows to choose matrices L and U with known inverses such that $\tilde{D} = L^{-1} D U^{-1}$ only has two non-zero parts. The upper left quadrant is a unit matrix. Thus, the solver can ignore the even sites as they are not modified by multiplications with $\tilde{D}$. Therefore, for each spinor field variable only half

the sites must be computed and stored. The other non-zero quadrant is the lower right quadrant, mapping odd sites to odd sites. Its smallest eigenvalue is about twice that of the smallest eigenvalue of D. Thus, solving $\tilde{D}\tilde{\psi} = \tilde{\phi}$ is faster than solving the original equation.

The inverse of D does not only show up in operator evaluation but also also in the fermionic part of the force F required by the Hamiltonian equations of motion. Thus, the solver is required in each integration step in the molecular-dynamics part of the HMC. *Mass preconditioning* allows to speed up the solvers in this case. It introduces additional pseudo fermions of a larger mass, for which the Krylov solver converges faster [31].

1.1.5. The Actions

The discretization of the continuum theory onto the lattice is not unique, as only the continuum limit is required to match continuum theory. Different discretization experience different discretization artefacts.

One specific artefact is that the naïve discretization increases the original number of fermion species by a factor of 2^4. To get rid of these so-called *doublers*, improved actions are introduced. These add redundant operators to the theory that remove the doubling. An example are *Wilson fermions* [32], which, however, break chiral symmetry.

Another prominent method to discretize the fermionic part of the action are *staggered fermions* [33]. Staggering shifts the fermionic lattice in relation to the gauge lattice. One numerical consequence for staggered fermions is that at each site of the lattice there is only one spinor component instead of four. Doublers are still present in the staggered discretization, but their number reduces from 15 to only three.

Recently the method of *domain wall fermions* [34] has also picked up traction. In this variant a lattice version of chiral symmetry is preserved. However, this comes at much higher cost, since a fifth dimension has to be introduced. As this dimension must be large, these are much more expensive than Wilson or staggered fermions. A more feasible variant are *overlap fermions* [35]. These are a relative to domain wall fermions but only require four-dimensional calculations.

A variant of the Wilson fermions which is important in the context of this thesis are *twisted-mass Wilson fermions* [36]. These add a second mass parameter μ. The mass μ allows to avoid so-called exceptional configurations for which the fermion matrix has very small eigenvalues and the inversion becomes ill-defined. More importantly, however, the parameters can be tuned such that the mass is determined solely by μ. In this *Maximal Twist* scenario the first order discretization errors vanish.

Common to the usual methods of discretization is that the fermion matrix D can be split into a diagonal part and the $\not{D}$ operator[1].

$$D = M_{\text{diag}} + \not{D} \tag{1.15}$$

The $\not{D}$ operator embodies the quark kinematics and the interaction between quarks and gluons. It is a sparse matrix acting only on nearest-neighbour sites with the following

[1] $\not{D}$ is pronounced as 'D slash'.

operational form:

$$\not{D}(i,j) \propto \frac{1}{2}\sum_{\mu=1}^{4}\Big(U_\mu(i)(1+\gamma_\mu)\delta(i+\mu,j) + U_\mu^\dagger(i-\mu)(1-\gamma_\mu)\delta(i-\mu,j)\Big). \tag{1.16}$$

The γ_μ are the usual Gamma Matrices which permute rows potentially multiplying the row with -1, i, or $-i$. $\not{D}$ is commonly not stored in memory but implemented by a routine that calculates the action of $\not{D}$ on a spinor field.

For each site of the resulting spinor field, the $\not{D}$ routine loads the spinors of the neighbouring sites in all four dimensions as well as the connecting links. For each direction the spinor is projected using the γ matrices and multiplied with the $SU(3)$ matrix of the connecting link. Finally, the multiplication results are summed up to form the resulting spinor.

The $\not{D}$ operator commonly dominates the execution time of LQCD applications. Therefore, it is the primary target for optimizations of LQCD applications and its performance is often used to characterize the performance of hardware and implementations.

1.1.6. Computational Costs

I want to end this section with a remark on the computational cost of LQCD calculations. Those can be estimated using the following formula [37]:

$$\text{cost} \propto \frac{1}{m_\pi^6}\frac{L^5}{a^7}. \tag{1.17}$$

The computational costs increase extremely when attempting to slightly improve the accuracy of the calculation. Increasing the edge length $L = \sqrt[4]{V_{4d}}$ of the lattice to increase the size of the physical problem increases the costs by the fifth power. Even more, reducing the lattice spacing to reduce the discretization errors scales the costs by the seventh power. Finally, reducing the pion mass of the investigated system also increases the costs by the sixth power. Thus, typically systems with an unphysical pion mass are used.

In addition, as the Euclidean path integral is evaluated using stochastic methods, the results carry stochastic errors in addition to the systematic ones. This is not taken into account in the above formula. Halving the stochastic errors requires four times as many configurations. This adds another cubic proportionality into the cost estimation.

The inherently serial nature of the HMC algorithm requires a single, fast machine. Otherwise no sufficiently long chain of configurations can be created in finite time. Studies of thermal systems—those with small extent in the time direction—require simulations over a large range of parameter values. Each set of parameter values requires an own instance of the HMC. Thus, in these cases throughput is important, too. Finally, the operator evaluation is a sum over the operator values for each configuration in a chain. These can be spread out onto as many machines as there are configurations. Thus, in this case, the overall throughput of the available machines is more important than single machine performance.

1.2. Traditional Lattice QCD Systems

In the past decades a wide variety of systems has been utilized for LQCD computations. Beside classical CPU systems a variety of specialized systems was utilized and is still in use.

1.2.1. PCs and PC Clusters

As personal computers (PCs)—and clusters based on them—are widely available, a multitude of LQCD applications exist for them. Examples are MILC Lattice Computation (MILC) [38], Chroma [39], the Columbia Physics System (CPS) [40], and tmlqcd [3]. All of these packages also support other architectures. The latter has been a reference implementation for CL^{2}QCD.

In the latest generation of CPUs the Advanced Vector Extensions (AVX) provide a single instruction multiple data (SIMD) width of eight for single-precision (SP) and four for double-precision (DP). CPUs usually clock at about 2 GHz to 4 GHz, delivering up to 64 GFLOPS per core in SP utilizing fused multiply-add (FMA). Currently, a typical CPU has about four to eight cores. Typical memory bandwidths range from 10 GB/s to 50 GB/s per CPU. To make up for the large latencies to memory, CPUs provide large caches in the order of 10 MiB.

A cluster of 128 nodes can provide multi-TFLOP performance in LQCD calculations [41]. To achieve this, the applications utilize vectorization, cache-friendly tiling, and compression techniques to minimize the required bandwidth.

1.2.2. APE

The APE processor family [42] of computing engines is optimized for numerical simulations of lattice gauge theories. All APEs are based on custom, very-large-scale integration (VLSI) processors.

The first prototype of the original APE was constructed in 1986. It was build from 16 nodes, each containing one processor core. A centralized switching network connected them to the 16 memory bars of the system. Special about this architecture is that the nodes act as a single SIMD processor together. They operate in lock-step on a common instruction stream.

To cope with the special demands of LQCD, each node has a pipelined floating-point unit (FPU) optimized for computations on complex numbers. In addition, the memory implements periodic-boundary semantics. Programming is done in a special APE language, which is similar to Fortran. The original APE provides a performance of about 1 GFLOPS.

The follow-up to the APE in the early 1990s was the APE-100 [42, 43]. The concept was upgraded to a performance of about 100 GFLOPS by updating to more current technology. It utilizes 4096 nodes. Each node contains one FPU and one memory bank. The FPU has access to a large register file of 128 registers of 32 bit. Instructions are still performed in lock-step over all nodes, effectively providing one very large SIMD

processor. Only floating-point operations are performed on the nodes. All integer operations, typically address calculations and loop handling, are performed by the central controller processor. This has the advantage of reducing the size and complexity of each node.

Communication is no longer performed using a switching network but via next-neighbour communication. For this, the nodes are organized into a three-dimensional lattice. Data transfer is performed by pushing the data into one of the six available directions. Given the SIMD nature of the machine, this is similar to a rotation operation, only that it occurs at a memory address instead of inside a register.

The latest member of the APE family is the apeNEXT [44] from 2006. It increases the performance of the APE to the multi-TFLOPS scale. Again, it utilizes three-dimensional neighbour-to-neighbour communication. However, other than its predecessors, it does not use a SIMD approach. Instead, its processors utilize a single process multiple data (SPMD) architecture, synchronizing by communication between the processors. Programming of the apeNEXT is performed in C or TAO.

In the apeNEXT the floating-point capabilities of the processors have been explicitly matched to the memory throughput and arithmetic density of the Dirac operator. To improve the available bandwidth, separate access paths are available from the processor to the memory and to the network. In addition, prefetch queues are used. The processors are clocked at 200 MHz. This results in a peak performance of the processor of 1.6 GFLOPS. The memory bandwidth is 3.2 GB/s and communication with the neighbours can be performed at 0.2 GB/s. The latency for next-neighbour communication is only about two to three times that of a memory access.

The machine is assembled by putting 16 processors on a single board. 16 boards are contained in one crate, and two crates form a single rack. Also most other dedicated LQCD machines use a similar assembly pattern.

1.2.3. QCDSP

The Quantum Chromodynamics on Digital Signal Processors (QCDSP) machine [45, 46] won a Gordon Bell Prize for lowest cost per delivered performance in 1998. It provided 0.6 TFLOPS using 12 288 nodes at $13.2/MFLOPS.

QCDSP is a multiple instructions multiple data (MIMD) architecture based on the Texas Instruments TMS320C31-50. To form a node, the processor is complemented with 2 MiB of random access memory (RAM) and a custom, application-specific integrated circuit (ASIC). The ASIC provides a small cache and communication with the nearest neighbours in a four-dimensional mesh.

The machine is assembled by attaching 63 node cards to a motherboard. The latter contains a 64th node. Eight motherboards are combined via one backplane. Finally, the system is made up of a varying number of backplanes.

Programming of the machine occurs in C and C++ utilizing cross compilers. Assembly language is used to optimize the application kernels. Communication utilizes a message-passing library. Login to the system is performed via a Unix node.

1.2.4. QCDOC

The QCDOC [46] machines from 2004 are the successors to the QCDSP machines. Instead of combining a digital signal processor (DSP) with an ASIC on a node card, they utilize a System on a Chip (SoC) manufactured by the International Business Machines Corporation (IBM). The design goal was to reach 10 TFLOPS at $1/MFLOPS. Communication is based on a six-dimensional grid instead of only four dimensions. This allows to partition the machine without negative effects on performance.

The SoC contains a PowerPC 440, a 64-bit FPU, 4 MiB of memory, 24 bit-serial communication links, two Ethernet controllers and a double data rate (DDR) RAM controller. The bit-serial links are used for communication with the nearest neighbours. Their total bandwidth is 1.3 Gbit/s. The Ethernet links are used for input/output (I/O), booting, debugging and diagnostics. The PowerPC 440 features an L1 cache for data and instructions. It operates at 420 MHz.

QCDOC provides some distinct architectural features. One is the on-chip memory, which can later also be found in the Cell Broadband Engine processor and GPUs. Another is that the serial links have direct access to the memory controller. In addition, all communication is coherent to the L1 caches. Here, special logic is involved to avoid unnecessary flushing of cache lines by communication. Also, the hardware has special support for broadcast operations and global sums, which are often a performance problem in nearest-neighbour communication networks.

QCDOC utilizes a custom kernel on the nodes. It supports only two threads, one for the kernel and one for the application. There is no scheduling. Context switches only occur at initiation and termination of an application. While a memory-management unit (MMU) exists, it is not used for address translation. It only serves to protect memory regions and to be able to mark memory as transient.

Programming of the machine is performed in C and C++ using cross-compilers. Again, compute kernels are optimized in assembly language. The LQCD-specific message-passing library QCD Message Passing (QMP) is available in addition to a basic message-passing library. However, standard Message Passing Interface (MPI) is not supported. Its additional functionality is not required for LQCD computations.

The machine is assembled very similar to the QCDSP. Two nodes are combined on one daughter board. Each mother board combines 32 daughter boards. Four mother boards are assembled on one backplane, of which two are filled into one create. Of these, two can be stored in one water-cooled rack.

1.2.5. Blue Gene

The Blue Gene [47–49] series by IBM is not used solely for LQCD computations. The gene part of its name hints at protein folding, which is another problem this machine was designed for. Still, it is based on the concepts and architecture of the QCDOC machines. In 2004, the Blue Gene/L, the first generation of the Blue Gene, was the first system to provide a sustained TFLOPS in LQCD computations. Today, the Blue Gene/Q, the third generation of the Blue Gene, allows LQCD computations at the

PFLOPS level. In addition, Blue Gene systems have always held top positions in the TOP500 and Green500 lists.

Like QCDOC, Blue Gene/L is based on an SoC using a PowerPC 440 processor. However, each chip contains two cores. While the QCDOC utilized scalar FPUs, each core is equipped with a two-way SIMD DP FPU. At an operation speed of 700 MHz it can deliver 5.6 GFLOPS peak. In addition to an L1 cache, the Blue Gene also features L2 and L3 caches. Only those are kept coherent by the hardware while the L1 cache is not. The interconnect is based on a three-dimensional torus, instead of the six dimensions used by QCDOC. Using 32768 cores the Blue Gene/L can sustain 12.2 TFLOPS.

The way the machine is assembled is very similar to the QCDOC. Again, two chips are put onto one compute card. 16 compute cards create a node card, of which 32 are combined via two mid-plane cards to create a rack. Thus, each rack of a Blue Gene/L features 2048 cores.

The third Blue Gene generation, Blue Gene/Q, again is based on an SoC integrating processors cores, cache memory, memory controller, and network logic. However, the processor is now based on 18 PowerPC A2 cores, of which one is reserved for the operating system (OS) and another one is disabled to increase yield. Each core features four-way simultaneous multi-threading (SMT) to hide latencies and four-way SIMD in DP including FMA. Useful for LQCD is that the FPU can also operate on complex types. For those the vector width is two. At 1.6 GHz the compute chip provides a peak performance of 204.8 GFLOPS.

To feed the FPUs, the L2 cache provides a bandwidth of 563 GB/s and is filled by a dedicated prefetch engine. Both network and DDR3 memory can provide about 40 GB/s. To avoid the overhead of the modified-exclusive-shared-invalid (MESI) protocol, a back-invalidate architecture is used in the memory system. Each write to memory causes invalidation messages. A consistent view of the memory is ensured after synchronization.

The assembly of the machine is basically the same as for the Blue Gene/L. A node board is formed from 32 compute nodes. Two mid-plane cards connect 16 node boards. Finally, two of these sets form a rack of 1024 compute nodes.

The Blue Gene runs a Unix system, utilizing a custom kernel on the compute nodes. Code can be written in C++ or Fortran. The BAGEL system provides a domain-specific compiler for LQCD.

1.2.6. QPACE

The QPACE [7] machines combine an IBM PowerXCell 8i with a custom interconnect based on a field-programmable gate array (FPGA). A set-up of eight racks reaches a performance of 200 TFLOPS.

The IBM PowerXCell 8i is a relative of the Cell Broadband Engine processor used in the Sony Playstation 3 gaming console. It features a PowerPC based Power Processing Unit (PPU), eight Synergistic Processing Elements (SPEs) and a DDR2 memory interface. All components are connected via the Element Interconnect Bus (EIB) with a bandwidth of up to 200 GB/s. The SPEs combine a Synergistic Processing Unit (SPU), a

Memory Flow Controller (MFC), and a local data store of 256 KiB. The compute power stems from the SPUs, which can perform two DP multiply-add operations per cycle. This results in a peak performance of 102 GFLOPS in DP. The bandwidth to the external memory is 25.6 GB/s.

The nodes are interconnected via a three-dimensional torus implemented using Xilinx Virtex-5 LX110T FPGAs. Communication is performed via messages. As latency is important, the network enables direct transmission from SPE to SPE. The torus network is only used for communication of data between the processors. Ethernet is utilized to boot the nodes and perform I/O. An additional signal network is utilized for global conditions, synchronization and kill signals. Based on an FPGA, the network can also be reconfigured for other usage scenarios. For HPL—used to benchmark the system for the TOP500 and Green500—the communication between the main memories is more important. Using a reconfigured network, QPACE topped the Green500 in November 2009 and June 2010 with 723 MFLOPS/W.

The machine is assembled similar to all other LQCD-specific machines. A node card combines one IBM PowerXCell 8i with the Xilinx Virtex-5 LX110T and 4 GiB of RAM. Each backplane combines 32 node cards and two root cards. Up to eight backplanes are combined to one rack. Thus, one rack contains up to 256 nodes with a beak performance of about 26 TFLOPS.

1.3. Utilization of GPUs

GPUs promise to deliver a lot of computational power, and the LQCD community is used to utilizing unconventional hardware to fulfil their computational requirements. As a result, once GPUs became programmable various research groups studied how to best utilize GPUs for LQCD computations.

1.3.1. 'Lattice QCD as a video game'

The first paper [18] presenting the use of GPUs for LQCD was published in 2006. Egri *et al.* show a performance of 33 GFLOPS on the NVIDIA GeForce 8800 GTX in SP.

They used the Open Graphics Library (OpenGL) to access the GPU. Therefore, they had to map LQCD to the graphics pipeline, making each site of the lattice a pixel of a texture. As each pixel contains only four elements per site, multiple textures are used to represent a single field. The computation is done in pixel shaders, which are programmed using the C for Graphics (Cg) language. One restriction of this approach is that the amount of threads is defined by the amount of pixels in the resulting texture. For each pixel a single thread is spawned. That thread can only write to that single pixel.

As the GPUs in 2006 were incapable of DP calculations, they implemented a mixed-precision solver. In the mixed precision solver all DP calculations are carried out by the CPU. For this mixed-precision solver they report a speed-up of five when comparing the NVIDIA GeForce 7900 GTX with a CPU.

They found synchronous data transfers to be a major performance bottleneck. Therefore, they used the OpenGL extension Pixel Buffer Objects (PBO), which allows to asynchronously read back results from the GPU to the host. Another obstacle were the limited control-flow capabilities of GPUs. To solve this, they removed all loops and conditionals from the shaders.

Based on OpenGL and Cg, their implementation should actually work on all GPUs. Yet, they only report performance numbers for GPUs by NVIDIA.

1.3.2. QUDA

QUDA [17] is probably the most prominent implementation of LQCD on GPUs. It provides high-performance $\not{D}$ and solver implementations that can be used from MILC, Chroma, CPS, and other LQCD frameworks. Initially it only supported Wilson and Wilson-Clover discretizations but others have been contributed [50]. Thus, it now covers a wide variety of discretizations.

The $\not{D}$ provided by early versions delivers 135 GFLOPS in SP, 40 GFLOPS in DP, and 225 GFLOPS in half-precision on an NVIDIA GeForce GTX 280. To achieve this performance, QUDA compresses the gauge field and performs similarity transformations to increase the sparsity of the matrix. The DP CG built on this $\not{D}$ implementation achieves 38 GFLOPS. In addition, QUDA provides a mixed-precision solver that exceeds 100 GFLOPS on the NVIDIA GeForce GTX 280.

QUDA has been extensively tuned for high performance in multi-GPU scenarios. In 2010 QUDA was shown to exceed 4 TFLOPS in weak scaling on 32 NVIDIA GeForce GTX 285 [16]. This probably makes it the first successful multi-GPU code for LQCD. Back then, QUDA only parallelized in time direction. For lattices of size $32^3 \times 256$ overlapping communication and computation was found to be a worthy optimization. But, for lattices of size $24^3 \times 128$ it was slower than using synchronous communication.

For Wilson-Clover and improved-staggered discretizations QUDA scales beyond 100 GPUs [2]. For this, it uses an additive Schwarz domain-decomposed preconditioner for the generalized conjugate residual (GCR) solver, which the authors call GCR-DD. Scaling reaches a limit at 256 NVIDIA Tesla M2050 GPUs.

On the recent Fermi architecture QUDA reaches up to 300 GFLOPS in SP [51]. For this it utilizes a cache-friendly streaming pattern which exploits the locality of spinor loads in the spatial dimension. Temporal locality is not exploited due to limits imposed by the shared-memory size and lack of global thread synchronization capabilities. The achieved performance is 79 % of that which could be reached with perfect memory reuse.

As the name implies, QUDA is based on NVIDIA CUDA. Thus, it is limited to the hardware by NVIDIA.

1.3.3. Other Efforts

Ibrahim *et al.* studied the effect of coarse-grained versus fine-grained parallelism in the $\not{D}$ kernel [52]. Coarse-grained parallelism means the well established technique of

each thread calculating one or more output spinors. In their approach of fine-grained parallelism the contribution to $\not{D}$ for each dimension is calculated by separate threads. A final reduction step then creates the output spinor. This increases the amount of threads that can be utilized by a factor of 8 or 16. The major challenge in this approach is branching. However, most branches can be avoided by doing address calculations in a special way. Another issue they identified is the size of the register file and the shared memory, which limits concurrency. On an NVIDIA GeForce 8800 GTX they reach 7.5 GFLOPS, outperforming the coarse-grained implementation by 19 %.

In 2010 the first application of domain-decomposition [4]—splitting the problem into separately solved subproblems—to GPU code has been shown by Osaki and Ishikawa. They used a restrictive additive Schwarz (RAS) preconditioner to utilize eight NVIDIA GeForce GTX 280 [53].

Bonati *et al.* implemented a Rational Hybrid Monte Carlo (RHMC) for the staggered discretization that performs all calculations on the GPU [15]. It was initially written in NVIDIA CUDA but extended to OpenCL via an abstraction layer. Run on the same GPU, the NVIDIA CUDA version outperforms the OpenCL one. The AMD Radeon HD 5870 shows similar performance to the NVIDIA Tesla S2050 but is limited to small lattices by device memory. They found the performance of the AMD Radeon HD 5870 to be limited by rather large kernel launch latencies. This implementation scales to multiple NVIDIA Tesla C1060 in the same node at an efficiency of 80 % to 90 %. Another full HMC, using a variant of domain-wall fermions, has been implemented by Chiu *et al.* But, that implementation utilized only NVIDIA CUDA [54].

Many more applications have been implemented on the NVIDIA CUDA platform. Among them are the generation of pure gauge lattice configurations [55]. Multiple NVIDIA Tesla M2070 have been utilized to calculate the overlap operator [56]. Also, the use of memory-lean single-mass solvers to solve multi-mass problems was investigated [57]. For these usually multi-shift solvers [58] are used. These solve the system for all masses using the same number of matrix-vector products as solving the system for a single mass. The presented implementation outperforms those by a factor of two using single-mass solvers.

LQCD has also been implemented on the Intel Xeon Phi. The implementation [59] resembles a CPU program. But, like GPUs, the Intel Xeon Phi is a many-core processor that is attached to the node via Peripheral Component Interconnect Express (PCIe). Yet, it also differs from a GPU, as it runs its own OS and can directly access the InfiniBand (IB). The CG runs completely on the Intel Xeon Phi and—in SP—scales to 3.9 TFLOPS on 32 Intel Xeon Phis.

Another OpenCL based implementation of $\not{D}$ has been reported by Kowalski and Shen [60]. However, they have only tested their implementation on CPUs and the NVIDIA Tesla C1060.

1.4. Conclusion

LQCD is a problem that is both complex and compute intensive. To cope with its huge demand for computational power and memory bandwidth, the LQCD community has continuously adapted and developed new platforms. This repeatedly required to port thousands of lines of code [61].

In early, dedicated machines RAM was fast compared to the processors. Therefore, each processor mapped the memory of its neighbours. As cores sped up, memory speed became a limiting factor. Thus, fast local memories and complex cache hierarchies were introduced. This made communication the prime problem to cope, as data of neighbouring processors is no longer directly accessible. As LQCD does not require full connectivity, dedicated machines put a lot of emphasis on low-latency, high-bandwidth communication between neighbouring nodes. Therefore, they usually utilize three- or four-dimensional torus networks, which can often transfer data directly between the caches of the processors.

Over the years there has also been a shift from full custom designs to more commodity hardware. While the APE utilized a custom-designed processor, the Blue Gene supercomputers are a pre-packaged solution. However, they are not exactly commodity. The most recent development in this aspect is the utilization of GPUs. Those offer a lot of memory bandwidth to a single compute chip. However, communication is tricky, as they are attached to the node via PCIe. Interestingly, they are similar to early dedicated machines, as they give many cores access to a common homogeneous memory.

The big machines—like the Blue Gene—are commonly used as capacity machines to generate the configurations for later evaluation. In the analysis stage—which depends less on latency and more on throughput—trivial parallelism is then exploited to spread the calculation out to smaller machines, e.g. single GPUs.

Many efforts to utilize GPUs for LQCD were performed after NVIDIA introduced the NVIDIA CUDA platform. However, while platform independent standards for GPU programming exists, there are hardly implementations that do not base on NVIDIA CUDA. Even fewer implementations have been tested on non-NVIDIA hardware, which makes the community quite dependent on this single vendor.

Chapter 2.

GPGPU

The term General-Purpose computation on Graphics Processing Units (GPGPU) describes the concept of utilizing GPUs for computations not related to rendering. A very early example of this concept was described by Hull in 1987 [62]. He used the blitting engine of the Amiga computer to speed up the Game of Life. However, the term itself only became popular in the last decade.

Early endeavours into GPGPU utilizing the shader units can be found in *Gpu Gems 2* [63]. There, among other applications, a fast Fourier-transform (FFT) implemented on GPUs is presented. Today, GPGPU can by found in consumer software. One example of this are games, where GPGPU is used to accelerate the physics engine. Another example are real-time shakiness removal filters, which allow real-time processing even on mobile devices.

Table 2.1 shows an overview over a variety of GPUs and CPUs which went to market in recent years. The GPUs surpass the CPUs in peak performance as well as in memory bandwidth. The same can also be seen in Figure 1.1, which shows the development of CPU and GPU performance in the last years. The raw performance advantage given by the GPU immediately shows why they are also an interesting platform for compute intensive workloads other than graphics.

To take advantage of GPUs, an appropriate programming model and a certain understanding of the hardware architecture is required. The most prominent programming model is NVIDIA CUDA, which is the proprietary framework by NVIDIA and currently limited to hardware by this vendor. Its sibling OpenCL is an open standard and available on a wide variety of platforms. Therefore it is used for the work presented in this thesis.

2.1. GPUs as General-Purpose Many-Core Processors

Like modern CPUs, GPUs are multi-core SIMD processors. However, while CPUs are tuned towards low-latency processing of individual threads, the GPU architecture is tuned towards high throughput over thousands of threads. The different design orientation can easily be understood by looking at the original purpose of GPUs. When rendering thousands of pixels, many completely independent calculations must be performed. However, as there is little use in a partial image, there is no need to render individual pixels especially fast. Only the rendering time for the whole image is relevant.

Table 2.1.: Theoretical peak performance of a variety of GPUs and CPUs. BW denotes bandwidth

	Chip	Peak SP GFLOPS	Peak DP GFLOPS	Peak BW GB/s
AMD Radeon HD 5870	Cypress	2720	544	154
AMD Radeon HD 6970	Cayman	2703	683	176
AMD Radeon HD 7970	Tahiti	3789	947	264
AMD FirePro S10000	Tahiti	2×3410	2×850	2×240
NVIDIA GeForce GTX 280	Tesla	933	78	142
NVIDIA GeForce GTX 480	Fermi	1345	132	177
NVIDIA GeForce GTX 680	Kepler	3090	258	192
NVIDIA Tesla K20	Kepler	3520	1170	208
AMD Opteron 6172	Magny-Cours	202	101	43
AMD Opteron 6278	Interlagos	307	154	51
Intel Xeon E5-2690	Sandy Bridge EP	371	186	51

2.1.1. The Execution Model

The SIMD model of CPUs is based on vector registers. A single processor executes an instruction in parallel for multiple elements in the vector register. GPUs implement a variant of this termed *single instruction multiple threads (SIMT)*. The registers are seen as scalar by each thread. But, groups of execution units—called *processing elements (PEs)* in OpenCL terminology—share a common instruction decoder. Therefore a core—called *compute unit (CU)* in OpenCL—always executes a group of threads in lock-step.[1] The group of lock-stepped threads is equivalent to the SIMD thread on a CPU.

The lock-stepped group is called *warp* by NVIDIA and *wavefront* by AMD. One or more of these groups make up a *work group*. All threads of a work-group are guaranteed to be scheduled to the same CU and can synchronize and communicate with each other.

Threads of different work groups cannot communicate with each other.[2] This allows GPUs to rather freely schedule the work groups depending on resource availability. Thus, two work groups might be scheduled concurrently on two separate CUs, sequentially on the same, or even concurrently on the same. In consequence, applications should be completely agnostic of the actual number of CUs available. This allows to scale the performance of GPUs simply by varying the number of CUs of the GPU.

[1] GPU manufacturers commonly use the term *(shader) core* for the PEs of the GPU. This generates much higher core counts, ideal for marketing purposes. Yet, from an architectural point of view and for consistency with other processor architectures, the CU is the actual core of a GPU.

[2] Modern GPUs provide atomics on global memory and allow for limited communication between work groups. Yet, while useful for some algorithms, inter-work group communication can cause problems, as it requires the work groups to be scheduled in the order expected by the algorithm. Thus, these algorithms usually must be adjusted for each GPU.

The SIMT model has some advantages over the SIMD model. Diverging control-flow conceptually comes for free. The hardware can mask out threads if their instruction counter does not match that of the instruction currently processed by the group. Thus, the programmer does not have to worry about masking out operations for elements or backing up and restoring single elements as it might be required with vector registers. Of course, this also comes at the danger of the programmer forgetting about the single-instruction nature of the hardware. This can lead to degraded performance as—unless all threads in a group take the same branch—the hardware will have to execute all branches and mask out the threads not active in the current branch. This problem, caused by diverging control flow within a lock-stepped group of threads, is known as *warp serialization*.

The memory access is more flexible in the SIMT model. Scatter and gather memory access comes naturally, as each thread performs loads and stores to individual addresses. The hardware tries to coalesce nearby memory accesses by threads of a group into single, larger memory transfers. Thus, full performance can be reached for suitable patterns. Memory access patterns completely ignoring the grouped execution of threads and the coalescing performed by the hardware can result in an increased amount of memory transactions and, thus, reduced performance.

GPUs provide a larger set of registers than CPUs. The AMD Radeon HD 5870 provides 16 384 registers, each 128 bit in size. The NVIDIA GeForce GTX 480 provides 32 768 registers of 32 bit each. These registers are mapped to threads dynamically. Therefore, a compiler can trade the number of threads in flight versus the number of registers available for each thread. On the one hand, it can host a smaller number of threads each using even more than hundred registers. On the other hand, if each thread uses only a dozen registers, more than a thousand threads can be in flight concurrently on a single CU. Like hyper-threading on a CPU, running more threads will allow hiding memory latencies, increasing overall throughput. The scheduling of the thread groups is performed by a hardware scheduler with minimal overhead. Registers stay allocated to each thread from its creation until it finishes execution.

2.1.2. The Memory Model

The memory architecture of GPUs explicitly exposes more complexity than that of CPUs, which appears uniform to the user. It is split into multiple logical regions. Figure 2.1 schematically shows where the different regions are located on a GPU.

Global memory is the normal main memory of the GPU that can be read and written to by all threads running on the GPU. Accessing this memory incurs high latencies in the order of a thousand cycles. This memory can also be read and written from the host. The host is also responsible for managing allocations of global memory. Such memory allocations on the GPU are referred to as *Buffers* by OpenCL.

Private memory is a part of the GPU's main memory that is partitioned among all threads running on the GPU. When addressing into private memory each thread accesses its own partition. This memory is also used to place spilled registers if the register file cannot hold a thread's full working set. These registers are also known as

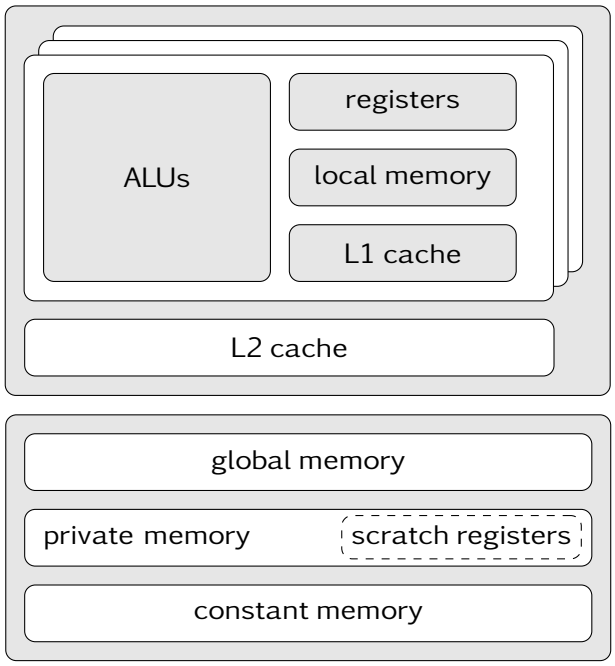

Figure 2.1.: The GPU memory is split into multiple regions. Registers and local memory are located within the CUs. The main memory of the GPU, not located within the GPU chip, is used for global memory, private memory and constant memory. Caches are not always available and might only be used by special memory requests.

scratch registers. As private memory is part of the GPU's main memory, it shares the performance characteristics of global memory. This means, large latencies can be incurred when accessing data from local memory. Therefore, usage of scratch registers usually comes with a large performance penalty. On recent GPUs, this might be mitigated by caches.

Another part of the GPU's main memory is used for *constant memory*. That memory can only be written to by the host. GPUs are usually able to cache accesses to this memory and broadcast values from this memory to all threads very efficiently.

In addition, modern GPUs also provide a *local memory*. Just like registers, local memory is on-die and can be accessed with similar performance. Local memory is shared between threads running on the same CU and can be used as a user-programmed, explicit cache.

Stemming from their graphics tradition, GPUs originally only had dedicated read-only caches for constant memory and textures. Textures are images stored in global memory in a special format. On the AMD Radeon HD 5870 and AMD Radeon HD 6970—when keeping to some restrictions—the AMD OpenCL compiler is capable to automatically utilize the texture cache to access buffers which are only read by a kernel. More modern GPUs—like the NVIDIA GeForce GTX 480 and the AMD Radeon HD 7970—provide multi-level read-write caches. While CPU caches aim to minimize latencies in memory access for a single thread, GPU caches are shared by many threads. One of their main functions is to enhance the GPU's capabilities of coalescing accesses by multiple threads to close-by addresses into single memory transactions. The first level of the caches is typically located within each CU, while the second level of the cache is either global or shared in between clusters of CUs.

2.1.3. GPU-Performance Explained

As shown in Table 2.1, GPUs offer higher peak performances than CPUs. The larger amount of available FLOPS does not stem from the clock rate of the GPUs. While many CPUs run at up to 4 GHz, GPUs commonly clock at about 1 GHz.

As mentioned above, GPUs are optimized for throughput instead of latency. CPUs spend a lot of silicon real-estate on control logic. This includes the logic for speculative execution, which is required to minimize latencies for a single thread. The GPU does not implement such features and relies on running many threads interleaved to hide memory latencies. Thus, it can spend a much larger portion of the silicon real-estate for FPUs.

For the higher peak bandwidth it is most relevant that GPUs usually have dedicated on-board memory. This allows to utilize very wide buses to access the memory. For example, the width of the AMD Radeon HD 7970's memory bus is 384 bit. Also, as the GPU manufacturer knows the particular memory model that will be used and as there is no socket connection, timing parameters of the memory can be chosen more aggressively.

This is different if the GPU is integrated with the CPU. In this case the GPU uses the same bus as the CPU to access the memory. Thus, it only reaches the same peak

bandwidth as the CPU. An exception to this is the GPU of the Sony Playstation 4, where the chip containing both CPU and GPU accesses on-board memory via a wide bus as it is usually done in dedicated GPUs.

The architecture of GPUs also has energetic advantages. With their lower clock rates, GPUs can utilize slower, more energy-efficient transistor designs. Also, GPUs do not waste energy on speculative execution. However, GPUs with their highly parallel execution model can require memory accesses which in serial execution might be avoided by caching. As data transfers also incur energetic costs, this can increase a GPU's energy consumption in comparison to that of a CPU.

2.1.4. Traditional GPUs

Early GPUs accelerated the drawing of two-dimensional graphics primitives—lines, arcs, rectangles—and were able to perform blitting operations. The later describes the process of moving, copying and combining bitmaps. While some GPUs were actually based on general-purpose microprocessors, the exposed function set was fixed to a small amount of operations.

In the 1990s the success of 3dfx Interactive's Voodoo Graphics PCI series established hardware acceleration for 3D graphics. Soon the now dominant vendors NVIDIA and ATI, which is now part of AMD, appeared with their own GPUs. Direct3D became the dominant API on Windows, and OpenGL became dominant on Linux. GPUs of this era already possessed floating-point-computation capabilities, while 2D graphics only required bit manipulation. Yet, these GPUs still used a so-called fixed-function pipeline. The programmer could not choose the calculations performed but only influence parameters like the projection matrix used.

In the first half of the 2000s, programmable shading was added to the capabilities of the GPUs. This, for the first time, allowed programmers to run small, custom programs performing floating-point operations on the GPU. Capabilities were still limited. The NVIDIA GeForce 3—which was the first GPU to ship with these features—did not allow any loops. In addition, the programs could only be executed in certain points of the still mostly fixed-function pipeline. One point allowed to modify the position and projection of the processed vertices. The second point allowed to customize the shading process of the rendered pixels. Thus, these programs are commonly called *shaders*.

Mirroring the logical positions of the shader executions inside the fixed-function pipeline, vertex and pixel shaders were executed by different algorithmic and logic units (ALUs). By mapping the problem to be computed into a rendering problem, these GPUs could be used for generic computations. Libraries like BrookGPU [64] could somewhat hide the graphics nature from the programmer, but in the end the GPU was still performing a rendering pass. One consequence of this was that the available precision was in most cases less then SP. Also, shaders were incapable of writing to arbitrary addresses. Each thread would emit exactly the vertex or pixel it was launched for by the hardware.

In 2006 the concept of unified shaders came up. Vertex and pixel shaders are since then executed on the same ALUs. The fixed-function pipeline was dropped, which also

allows the introduction of shaders at other stages of the graphics pipeline. Not all steps are performed in software, though. At least rasterization—generating the set of pixels covered by a graphics primitive—is still performed using special-purpose hardware on all current GPUs. On the software side this conceptual change was mirrored by the release of Direct3D 10 and OpenGL 3.3.

Unified shaders are important for GPGPU, as it is now possible to utilize the ALUs outside of the graphics pipeline. They do, however, also have advantages in the traditional rendering process. Traditional GPUs had a specific amount of computational power for each of the programmable rendering stages—vertex processing and pixel shading. However, the computational demand of these stages varies from application to application. Thus, an application could be limited by the available vertex shaders while the pixel shaders were idling and vice versa. Unified shaders allow to increase the utilization of the GPU as the shaders can be used for both processes.

2.1.5. Development of NVIDIA GPUs

G80 based GPUs—like the NVIDIA GeForce 8800 GTX—where the first generation of NVIDIA GPUs to support NVIDIA CUDA—and later also OpenCL. This allowed to perform GPGPU without mapping the problem to the rendering process. These GPUs supported SP computation but not DP. Each CU contained one instruction decoder, 16 KiB of local memory, eight scalar ALUs, and a special-function unit (SFU) for transcendental operations. The ALUs ran at twice the speed of the instruction decoder. For each cycle of the instruction decoder an ALU processed two threads. Thus, the lock-step size on this generation of GPUs is 16. Cached access to the global memory was only available via the texture unit.

The next architecture by NVIDIA was called Tesla and was used in GPUs like the NVIDIA GeForce GTX 280. It is an evolution of the G80 that added support for DP arithmetics. For this, the CU units were given a single DP FPU in addition to the eight ALUs. Thus, DP operations were executed at an eighth the speed of SP.

The Tesla architecture was followed by Fermi, which introduced read-write caches for global memory. Those are always active and do not require the explicit use of the texture unit. The L1 cache shares its storage with the local memory and NVIDIA CUDA allows to reduce the size of the local memory from 48 KiB to 16 KiB to increase the size of the L1 cache. Each CU unit now contains two instruction decoders and 32 PEs. The lock-step size was kept at 16 threads. In the configuration used in the NVIDIA Tesla[3] product line, Fermi is able to provide half the SP performance in DP. For the consumer grade GPUs like the NVIDIA GeForce GTX 480 this ratio was kept at one eighth. Starting with this generation the NVIDIA Tesla product line also supports error correcting codes (ECC).

With the Kepler architecture the CUs became even larger. Each contains four instruction decoders that distribute their work to 192 PEs and 32 SFUs. The DP performance is reduced to one third of the SP performance on NVIDIA Tesla devices and to 1/24th

[3] NVIDIA uses the name Tesla for both one of its GPU generations and for a product line. Unless explicitly stated otherwise, Tesla usually refers to the product line.

on consumer GPUs. An exception is the NVIDIA GeForce GTX Titan, on which the DP performance can be switched between the two levels.

2.1.6. Development of AMD GPUs

AMD's first GPU generation with unified shaders was the Radeon HD 2000 series based on the R600 architecture. On these GPU's, AMD supported GPGPU via its Compute Abstraction Layer (CAL) technology, which required compute kernels to be written in an assembler language. In addition there was a variant of *BrookGPU* [64] which used CAL instead of OpenGL. The succeeding R700 architecture, on which the Radeon HD 4000 series was based, later received OpenCL support. However, local memory never worked efficiently on these devices.

The Evergreen series of GPUs—with its flagship AMD Radeon HD 5870 based on the Cypress chip—was AMD's first GPU with proper OpenCL support. Contrary to its NVIDIA counterparts of the Tesla and Fermi generation, it does not use scalar PEs. Each PE contains four FPUs and one SFU. The commands are encoded as very long instruction words (VLIWs) containing up to five instruction. Thus, the full compute performance can only be exploited if the compiler can extract a sufficient level of instruction-level parallelism (ILP) from the code, as each PE processes one thread. One CU contains 16 of these PEs. Instructions are repeated over four cycles, resulting in a lock-step size of 64 threads. For DP computations the four FPUs are combined into one. Thus, the DP performance is one fifth of the SP performance. The local memory available on each CU is 32 KiB. Like the Tesla generation by NVIDIA, the cache can only be used via the texture unit. However, AMD's OpenCL compiler is capable of automatically generating the required instructions for read-only buffers. Thus, for those the programmer can benefit from the texture cache without additional programming overhead.

In the following Northern Islands architecture, AMD reduced the VLIW size and the PE width to four. The ratio of DP to SP adjusted to one fourth. Otherwise the Southern Islands architecture matches the Evergreen one. The flagship GPU of this generation was the AMD Radeon HD 6970.

With Southern Islands AMD completely redesigned its GPU architecture, which is now called graphics core next (GCN). PEs are scalar and 16 of them form a vector unit. Of these, four exist in each CU. Commands are still repeated over four cycles, keeping the lock-step size at 64 threads. In addition, each CU features a scalar unit, which can be used to execute commands, like loop counters, that are scalar over a whole work group. The local memory available on each CU is 64 KiB. Like NVIDIA with its Fermi architecture, AMD also introduced full read-write caching. The DP performance is one fourth of the SP performance. Contrary to NVIDIA, AMD still provides the full DP performance in its *high-end* consumer GPUs. The flagship GPU of that series is the AMD Radeon HD 7970. It is based on the Tahiti chip, which is also the basis of the AMD FirePro S10000.

Listing 2.1: A simple scalar implementation and invocation of the saxpy routine from the BLAS library.

```
// Performs saxpy on n elements
void saxpy(unsigned int const n, float const a,
           float const * const x, float * const y)
{
  for(unsigned int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}

// Invoke saxpy on N elements
saxpy(N, a, x, y);
```

2.1.7. Other Devices

Even though not marketed as such, the Intel Xeon Phi has some similarities with the current GPUs. Though based on an x86 CPU design, its SIMD vector width of 16 is similar to the lock-step size of the GPU and its four-thread hyper-threading enables it to hide latencies via thread swapping. In addition it uses an in-order architecture to keep the control logic simple.

GPUs with GPGPU capabilities can not only be found in the PC market. The Mali GPUs by ARM fully supports OpenCL and can be found in smartphones and tablets.

2.2. Programming Models

GPU applications generally consist of a controlling program—called *host*—running on the CPU that executes suitable smaller programs—called *kernels*—on the GPU. All the memory management, like buffer allocation and data transfer between host and GPU, is performed by the host program. This model is also known as *offload computing*, as an application running on the CPU offloads some of the workload to the GPU.

In the remainder of this section I will give an overview over a selection of GPGPU programming models. To convey a sense of each programming model, I will give a simple implementation and invocation of the saxpy routine from the Basic Linear Algebra Subprograms (BLAS) library. This routine performs the following operation on arrays $\vec{x}$ and $\vec{y}$.

$$\vec{y} = a\vec{x} + \vec{y} \tag{2.1}$$

A simple scalar implementation and invocation for CPUs is given in Listing 2.1. To keep the examples short they do not include resource management and initialization.

Besides the models presented here, there is a variety of further models for GPGPU programming. One of the earliest models was to utilize OpenGL, writing the shaders

Listing 2.2: A simple implementation and invocation of the saxpy routine from the BLAS library in NVIDIA CUDA.

```
// Performs saxpy on n elements
__global__ void saxpy(unsigned int const n, float const a,
                      float const * const x, float * const y)
{
  unsigned int const threadId = blockIdx.x * blockDim.x
                              + threadIdx.x;
  unsigned int const globalSize = gridDim.x * blockDim.x;
  for(unsigned int i = threadId; i < n; i += globalSize) {
    y[i] = a * x[i] + y[i];
  }
}

// Invoke saxpy on N elements
saxpy<<<1024, 128>>>(N, a, x, y);
```

in the Cg programming language. For Java there is Aparapi [65], which allows to formulate data parallel workloads using Java objects such that under the hood OpenCL can be used for all operations. In addition, there are experimental models like Copperhead [66], which provides a data parallel extension to the python language based on annotations and functional programming.

2.2.1. NVIDIA CUDA

NVIDIA CUDA is the most prominent tool for GPU computation. Virtually all existing LQCD applications are based on NVIDIA CUDA, at the disadvantage that these are destined to run on NVIDIA hardware exclusively [15–17, 55, 57].

Originally NVIDIA CUDA allowed to use the C subset of C++ and static C++ features like method overloading and templates. More recent versions extended this to the full functionality of C++, including features like virtual functions and exceptions. Proprietary compilers even allow to use other languages—like Fortran or Python—for the GPU code.

Listing 2.2 shows an implementation of saxpy in NVIDIA CUDA. Kernels are marked by the `__global__` annotation. The code of a kernel describes the work each thread is supposed to perform. Thus, the code first checks the index of the current thread to select the elements to operate on.

NVIDIA provides a compiler driver called nvcc, which divides each source file into the part of the code that is to be compiled by the host compiler and that part which is supposed to be compiled by the GPU compiler. In addition, it also takes care to remove all NVIDIA CUDA specific syntax from the host code, adding equivalent C++ code instead.

For cases where nvcc cannot be used, NVIDIA CUDA also provides the so-called driver API. This C-API is very similar to that of OpenCL and does not add any custom syntax. As the high-level API uses C++ features, this API is also required if no C++ compiler is available or the language used cannot bind to C++ APIs.

A major advantage of NVIDIA CUDA is that it provides access to the latest features of NVIDIA GPUs. Such, it is possible to dynamically allocate memory or launch kernels from code running on the GPU. However, despite NVIDIA officially having opened its specification for implementation by others, at the moment NVIDIA CUDA does not support any other vendor's devices.

2.2.2. OpenCL

OpenCL is a hardware independent approach to parallel computing. It is an open standard to perform calculations on heterogeneous computing platforms. From an architectural and API point of view it is a sibling of NVIDIA CUDA. However, it lacks the higher levels of the API and does not exploit all features available on the NVIDIA CUDA platform. These drawbacks are partially overcome by higher level APIs built on top of it and by extensions to OpenCL.

Implementations of OpenCL are available by AMD, ARM, Intel, NVIDIA and other vendors. These implementations are not exclusively targeted at GPUs but also support CPUs and other devices. The standard allows to use multiple implementations in a single application. Such, computations can be spread over all devices available in a system.

OpenCL defines a programming language that is based on C99 and provides a C-API. Listing 2.3 shows an implementation of saxpy in OpenCL. The kernel function is annotated by `__kernel` and the pointers to data in global memory are qualified by `__global`. Like in NVIDIA CUDA, the work for each thread is described by the kernel code. Thus, the elements operated on are chosen based on the ID of the current thread.

The code example also shows that the API is more verbose than the high-level API of NVIDIA CUDA. It is a pure C API, allowing to compile the host code using any compiler and binding from virtually any language.

OpenCL expects the kernel code to be provided as a string to the API. Thus, in the source code of the application the kernel must be stored such. The example ignored this for reasons of readability. A common solution to this problem is to store the kernel in a separate source file which is read at application start.

As OpenCL code can also be run on CPUs, it can also be used if no GPU is available. Thus, it is not required to provide an alternative implementation for this case. This allows to use scripting languages like Python for the host code. Advanced OpenCL bindings—like PyOpenCL, which utilizes features of the scripting language—allow to reduce the verbosity of the host code. For these the kernel invocation shrinks to a single line, as in other programming models. In addition, features like automatic life-time handling of OpenCL objects and error handling via exceptions become available.

AMD has proposed to extend the OpenCL language by C++ features. However, currently only AMD's implementation offers support for overloading and other static C++

Listing 2.3: A simple implementation and invocation of the saxpy routine from the BLAS library in OpenCL.

```
// Performs saxpy on n elements
// The API requires this code to be passed as a string.
// Thus, it cannot be stored in the C or C++ file.
__kernel void saxpy(unsigned int const n, float const a,
                    __global float const * const x,
                    __global float * const y)
{
  size_t const threadId = get_global_id(0);
  size_t const globalSize = get_global_size(0);
  for(size_t i = threadId; i < n; i += globalSize) {
    y[i] = a * x[i] + y[i];
  }
}

// Invoke saxpy on N elements (in a normal C or C++ file)
clSetKernelArg(saxpy, 0, sizeof(cl_uint), &N);
clSetKernelArg(saxpy, 1, sizeof(cl_float), &a);
clSetKernelArg(saxpy, 2, sizeof(cl_mem), &x);
clSetKernelArg(saxpy, 3, sizeof(cl_mem), &y);
size_t const group_size = 128;
size_t const work_size = 1024 * group_size;
clEnqueueNDRangeKernel(queue, saxpy,
                       1, nullptr, &work_size, &group_size,
                       0, nullptr, nullptr);
```

features. Hence, this feature can currently not be used without sacrificing the platform independence of OpenCL.

2.2.3. OpenGL Computer Shaders

Starting with version 4.3, OpenGL provides compute shaders. These allow to perform GPGPU calculations without adhering to the limits of the graphics pipeline. For example: kernels can read and write to arbitrary positions in the used buffers. This is especially interesting as all GPU vendors support OpenGL. Thus, this feature can be used on all GPUs supporting the latest version of OpenGL.

However, OpenGL compute shaders are meant to be used by applications that require to perform some GPGPU computation in addition to their normal use of OpenGL. It only supports a single device and operations are in-order and synchronous. In addition, OpenGL compute shaders do not require IEEE compliance for floating point operations. The shaders are written in OpenGL Shader Language (GLSL), which is also used for all other shaders in OpenGL. Their syntax is quite different from that used by CPU code.

OpenGL compute shaders provide a wide range of supported hardware. They provide an interesting alternative for OpenGL based applications that need to perform some data-processing, but the limitations of the programming model make them of limited use to scientific applications.

2.2.4. C++ AMP

C++ AMP extends C++11 to support parallel data processing. It allows to declare functions and lambdas to be using a specified subset of the language which can be mapped to CPUs and GPUs. Library functions allow the execution of these on index ranges, thus providing parallel execution. Like OpenCL this can be used to perform calculations on both CPUs and GPUs.

Listing 2.4 shows an implementation of `saxpy` in C++ AMP. The lambda specifies the work for each index of the index range. The index to be processed is passed via the argument of the lambda. The `parallel_for_each` function executes the lambda over the whole index-range, which, in this case, is chosen to be identical to the size of the array.

C++ AMP has been developed by Microsoft, which is so far the only vendor to support it. Hence, it can only be used on the Windows OS, which severely limits its use to scientific applications. AMD had originally announced to add support for C++ AMP to its version of the Open64 compiler but has not delivered on this promise, yet. In 2012 Sharlet of Intel presented an implementation based on Clang, LLVM, and OpenCL at the LLVM Developer's Meeting [67]. However, that version is not yet publicly available, either.

Listing 2.4: A simple implementation and invocation of the saxpy routine from the BLAS library in C++ AMP.

```
// Performs saxpy on n elements
void saxpy(unsigned int const n, float const a,
           float const * const x_host, float * const y_host)
{
  array_view<float const, 1> x(N, x_host);
  array_view<float, 1> y(N, y_host);

  parallel_for_each(
    y.extent,
    [=] (index<1> i) restrict(amp)
  {
    y[i] = a * x[i] + y[i];
  }
}

// Invoke saxpy on N elements
saxpy(N, a, x, y);
```

2.2.5. OpenACC

OpenACC provides a high-level approach to GPGPU computing. It defines a set of pragmas, which allow compatible compilers to extract GPU code from a serial CPU code. Incompatible compilers ignore the pragmas, producing an equivalent, single-threaded CPU code. This is very similar to the way OpenMP provides parallel execution on CPUs.

Listing 2.5 shows how the scalar implementation from Listing 2.1 can be transformed to a GPU code by a single pragma. The first pragma is optional and ensures that the contents of `x` are not transferred back to the host after the kernel execution. Further pragmas allow to keep data on device between kernels, utilize local memory, and perform other performance optimizations.

Currently all compilers that support OpenACC are proprietary.

2.2.6. OpenMP 4.0

In version 4.0, OpenMP has been extended with functionality to offload computations to other devices. The model can be viewed as a merge of OpenACC and the similar proprietary solution Intel provides for its Intel Xeon Phi. In addition to the offloading mechanism, OpenMP 4.0 also introduces functionality for vectorization.

Listing 2.6 shows how the scalar implementation from Listing 2.1 can be transformed to a parallel code executed on another device via offloading. As for OpenACC, the copy

Listing 2.5: A simple implementation and invocation of the saxpy routine from the BLAS library. using OpenACC

```
// Performs saxpy on n elements
void saxpy(unsigned int const n, float const a,
           float const * const x, float * const y)
{
#pragma acc declare copin(x), copy(y)
#pragma acc kernels
  for(unsigned int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}

// Invoke saxpy on N elements
saxpy(N, a, x, y);
```

specification for the parameters x and y is optional and ensures that the contents of x are not transferred back to the host after the kernel execution.

The specification for OpenMP 4.0 was released in July 2013. Sadly it is not yet supported by any compiler. However, support for OpenMP 3.2 has recently been added to Clang. As LLVM is already the basis of all major OpenCL compilers, this might lead to basic support for OpenMP 4.0 quite soon.

2.2.7. Conclusion

A variety of programming models for GPGPU computing exists. High-level programming models like OpenACC and OpenMP 4.0 promise to quickly transfer an existing serial CPU code to a GPGPU code. However, compilers with support for OpenACC come with a price tag and compilers supporting OpenMP 4.0 are not yet available.

Microsoft's C++ AMP also allows high-level access to GPGPU computing. But this model, too, suffers from limited compiler support and is only available on the Windows OS.

NVIDIA CUDA and OpenCL offer the most fine grained control over the hardware. Thus, they also promise to offer the best potential for code optimization. NVIDIA CUDA is less verbose than OpenCL and grants access to some features not yet available in OpenCL. However, NVIDIA CUDA is limited to NVIDIA's hardware.

Thus, OpenCL is currently the only viable option for software that supports multiple platforms. Libraries and advanced language bindings can help reduce its verbosity and reduce development efforts. Verbosity is mostly an issue of prototypes and examples, anyway. In real applications code reuse limits the effect, making it less of an issue. OpenMP 4.0 might become an interesting alternative for prototyping and porting of existing code once compilers become available.

Listing 2.6: A simple implementation and invocation of the saxpy routine from the BLAS library. using OpenMP

```
// Performs saxpy on n elements
void saxpy(unsigned int const n, float const a,
           float const * const x, float * const y)
{
#pragma omp target in(x), inout(y)
#pragma omp parallel for
  for (unsigned int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}

// Invoke saxpy on N elements
saxpy(N, a, x, y);
```

Chapter 3.

Optimization Techniques

In this chapter I will introduce several general optimization techniques that can be used to speed up applications run on GPUs. The focus will be on those optimizations which can be applied to bandwidth limited codes, as LQCD codes fall into that category [17, 20].

3.1. Bandwidth

It is an obvious limitation for every bandwidth limited code that it cannot be faster than data can be read from and written to the memory of the GPU. Therefore, it is important to make sure that neither the data types used nor the layout of the data limits the code's performance. This section analyses the effect that the choice of data type and layout has on a variety of GPUs.

3.1.1. *clBandwidth*

To measure the bandwidth that can be achieved using a certain data type and layout, the code must not perform any operations other than memory access. There are three possible simple kernels for this task:

1. A kernel that only reads data from a buffer
2. A kernel that only writes data to a buffer
3. A kernel that copies data in between two buffers

The kernel that only reads data from a buffer cannot be implemented properly. Unless some data is written out in the end of the kernel, the compiler should—and from experience will—remove all memory reads. This way the kernel achieves incredible performance numbers, but they do not reflect reality. The problem can be solved by writing out a small dataset that depends on all the input data. However, in that case the result will always also depend on the quality of the data reduction method used and the ratio of input to output data size.

Listing 3.1: A copy kernel using the float4 datatype

```
__kernel void copyScalar(
    __global float4 * const restrict out,
    __global const float4 * const restrict in)
{
    PARALLEL_FOR(i) {
        out[OFFSET + i] = in[OFFSET + i];
    }
}
```

The write-only kernel is a viable approach. As the memory written to can be read from outside the kernel, the compiler cannot cut corners and remove any writes[1]. However, for many kernels read performance is more important than write performance.

The copy kernel gives a reliable measure of read and write bandwidth available. Although it does not allow to measure read performance solely, it is at least not dependent on any device characteristic other than memory and memory controller.

The application *clBandwidth* [68]—developed in the context of this thesis—uses Python to generate[2] and execute OpenCL kernels that copy data between buffers. A simple version of such a kernel is given in Listing 3.1. It can also generate kernels that operate on non-scalar data types. For those it can use a variety of data layouts, which are shown in Subsection 3.1.4. To test the actual performance of memory and memory controller—not the size of cache—the number of bytes read and written is given by the size of the buffer. The kernel always performs exactly one copy from the source to the destination buffer.

To increase measurement quality *clBandwidth* always runs the kernel several times in a row. The measurement is stopped once the error of the mean kernel execution time drops below 1 %. This ensures that runtime variations caused by external effects, like the graphical user interface of the system, do not impair the measurement. To avoid any bias caused by the GPU powering up from idle state there are also always a few warm-up runs to wake the GPU before each measurement. This is visualized in Figure 3.1

clBandwidth swaps the input and output buffer between two kernel executions. This configurable behaviour ensures the runtime does not play any tricks by avoiding subsequent kernel executions that should obviously give the same result, as well as any cache effects. On actual hardware I have not been able to observe any difference in performance, whether swapping the buffers or not.

[1] There is the obvious exception of writing to the same memory address multiple times. In that case the compiler should remove all but the last write operation.

[2] This technique is commonly referred to as *metaprogramming*.

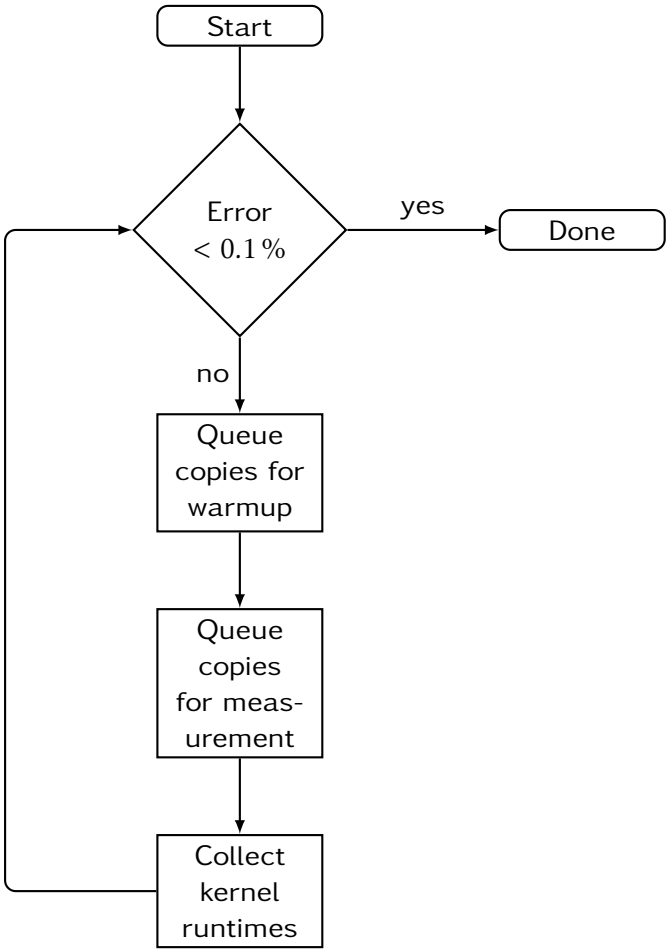

Figure 3.1.: Bandwidth measurements are repeated until the error of the average is below 1 %.

3.1.2. Data Type

The data type used has a major impact on the achievable bandwidth. Therefore, I benchmarked the bandwidth of each device for a variety of data types.

Note, that although in this section I will always use specific types like float and double, the same will also hold true for other types of the same size. In my benchmarks I was unable to observe any performance difference between a float2, a double, and a properly defined structure of two floats. However, as one usually uses types like float and double, I use the type names instead of always breaking them down to their size in bytes.

The bandwidth that can be achieved using a selection of data types for memory access on an AMD Radeon HD 5870 can be seen in Figure 3.2. The graph shows some small buffer effects until the size of the input and output buffers reaches 10 MiB. The effects are especially large for the float type. For larger buffers all types reach a plateau of bandwidth that can be achieved using this type to copy data from one buffer to another. On the AMD Radeon HD 5870 the best performance is achieved using the float4 data type, which matches documentation provided by AMD [69]. Note that float performance is about 20 GB/s worse than that of double and float4. The worst performance is achieved for data types larger than 16 B. Contrary to smaller types their performance decreases for larger buffers where the cache can no longer mitigate the inefficient memory access pattern they require.

I have repeated the above measurement on the last three generations of NVIDIA and AMD GPUs. In Figure 3.3 the bandwidth to copy buffers of 50 GiB is shown. This buffer size is chosen as it is well inside the plateau for all data types and GPUs.

The graph shows that large data types are a problem on every GPU. However, the best performing data type varies. On the NVIDIA GPUs all data types up to 16 B perform well, larger types drop to roughly half the performance. The older AMD GPUs show a much smaller performance difference between the data types. Still, types of 16 B size perform best, and in contrast to the NVIDIA GPUs types of 4 B size are about 20 GB/s slower than those of 8 B and 16 B size. The AMD Radeon HD 7970 shows the best performance. It clearly favours types of 4 B and 8 B size. Its performance drops by about one third for types of size 32 B and by more than half for types of 64 B size.

The graph also shows the development of the GPU architectures. The graph for the NVIDIA GeForce GTX 480 is nearly identical to that of the NVIDIA GeForce GTX 580, reflecting the fact that the latter is based on the same GPU architecture and only uses a smaller chip production process. AMD changed the architecture of the GPU when going from the AMD Radeon HD 5870 to the AMD Radeon HD 6970, however, the memory system was not effected by this. Moving to the AMD Radeon HD 7970 AMD performed a major architectural change, which is also reflected by the fact that this GPU has completely different characteristics with regards to memory performance.

Note that in this benchmark no GPU reaches its theoretical bandwidth limit. On the AMD Radeon HD 5870 only about 110 GB/s of the theoretical 154 GB/s are reached. That is an efficiency of about 70 %. The theoretical peak of the AMD Radeon HD 6970 is 176 GB/s, but only 125 GB/s are reached in the benchmark, showing approximately

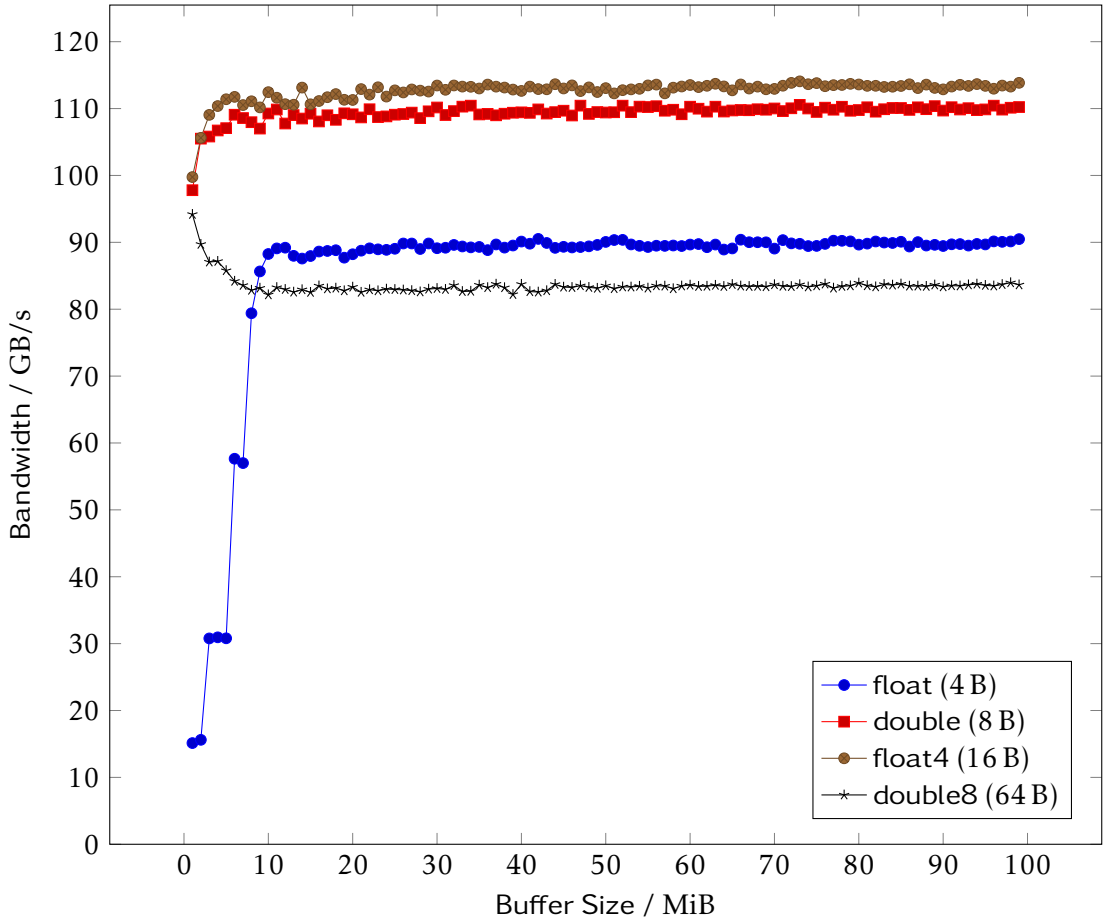

Figure 3.2.: Achieved bandwidth on an AMD Radeon HD 5870 when copying buffers of varying sizes for a selection of data types. The deviant shape of the double8 curve is caused by cache effects.

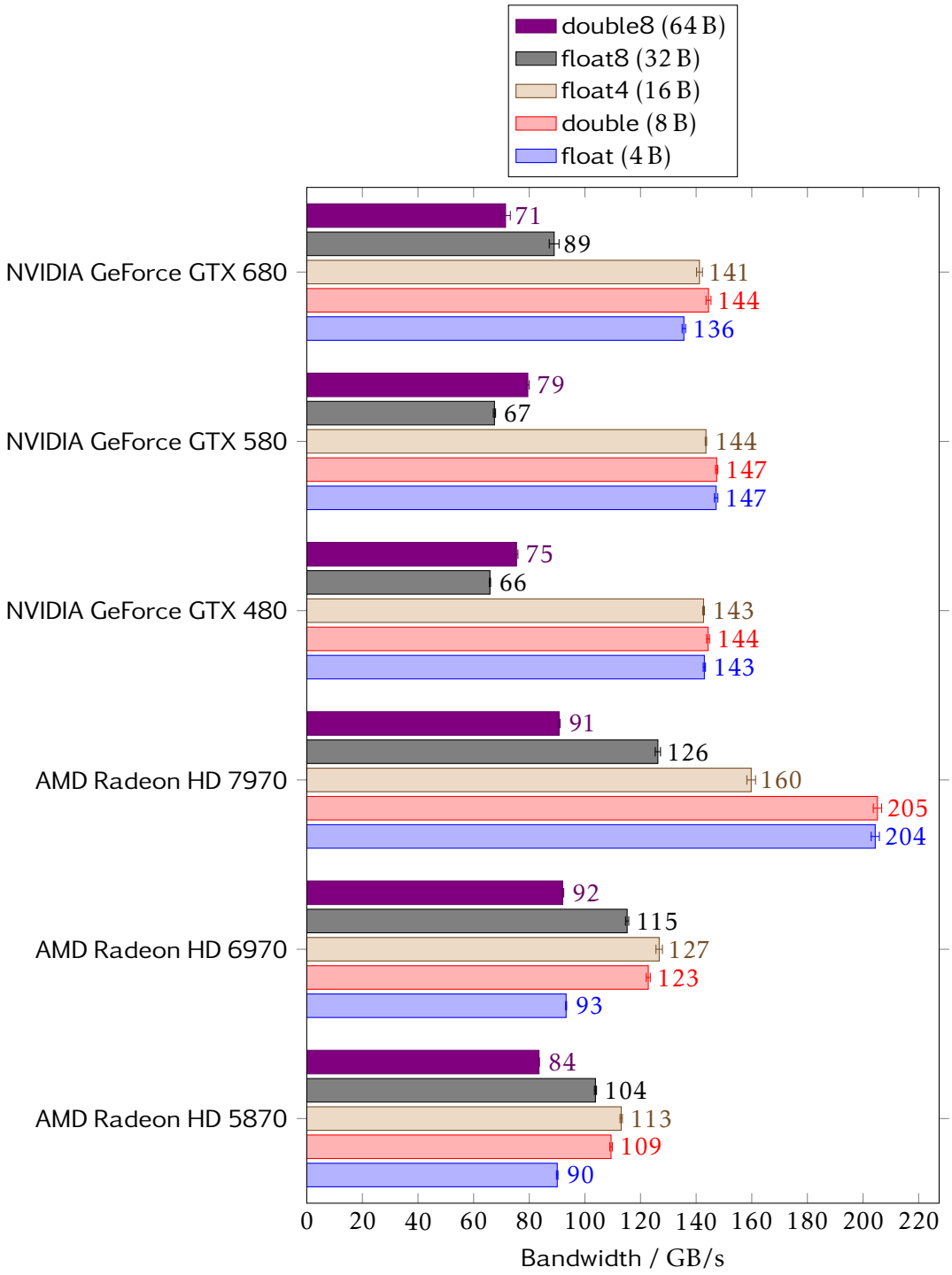

Figure 3.3.: Achieved bandwidth when copying buffers of 50 MiB size using a selection of data types on various GPUs.

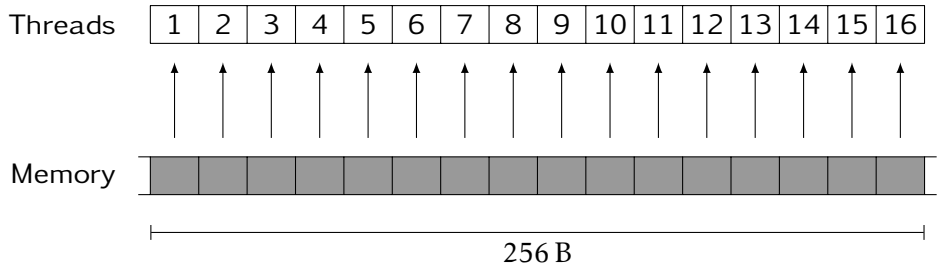

Figure 3.4.: The memory controller coalesces memory accesses by a group of neighbouring threads into actual memory requests. In the sketched example the group contains 16 threads. The data is stored as float4, such that each element has a size of 16 B. Thus, the threads, each requesting 16 B, together fully utilize a memory request of 256 B.

the same utilization. On the AMD Radeon HD 7970 utilization goes up to about 78 %, reaching 205 GB/s of the 264 GB/s theoretically available. On all of the NVIDIA GPUs a bandwidth of about 145 GB/s is reached. The theoretical peak of the NVIDIA GeForce GTX 480 is 177.4 GB/s, of which 82 % are utilized. In theory, the newer NVIDIA GPUs provide about 192 GB/s. But, that speed-up cannot be observed in the benchmark. Thus, utilization drops to about 75 %. Overall, it seems that in such a simple scenario only about three quarters of the memory bandwidth can be utilized on any GPU.

The varying performance of different data types is caused by the way the reads map onto actual memory accesses. In hardware there is a limited number of read and write commands. Usually each thread can explicitly write between four and sixteen bytes in one command. Larger types must be broken into multiple read and write commands. While this pretty obviously increases latency, it also reduces achievable bandwidth.

The memory controller coalesces memory operations of neighbouring threads. If the memory controller cannot map all requests of a group of threads into one access to memory the achieved bandwidth goes down. This is visualized in Figure 3.4. On NVIDIA GPUs the memory controller can typically issue memory accesses of 32 B, 64 B and 128 B [70]. AMD does not document the memory access size, but each memory channel has a width of 256 B [69]. In the example I use a hypothetical GPU with a lock-step size of 16 and a memory access size of 256 B.

If a type is larger than the largest memory request a thread can issue, the thread has to split its memory access into multiple commands. In this case, if neighbouring threads read neighbouring indices of the input buffer, there are holes in the memory accessed concurrently. Figure 3.5 shows this for a type of 32 B. In this case, each thread splits its operations into two reads of 16 B. When each thread performs its first read operation, those are offset by 32 B. The memory controller still coalesces the requests into common memory accesses. But, while requesting 256 B from the memory, only 128 B will be used. This way the net bandwidth achieved is only 50 % of the hardware's capabilities. Caching can mitigate the issue to some extent. However, in Section 3.3 I show why it cannot completely remove the issue.

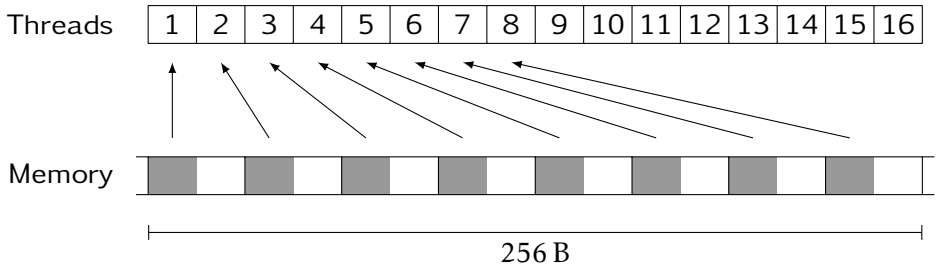

Figure 3.5.: The memory controller coalesces memory accesses by a group of neighbouring threads into actual memory requests. For types of 32 B the 16 B reads issued by the threads—shown in grey—leave holes—shown in white—in the actual memory request, wasting bandwidth.

On Cypress and Cayman GPUs there is an additional effect that explains why the float type performs worse than the float4 type. All of the internal memory paths have a width of 16 B. Also the native data type of the L1 cache has that size. Thus, if threads perform loads of 16 B, a better performance is reached.

3.1.3. Buffer Alignment

There are many situations where a kernel does not read the data of a buffer in the most naïve way. It is unusual that the first thread reads the first element of a buffer, the second thread reads the second element, and so on. A typical example are stencil kernels, where the second thread will also have to read the first element. Another example are situations where multiple datasets are stored in a single buffer. In that case the second dataset must be stored at some offset from the start of the buffer. The question is: What impact on performance does the choice of this offset have?

On early NVIDIA GPUs this effect was quite large. If the strict rules for coalescing were violated, an own memory transaction was performed for each thread [70]. Since then, NVIDIA changed the architecture of its memory controller to mitigate the problem. By giving the memory controller the possibility to read only 64 B from memory less bandwidth is wasted in case only a few bytes are required from a neighbouring memory segment. In addition, the introduction of caches allows to reuse read data of those segments in case it is required by a neighbouring group of threads. According to AMD the effect is small on their GPUs [69].

To verify the validity of the claims in the programming guides [69, 70], I have measured the effect of offset memory accesses on GPUs from AMD and NVIDIA. For this I used the same copy kernel as for the raw bandwidth measurement. As shown in Figure 3.6, every thread used a specified additional offset when accessing the elements. In the original case the offset into the buffer was only given by the thread's index. The offset was always specified in elements instead of bytes. While an offset specified in bytes allows more fine grained control, it can only be implemented by casting pointers, complicating the code for both the developer and the compiler. It also introduces ali-

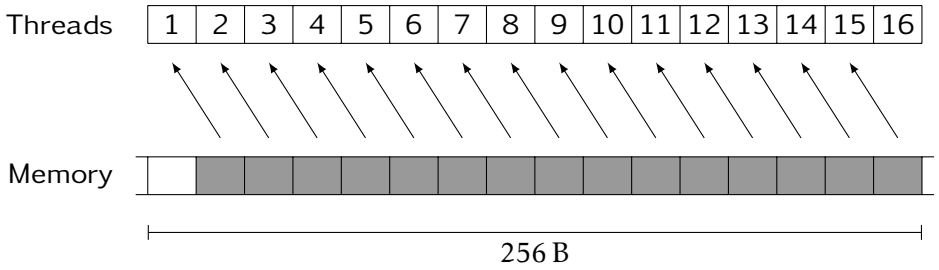

Figure 3.6.: Each thread accesses the buffer using a given offset in addition to its thread index. In this example an element is 16 B and the offset is one element. Thus, the last thread cannot be fed by the same 256 B memory transaction as the first 15 threads.

asing, which limits the optimizations the compiler can perform. Any desired offset can also be reached via offsetting whole elements, though it requires some more memory in certain cases. As before I used a buffer size of 50 MiB. Of course, the actual buffer allocated on the device was larger to accommodate for the offset into the buffer.

On all the GPUs tested the best performance is achieved when the offset is aligned to 256 B. As an example Figure 3.7 shows the bandwidth achieved using the double data type on the AMD Radeon HD 7970. For offsets aligned to 256 B the previously measured peak performance of about 205 GB/s is achieved. For other offsets, however, the performance can drop by more than 25 % to less than 150 GB/s. The behaviour of the other GPUs is the same. The NVIDIA GPUs level between 110 GB/s and 120 GB/s, while their performance for properly aligned offsets is close to 150 GB/s.

This can be understood by looking at the example sketched in Figure 3.6. The 16 threads of the group read 16 B each, thus requesting a total of 256 B. Without any offset this results in the situation sketched in Figure 3.4. There, the reads by all 16 threads can be coalesced into a single memory transaction. With the offset of a single element—equivalent to 16 B—the 256 B memory transaction would no longer be aligned to 256 B. This is, however, required. Thus, the 256 B memory transaction can only fulfil the request of the first 15 threads and 16 B of the requested data are discarded. To fulfil the request of the 16th thread, a second memory transaction is required. Assuming a minimum memory transaction size of 64 byte, this adds an overhead of 25 %. Caches can help mitigate this problem, potentially allowing the reuse of the otherwise discarded data by threads of other groups of lock-stepped threads. Still, as the results show, the effect is significant.

3.1.4. AoS versus SoA

Algorithms often rely on composite data types, represented by structures containing smaller types. However, in Section 3.1 I showed that these types do not allow to reach the maximum bandwidth if they exceed a device specific size, usually 16 B. On the one hand, it is usually possible to reformulate the algorithm to only use primitive types.

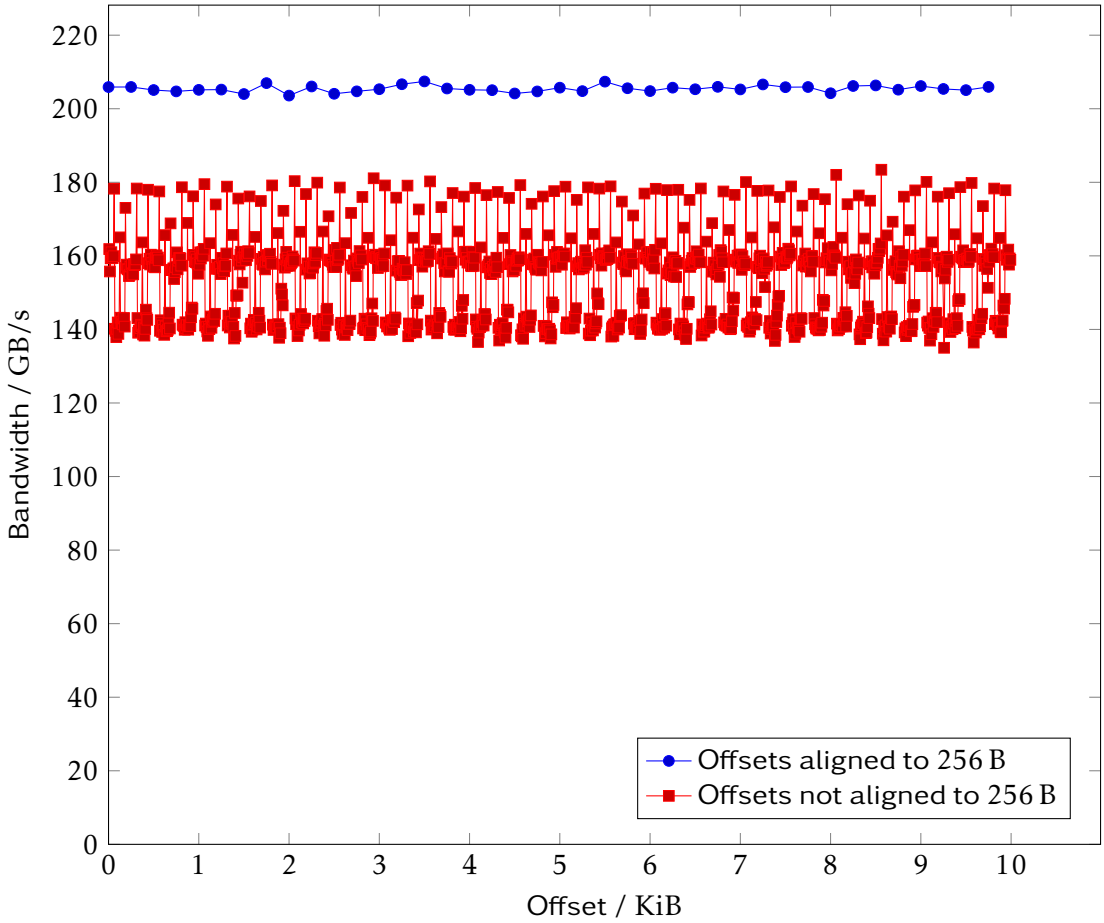

Figure 3.7.: Achieved bandwidth on an AMD Radeon HD 7970 when copying 50 MiB using the double (8 B) data type for different offsets into the buffer.

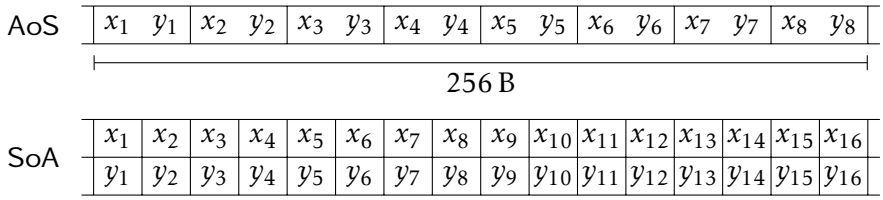

Figure 3.8.: An array of structure build from two scalars can be stored in two ways. In the AoS layout the two elements—x and y—of each structure are stored next to each other in consecutive memory. In the SoA approach the elements of the structure are stored independently. All x elements are stored in one array, all y elements in another.

On the other hand, this usually makes the code less readable and limits reuse of basic operations specified for the composite data type.

There is a third way: If implemented properly, the structure of arrays (SoA) approach provides both the convenience of the composite data types as well as the high performance of the primitive types. In the traditional approach—commonly called array of structures (AoS)—the composite structures implementing the composite data type are written directly to memory. This allows to read each entry by simply accessing the corresponding array element. By splitting the composite data type into its elements, storing each in an own array, an SoA layout is achieved. In the following I will refer to each of these arrays as *lanes*, as in C the term array is synonymous to the term buffer—which can lead to confusion. Figure 3.8 shows the difference in memory layout of the two approaches. This restricts memory accesses to data types yielding good performance. The memory accesses can be wrapped into dedicated functions which perform the transformation in between the structure and the memory representation of the type. This hides the additional complexity from the algorithm.

In theory it is also possible to always use the best-performing data type as the storage data type in an SoA storage layout. This does, however, require the use of some aliasing or passing the read data through a union. This can easily be hidden in the load and store functions implementing the SoA storage layout and, therefore, is not a problem concerning program readability. An example for a load function using this concept is shown in Listing 3.2. However, the compiler and optimizer will have to deal with the additional complexity. This might overcompensate the advantage of a slightly better performing data type.

There are two ways to implement the array part of an SoA memory layout. One is to use a separate buffer for each lane. The other is to use different areas of a single buffer. The latter is very easy to implement for a data structure containing N entries of the same type. A simple approach for M structures is to use an array of $M \cdot N$ elements for storage. The jth entry of the ith structure would then be stored in entry k of this buffer, which is calculated as follows:

$$k = i + j \cdot M. \tag{3.1}$$

Listing 3.2: An example of using float4 as the storage type for a structure containing only float elements.

```
typedef struct {
    float e1, e2, e3, e4, e5, e6, e7, e8;
} BigType;

typedef struct {
    float4 part1, part2;
} BigTypeHelper;

BigType loadBigType(
    float4 const * const restrict lane1,
    float4 const * const restrict lane2,
    size_t index) {

    BigTypeHelper tmp;
    tmp.part1 = lane1[index];
    tmp.part2 = lane2[index];

    return (BigType) tmp;
}
```

Of course, other patterns are possible. However, if neighbouring threads are supposed to read neighbouring structures, the layout should ensure to keep the structure index the innermost. Otherwise, accessing subsequent structures will not result in subsequent memory accesses.

When the pointers to device memory are exposed to the host side of the application and can be wrapped into an object, the approach of separate buffers works well. A major advantage of this approach is that each buffer should automatically fulfil all alignment restrictions. Therefore, optimal performance should be expected. In addition, memory size restrictions get stretched, as only part of the dataset has to fit into the maximum buffer size.

Contrary to NVIDIA CUDA, OpenCL does not easily allow this approach. In OpenCL all the buffers must be passed to a kernel as separate arguments. This becomes tedious and error-prone even for small numbers of buffers. In addition, many implementations of OpenCL impose a limit on the maximum number of kernel arguments. Still, this approach has been used successfully [55]. Yet, it cannot be applied in all situations.

The single buffer approach does not have these problems. In this case, it is always sufficient to pass the single buffer and the number of structures stored in it. Thereby, it is also easily possible to switch between an AoS and an SoA implementation. Also, the storage type used in the SoA implementation can easily be switched, as only the functions reading and writing to memory will be affected. Any function that only passes around the pointer to the buffer and the buffer size is not effected, assuming the data type used for the buffer has been properly defined. An additional advantage of the single buffer approach is that the hardware might only be able to read a limited number of buffers through its caches. This can cause additional performance losses if more than this number of buffers are used. Therefore, I usually prefer the single buffer approach.

3.1.5. SoA Stride

In Subsection 3.1.4 I introduced the SoA memory layout for storage of composite types. The offset between the lanes is called *stride*. Previously I have simply chosen the stride to be equivalent to the number of elements stored. This is the minimal stride that can be used. Otherwise the lanes would overlap. However, in Subsection 3.1.3 I also showed that not all offsets into a buffer are able to provide optimal performance. Therefore, I also investigate the performance impact of using different strides.

Figure 3.9 shows the performance that can be achieved on an AMD Radeon HD 5870 for buffers of different size using an SoA memory layout for a composite type built from two float4. The float4 type is the fastest type available on the AMD Radeon HD 5870. It achieves more than 110 GB/s when copying buffers using scalars of that type to access memory. The SoA layout using the naïve stride introduced before reaches only about 70 GB/s. This means a loss in performance of about 30 %.

Figure 3.9 also shows the performance reached using separate buffers for each lane. In this case, this means two buffers for the storage of the data to be copied and two buffers to write the data to. The graph shows that for some buffer sizes this provides close

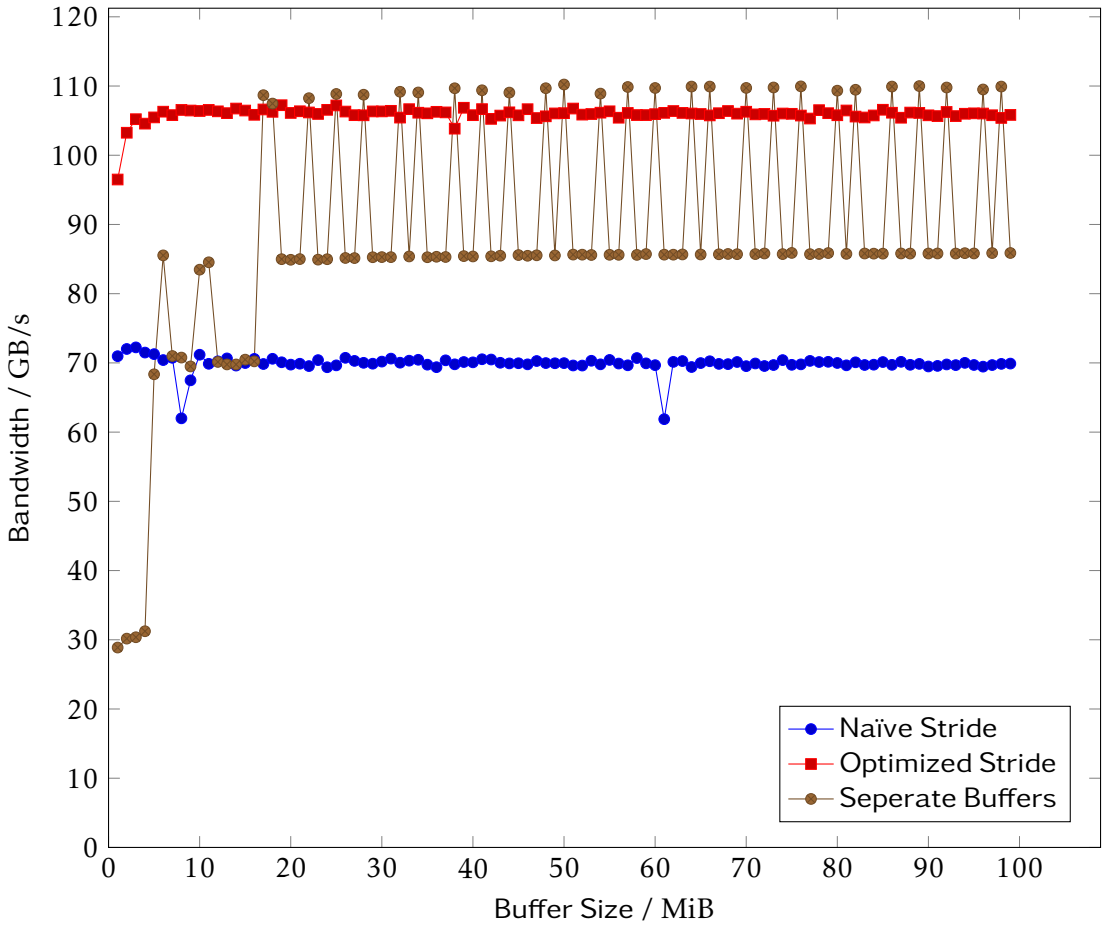

Figure 3.9.: Copy performance using SoA for structures of two float4 (16 B) on the AMD Radeon HD 5870. In the case of separate buffers the stride is chosen by the GPU driver.

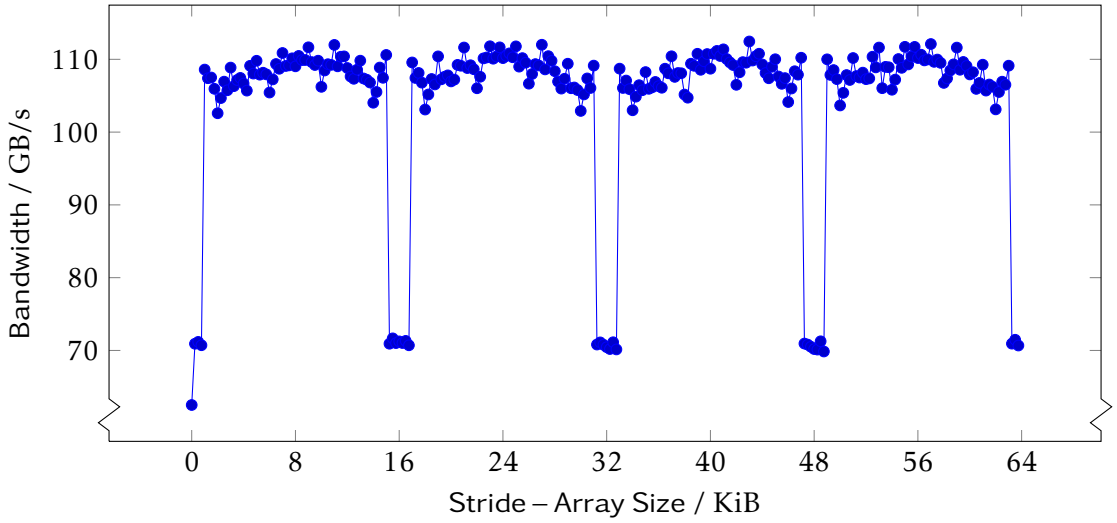

Figure 3.10.: Performance using SoA for a structure of two float4 (16 B) to copy 50 MiB of data on the AMD Radeon HD 5870.

to peak performance. However, for other buffer sizes the performance drops below 90 GB/s. This is a performance loss of more than 20 %.

Obviously the stride between two buffers allocated by the GPU driver is not always optimal. This also effects kernels like the addition of float vectors which are not using SoA memory accesses. That kernel has the same memory access pattern—each threads reads corresponding elements from each buffer—and will suffer from reduced performance on certain buffer sizes.

The variant using optimized strides reaches about 105 GB/s for all buffer sizes, except for small buffer effects. But, before I present how the stride was optimized, I want to show how different strides effect the performance.

Figure 3.10 shows the effect of varying the stride for structures of two float4 on an AMD Radeon HD 5870. The measurement is performed by copying 50 MiB of data. Based on the results of Subsection 3.1.3 all lanes should be aligned to 256 B. Thus, only strides that are a multiple of 256 B are used. Each of the two lanes stores 16 B of each element. The graph shows a clear structure with dips in performance every 16 KiB, which is equivalent to 1024 elements. Centred on strides that are a multiple of 16 KiB they range 768 B in each direction. While most strides provide more than 100 GB/s of bandwidth, in the dips performance drops to about 70 GB/s. This means: More than 30 % of the performance is lost.

To investigate how this translates to structures with more elements, Figure 3.11 shows the same measurement. However, this time structures of four float4 are used. Overall performance is slightly lower this time. Again, major dips every 16 KiB can be observed. In addition, there are now smaller dips in between the dips at 16 KiB. Of these, the dips located at an offset of 8 KiB from the major dips are the largest.

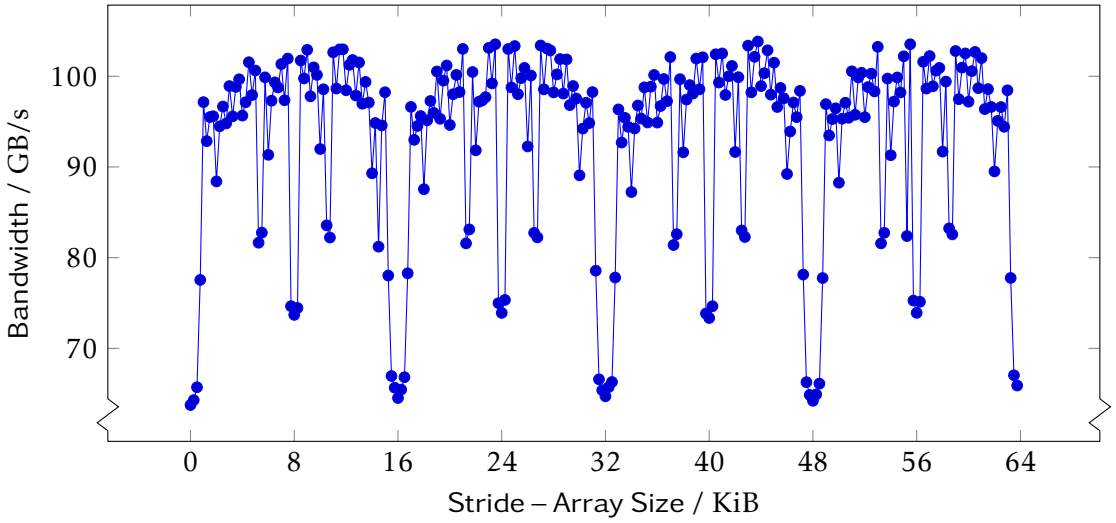

Figure 3.11.: Performance using SoA for a structure of four float4 (16 B) to copy 50 MiB of data on the AMD Radeon HD 5870.

Measurements using structures of other numbers of entries show similar patterns, always including the major dips at 16 KiB. When using three float4 per structure only the dips at 8 KiB offset are added. Structures of eight float4 show a complex pattern including five clearly visible additional dips. With structures of sixteen float4 it becomes difficult to recognise any pattern but the major dips because there is no clear plateau of well performing strides any more.

Up till now the term stride always implied the offset between two consecutive lanes used for storing the structure. However, when storing a structure with more elements there are additional strides that occur. The offset between the first and the third lane also defines a stride. When using a structure of four elements there are three different strides given by the possible combinations of the four lanes. This assumes that the offset between consecutive lanes is constant, as it is the case in the measurements and analysis performed here. In that case the stride between the first and the third lane is the same as that between the second and the fourth. This simplifies the problem to the strides between the first lane and all other lanes. In general for a structure of N entries there are $(N-1)$ strides. Naming the stride between consecutive lanes $\text{stride}_{\text{base}}$, those can be expressed via the following formula:

$$\text{stride}_n = n \cdot \text{stride}_{\text{base}}. \tag{3.2}$$

The additional strides explain the additional dips in Figure 3.11. Whenever one of the strides is a multiple of 16 KiB—or within 768 B of such a multiple—performance drops. The different levels of the performance drops can be explained by how many of the combinations of the lanes are in the range of such a multiple. If the base stride is a multiple of 16 KiB all combinations of lanes are effected. In the case that only

$stride_2$ is effected; only the combinations between the first and third, as well as between the second and forth are effected. Thus, the performance loss is less. Finally, if only $stride_3$ is effected, only the combination of the first and fourth lane is effected and the performance loss is even smaller.

Figure 3.12 outlines an algorithm that can be used to find a base stride that is not effected by the drops. It repeatedly increments the stride by one element until it fulfils all requirements—all lanes are aligned to 256 B and no two lanes have a stride of 16 KiB in between them. While there might be edge-cases where no such stride can be found, I have not observed such a case so far. Using this algorithm results in the optimized curve shown in Figure 3.9, which only shows a minimal loss of about 10 GB/s versus the performance using plain float4 shown in Figure 3.2.

The AMD Radeon HD 6970 shows the same behaviour. However, as shown in Figure 3.13, the frequency of the dips halves. Only strides that are within 768 B of being a multiple of 32 KiB drop in performance. Therefore, the stride optimization targeted at the AMD Radeon HD 5870 works fine for this GPU, too. It improves the SoA performance for structures of two float4 from less than 90 GB/s to more than 120 GB/s for all buffer sizes.

The interesting bit about this is that this effect is caused by the stride between subsequent reads performed by a single thread. Other bandwidth optimizations are usually required due to the interaction of reads by multiple threads.

One potential cause for the performance impact could be the organization of the L1 cache of the GPUs. For both the AMD Radeon HD 5870 and the AMD Radeon HD 6970 that cache has a size of 8 KiB [69]. Assuming an associativity of two, reads with a stride that is a multiple of 16 KiB would hit the same cache line. In that case the second memory request would have to wait until the first request has been serviced while a request using a different cache line could overlap with the first. This would explain the observed performance reduction. The doubling to 32 KiB on the AMD Radeon HD 6970 could be explained by a doubled cache associativity.

On the AMD Radeon HD 7970 and on the GPUs by NVIDIA, no similar behaviour is observed. Therefore, on those GPUs only the alignment of the lanes to 256 B must be taken into account. This simplifies the SoA stride optimization algorithm by leaving out the second decision.

Using an SoA memory layout the AMD Radeon HD 7970 is limited to about 160 GB/s when using the double data type, which is far below the maximum bandwidth that can be achieved on this GPU. Other types show similar performance. The NVIDIA GPUs show about 140 GB/s, which is at the level of their peak performance.

Using the given optimizations, optimum performance can be reached for any data size using only a single buffer to store all lanes of an SoA data layout. Given that this—as shown in Subsection 3.1.4—is also the better approach in terms of program readability and error avoidance, I recommend to always use this approach.

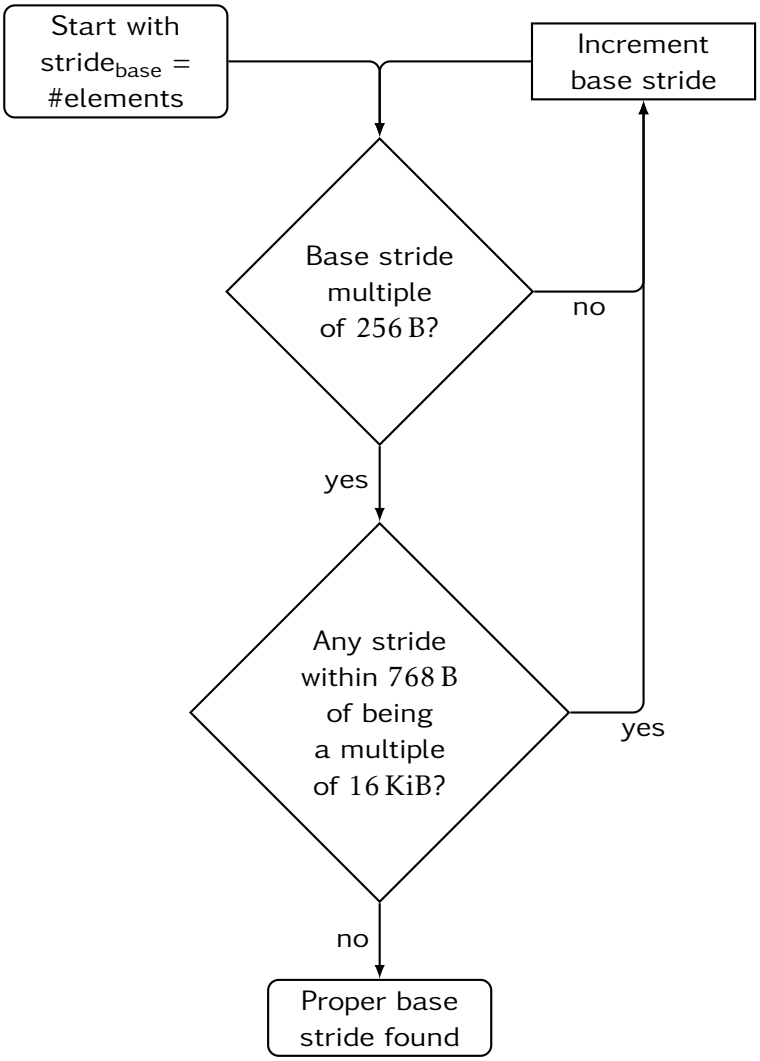

Figure 3.12.: An algorithm to find proper SoA strides.

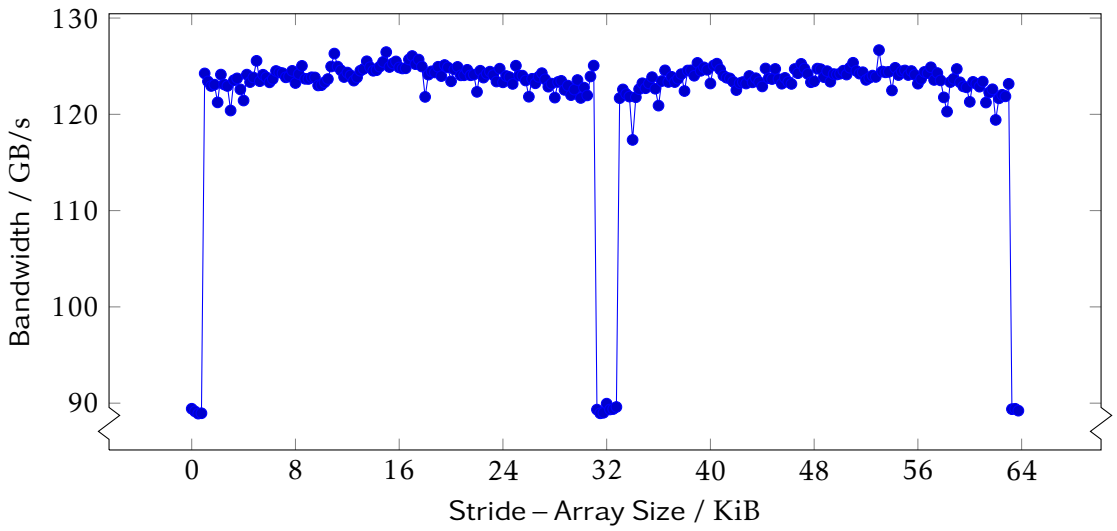

Figure 3.13.: Performance using SoA for a structure of two float4 (16 B) to copy 50 MiB of data on the AMD Radeon HD 6970.

3.1.6. ECC

Contrary to traditional servers, GPUs historically do not have ECC RAM. In their traditional use case of rendering graphics it simply does not matter if a pixel shows a wrong value for a single frame. Even in the less likely case that some data that is used in more than one frame is effected by a memory error, the cost of restarting an application is much lower than the additional cost of ECC RAM. Especially, as there are no problems in case they go undetected. And in case they are detected they are only a visual annoyance.

With the appearance of GPGPU an interest in ECC memory on GPUs came up, as they are now used in areas where memory errors might potentially cause much higher cost. For example, GPUs are now used to process medical data [71]. NVIDIA presented the first ECC enabled GPU in 2011 when it introduced the Fermi architecture. It now offers ECC capabilities for both its NVIDIA Quadro series targeted at workstation graphics and its NVIDIA Tesla series meant as pure compute cards. AMD followed in 2012 by adding ECC capabilities to its AMD FirePro GPUs based on the Tahiti architecture. On all of these GPUs the available memory bandwidth decreases if ECC is enabled as parts of GPU memory and buses are used to store and transfer the ECC data.

ECC only protects from a very special kind of error: that in which the memory does not return the same value as previously written to the read address. Therefore, one can always argue whether other error detection schemes, that verify the result of a computation, are not suited better. For example, when solving a system of linear equations the solution can easily be verified by inserting it back into the system of equations. This has the advantage of also being able to detect implementation errors. However, the

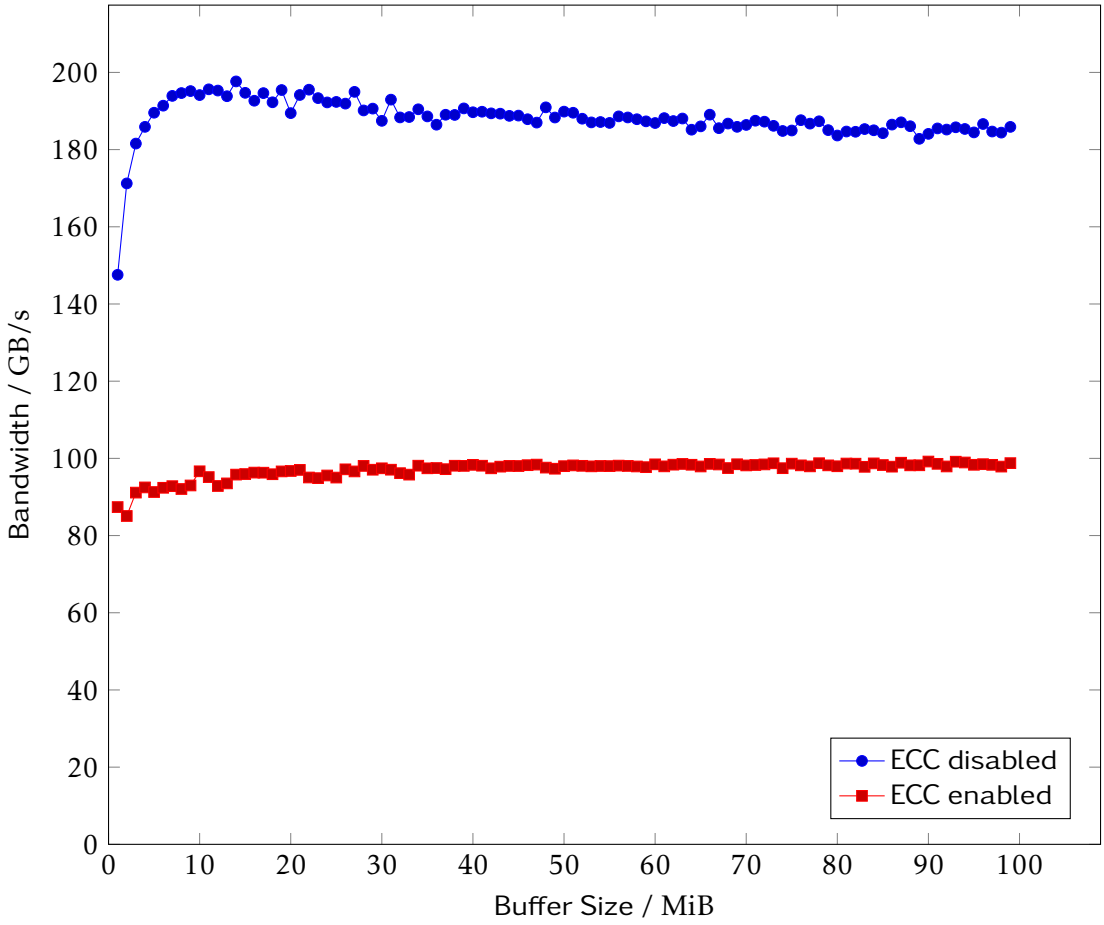

Figure 3.14.: Copy performance using the double (8 B) data type on one GPU of an AMD FirePro S10000 with and without ECC enabled.

feasibility of such a solution depends on the problem to be solved. In addition, it seems that memory errors are dominated by hard errors [72]. These errors, which indicate a broken memory chip, could also be found by frequent memory tests. Here, I will focus on the performance implications that the use of ECC has on the bandwidth available to applications.

As before, I measured the performance achieved when copying buffers of varying size. Figure 3.14 shows the performance when using the double data type with and without ECC enabled. Usage of ECC limits the performance to less than 100 GB/s in contrast to 180 GB/s to 200 GB/s, which are possible without ECC. This means nearly half the performance is sacrificed.

ECC has similar effects on larger data types with SoA data layout. Figure 3.15 compares the performance with and without ECC enabled to copy buffers of varying size using an SoA layout for a structure of two entries of type double. Again the GPU is

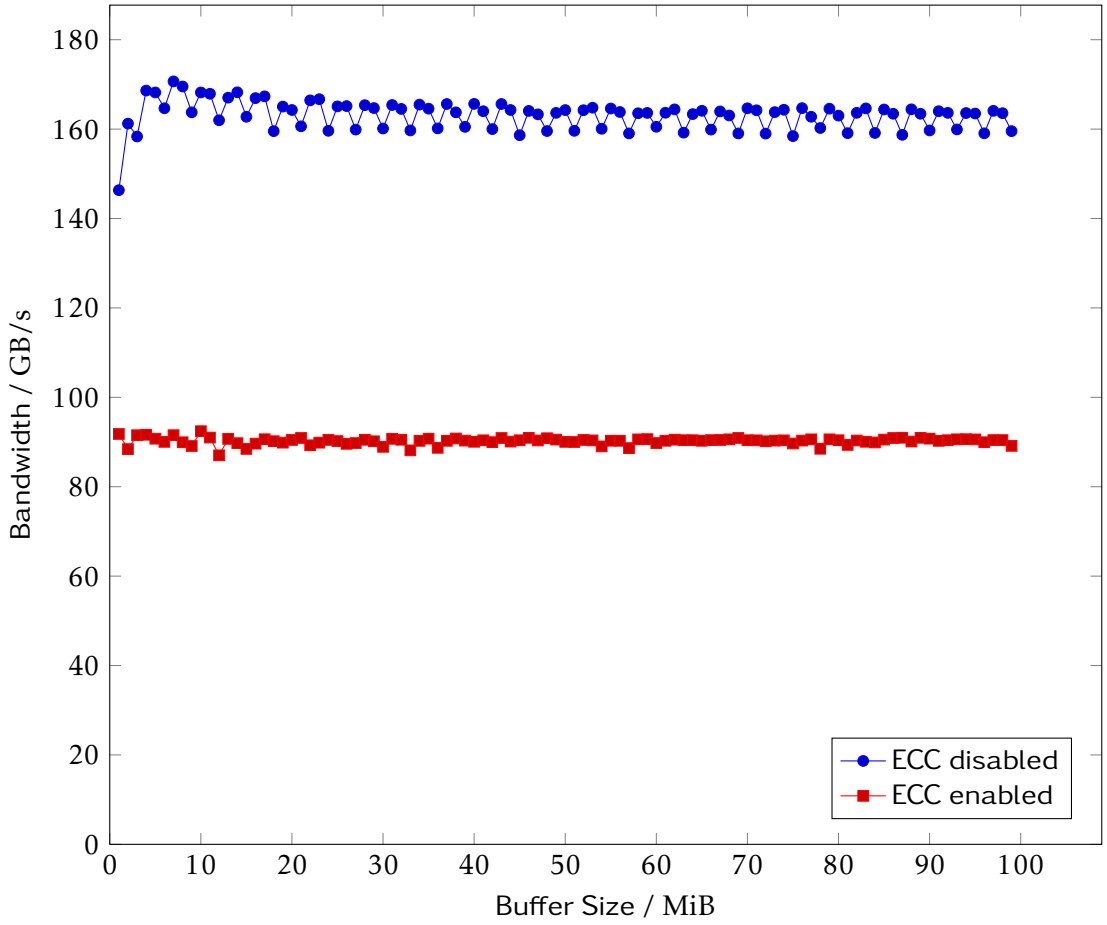

Figure 3.15.: Copy performance using a structure for two entries of type double (8 B) on one GPU of an AMD FirePro S10000 with and without ECC enabled. The data was stored in an SoA layout with optimized strides.

limited to less than 100 GB/s if ECC is enabled. However, the loss versus not using ECC is less in this case. Obviously on this low level of performance SoA does no longer loose as much performance versus the plain types. Still, nearly half the performance is lost versus not using ECC.

Overall ECC comes with a major performance penalty. The loss of nearly half the available bandwidth suggests to not use ECC for bandwidth limited codes. In this case, I clearly recommend to use other verification schemes to ensure correct results.

This is different for codes which are not bandwidth limited. An extreme example of this is Bitcoin mining [73]. The Bitcoin miner Phoenix [74] is compute bound and hardly utilizes the memory at all. I sampled the performance of version 2.0.0 of that application on the AMD FirePro S10000 for one hour. The miner was connected to a

Bitcoin client running on the same system.[3] With ECC enabled, 464.413(15) MHash/s are achieved. Without ECC enabled, it achieves 464.591(42) MHash/s. This shows that the performance impact of ECC can be neglected if the application is sufficiently compute bound.

3.1.7. Conclusion

The bandwidth that can be achieved depends strongly on the data types and memory layouts used. The choice of the wrong data type or of the wrong memory layout can easily cost 50 % of performance. In addition, care must be taken when performing memory accesses with an offset into the buffer. While the performance loss in these cases has reduced on current hardware, it can still cost a quarter of the possible performance.

As it can be seen in Figure 3.3, types with a size of 8 B, like double or float2, provide best performance on all GPUs. If there is no need for high performance on older GPUs by AMD, types of size 4 B, like float are also a good choice.

SoA data layouts allow to reach high performance for data types larger than 16 B. However, on Tahiti based GPUs like the AMD Radeon HD 7970 and the AMD FirePro S10000, bandwidth is still limited to about 160 GB/s instead of the 200 GB/s that can be achieved using small, scalar types like float and double. On older AMD GPUs naïve SoA layouts can cost more than 30 GB/s of performance, which is more than 25 % of the achievable bandwidth. I have developed an algorithm that generates a proper data layout for these GPUs. On all other GPUs it is sufficient to ensure that each lane of the SoA layout is aligned to 256 B.

Finally, on current GPUs ECC costs nearly half the available bandwidth. Therefore, the use of ECC should be avoided for bandwidth limited applications.

3.2. Registers

GPUs have a register file of fixed size. From this they dynamically allocate registers for threads. The exact number of registers allocated per thread depends on the kernel used and is defined at compile time—of course within limits imposed by the GPU architecture. This is a major difference to CPUs, where the number of registers available to each thread is fixed by the CPU architecture.

Register file size and register requirements of the GPU kernel define the number of threads that can be run concurrently. Lightweight threads, that use only a small number of registers, allow to keep a maximum number of threads in flight on the hardware. This helps to hide the latencies incurred by memory accesses. If one group of threads issues a memory access, the hardware will simply continue with a different group of threads on the next cycle. As memory accesses have latencies in the order of a thousand cycles, many logical and arithmetical operations can be performed until the memory operation completes.

[3] The client was using the Bitcoin test network. Thus, it did not take part in the global Bitcoin economy.

Using a larger number of registers limits the number of threads that can be in flight at the same time. This does, however, not necessarily have to be an issue if memory latency is not a limiting factor for the application. Using more registers can reduce the number of memory accesses. This is useful in bandwidth limited codes as it reduces the bandwidth requirements.

In case the hardware cannot provide as many registers as the calculations require, registers will be swapped out into the private memory of the GPU. This has two major disadvantages: Access to these swapped-out registers has the same latencies as normal memory access. In addition, writing and reading these registers increases the overall bandwidth requirements of the application. In an application that is not bandwidth limited, the latter is not an issue and the former might be outweighed by other advantages as long as the scratch registers are not accessed too often or remain in cache. This can be observed in the GPU based tracker for the time projection chamber (TPC) of A Large Ion Collider Experiment (ALICE) [13, 75]. However, in a bandwidth limited application the additional bandwidth requirement will deteriorate application performance. Therefore, bandwidth limited codes must always make sure to not swap out registers into memory.

I have measured the effect of different register usage quantities on performance. It is not easily possible to create simple copy kernels with one specific register requirement. However, allocating a specific amount of local memory for each group of threads has the same limiting effect on the maximum number of executed threads as register utilization has. Therefore, I used this to emulate the effect that using more registers would have on the achievable bandwidth.

Figure 3.16 shows the bandwidth for copying 50 MiB of data using the float4 data type using different amounts of local memory on the AMD Radeon HD 5870. As a result the amount of threads running concurrently on each CU is limited. The graph shows that to achieve high performance using the plain float4 type at least 192 threads are required. Using a structure of two float4, even only 64 threads reach good performance. Thus, this minimal number of threads per CU is enough for a bandwidth limited code as long as enough independent memory operations are performed by each thread. Therefore, on the AMD Radeon HD 5870 bandwidth limited codes should be able to use up to 256 registers per thread without a negative impact on performance. Factually, the maximum number of registers per thread seems to be limited to 124, then allowing the use of 128 threads.

Figure 3.17 shows how the number of threads per CU effects performance on the AMD Radeon HD 7970. The graph shows a similar structure as that for the AMD Radeon HD 5870, but on the AMD Radeon HD 7970 a larger number of independent memory accesses is required to be able to saturate the memory bandwidth. Also note that this measurement uses the double data type as it is the best performing type on the AMD Radeon HD 7970. Therefore, only half as many bytes are transferred per memory access. Still, starting from only 256 threads good performance is reached. This means, each thread can use up to 256 registers without performance losses. A larger number of used registers should be possible if each thread performs a sufficient number of independent memory accesses.

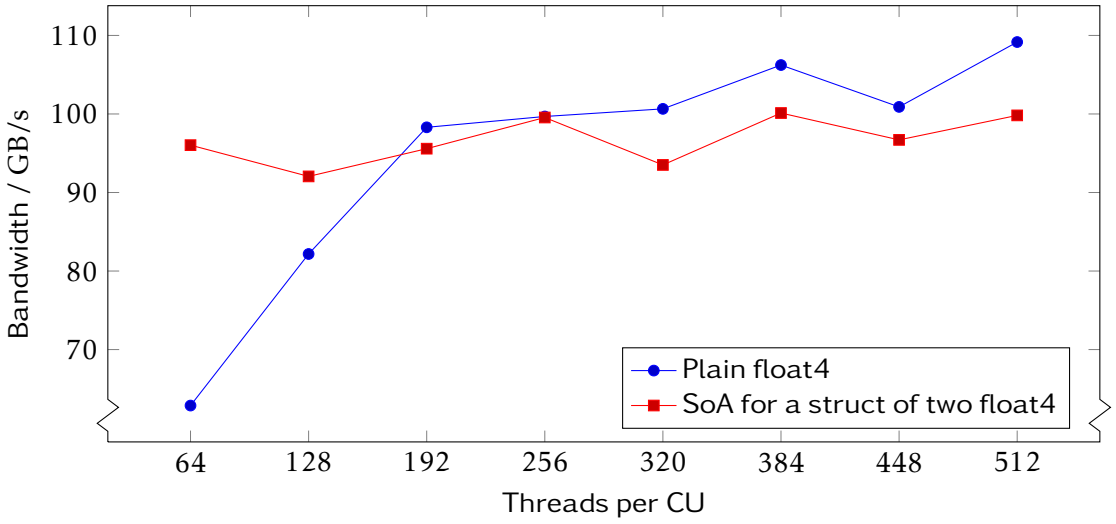

Figure 3.16.: Copy performance for 50 MiB of data with different numbers of threads per CU on an AMD Radeon HD 5870.

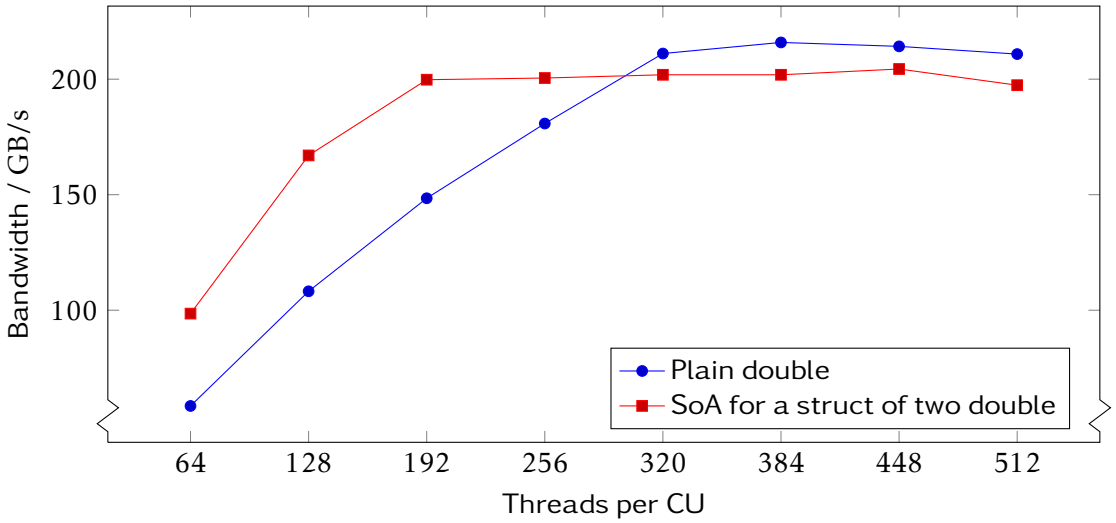

Figure 3.17.: Copy performance for 50 MiB of data with different numbers of threads per CU on an AMD Radeon HD 7970.

The bandwidth of a GPU can be saturated without running the maximum number of threads a GPU can run. Therefore, in bandwidth limited codes it is not essential to reduce the registers used to an absolute minimum. The additional margin can be used to keep data in registers, minimizing the bandwidth requirements of the application. However, care must be taken to not exceed the register file's resources, as swapping in and out registers will cause additional use of the limited bandwidth available.

This also opens up room for ND-Range optimizations. As only few threads local to the CU are required to saturate memory bandwidth, it is sufficient to queue a minimum number of threads per CU. This allows to use large working sets. If a kernel only uses few registers and performs only few independent memory operations per thread, a large number of threads should be queued on each CU to achieve optimum bandwidth.

To allow the developer to monitor the number of registers used by a kernel, the compilers can report this number. This is especially important to verify that a kernel does not use scratch registers. The AMD compiler prints this number to intermediate files during the compilation process. To ease the analysis of the kernel register usage I have created the application *pyclKernelAnalyzer* [76]. It compiles a file containing OpenCL code, parses the intermediate files, and reports the register usage.

Experience shows that the register usage of kernels increases with kernel complexity, which can be caused by loops, conditional execution, or simply kernel length. While this does not seem too surprising, the compilers often do not properly reuse registers, even if this leads to register spilling. Thus, it is usually a good idea, to avoid kernels growing too long.

Utilizing more registers is not necessarily a bad thing, as it increases the possibilities for ILP. This improves shader utilization and thus overall performance. When aiming for the best possible performance, it makes sense to perform as much work in a single kernel as possible. However, one must carefully check the register usage to avoid falling victim of degraded performance due to the usage of scratch registers.

3.3. Cache Usage

The purpose of a cache differs in between a CPU and a GPU, and even in between the different GPUs. On a CPU, a cache's purpose is to reduce the latencies a thread experiences when accessing main memory. It speeds up multiple subsequent accesses to nearby memory areas. Its caching is based on time locality of memory accesses in the current thread.

The cache of a GPU does not necessarily provide low latency access [77]. Due to the large number of threads, optimizing temporally local accesses by a single thread is not that important. If a GPU runs 1024 threads on a single CU and each thread reads 16 B, a total of 16 KiB is read. This will completely fill the L1 cache of an AMD Radeon HD 7970 [69]. If each thread should process more than 16 B of continuous data, the threads would constantly thrash out each others cache lines.

The purpose of a cache on the GPU is to enable coalesced access to memory if the accesses of neighbouring threads do not fulfil the strict rules for direct coalescing. In

a simple stencil algorithm lots of misaligned reads are required. In Subsection 3.1.3 I showed why this reduces the memory throughput. In such a scenario, the cache can ensure that all memory accesses are of the maximum size. Ideally, afterwards all groups queued on this CU unit can utilize the data stored in the cache. Such, memory accesses that would otherwise have only provided a few bytes of data for a single thread of a single group now are used by more threads, potentially part of multiple groups. As a result, the ratio of bytes used versus bytes read from memory increases. This requires that—instead of a temporal locality of memory requests in a single thread—the algorithm has a spatial locality of memory requests of multiple threads. Therefore, algorithms need to be structured differently on the GPU than on the CPU to optimally utilize the caches [78].

On modern CPUs, features like hyper-threading introduce some of the GPU cache characteristics. The two threads that in case of hyper-threading can be executed on the same core of an Intel CPU must share a common cache. Therefore, in this case, it is of advantage if these two threads access the same cache lines in an interleaved fashion [41]. This is the same pattern that is ideally used on a GPU. The difference is that on the CPU the memory accesses are coalesced by the temporal locality of the two threads accessing the same cache line, which the cache always fully fetches from memory. On the GPU the memory requests actually occur at the same time by all threads. Ideally, in this case, the memory controller coalesces those requests even if the cache is not involved.

Historically caches on GPUs have only been used to access textures. Therefore, there are some restrictions how they can be utilized. OpenCL has a special buffer type called *image* which exposes this capability. The major drawback is that, instead of normal array notation, special access functions must be used to access data in these buffers from a kernel. The current generation of GPUs from both NVIDIA and AMD do provide generic caches that are utilized when accessing normal buffers as well [69, 79].

For GPUs of the Cypress and Cayman generation, AMD provides an intermediate solution. If a normal buffer is only accessed via a non-aliased pointer to a constant type, the compiler will automatically create the instructions to read the data through the cache. The `restrict` keyword of C99 is used to specify that the pointer is not aliased. This allows to use the caches of those GPUs with normal OpenCL buffers in many cases.

For the simple buffer copying benchmark used in this chapter, the pointers to the buffers were always used with all possible qualifiers applied. This means that the compiler was free to utilize the cache on the AMD Radeon HD 5870 and AMD Radeon HD 6970 to speed up reading the input data. I removed the qualifiers and performed an additional benchmark on the AMD Radeon HD 5870 for the float4 data type and using SoA for a structure with two entries of this type. The data shows no difference from the performance using the cache. This is of little surprise, as the memory accesses in this simple case are already optimally structured.

The cache of the GPU can also mitigate register spilling. If spilled registers are cached—instead of accesses reaching the memory—this will reduce the impact of spilled registers on the achievable bandwidth.

The scratch-pad memory local to each CU can usually be accessed with low latency [69, 77, 79]. As it is shared by all threads of a group it can be used as an explicitly managed cache. This can be especially useful if values need to be accessed often but cannot be stored in registers. Another use case is if multiple threads in a group need to access the same data. In this case, the data can be loaded into the scratch-pad memory in an optimal way and afterwards used with high performance.

On a GPU it is possible to achieve full memory bandwidth without any cache utilization. However, this does require the proper memory access patterns. Algorithms that cannot fulfil the restrictions of these patterns, as it occurs for stencil based ones, can profit from caching. For this, they need to be written such that there is a spatial locality in the memory accesses. The local scratch-pad memory in each CU can be used as an explicitly managed cache, providing a valuable resource for optimization.

3.4. Communication

The previous sections focused on speeding up the performance of calculations on the GPU. In this section I evaluate the performance of communication via PCIe, which is the primary interconnect of current GPUs. This communication is important, as before any calculation is performed data will have to be transferred to the GPU and any result will have to be transferred back the host afterwards.

While the initial and final transfer can be negligible for lengthy calculations, many algorithms also require intermediate results to be transferred to the CPU. In this case, communication latency can be more important than data rate. An example of such algorithms are iterative inverters. The impact of communication latency on such algorithms is shown in Subsection 4.3.3.

Another situation where intermediate communication often cannot be avoided is when utilizing multiple devices. This is often desirable due to the larger aggregate memory bandwidth provided by multiple devices and because of the larger aggregate memory size. In this case, data must often be transferred in between devices, instead of between host and device.

To analyse the performance of different communication techniques, I have developed *clPCI* [80]. This collection of Python scripts handles benchmark execution and data correctness verification. All the benchmarks in this section have been performed on SANAM. The measurements have all been performed over 1000 consecutive transfers to keep the errors below the percentage limit.

3.4.1. Communication between Host and Device

Figure 3.18 shows the performance reached when transferring data from one GPU on the AMD FirePro S10000 to the memory of the CPU. When using plain memory, which has been allocated via a usual `malloc`, only about 5 GB/s are reached. For pinned memory a much higher data rate of more than 12 GB/s is reached. Pinned memory is page-locked and registered with the GPU, such that the GPU can directly write to

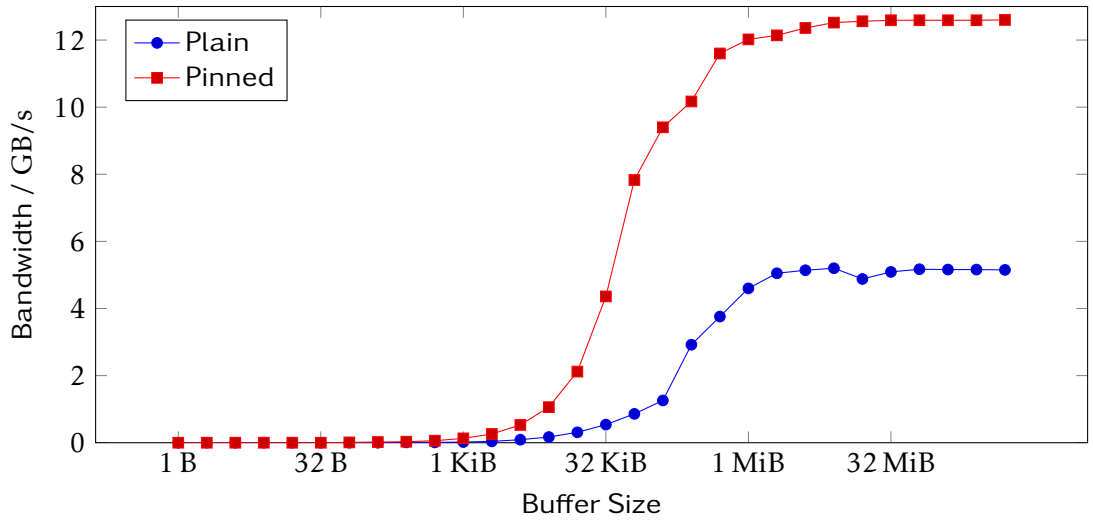

Figure 3.18.: Utilized Bandwidth when copying data from GPU to CPU memory in a node of SANAM

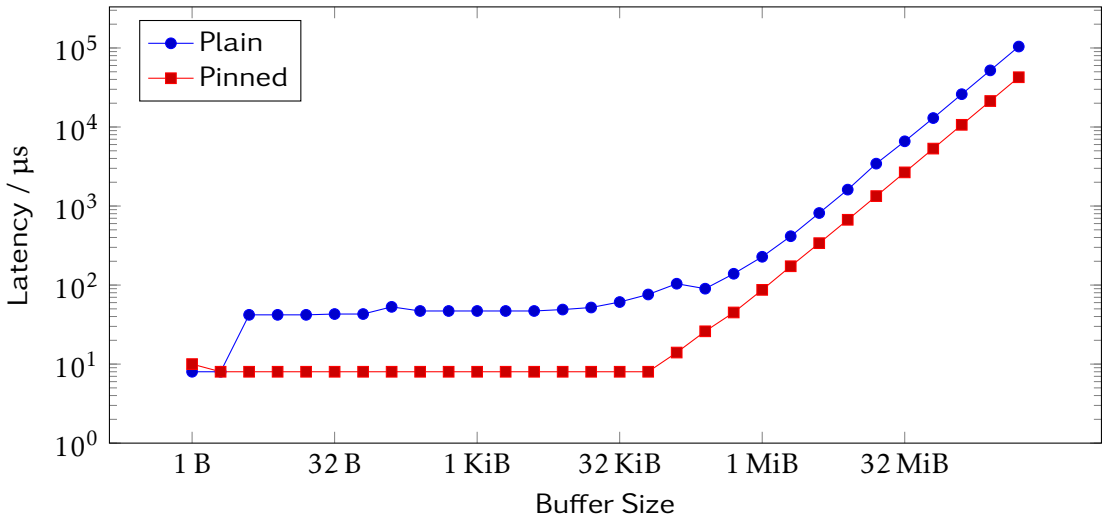

Figure 3.19.: Latency when copying data from GPU to CPU memory in a node of SANAM

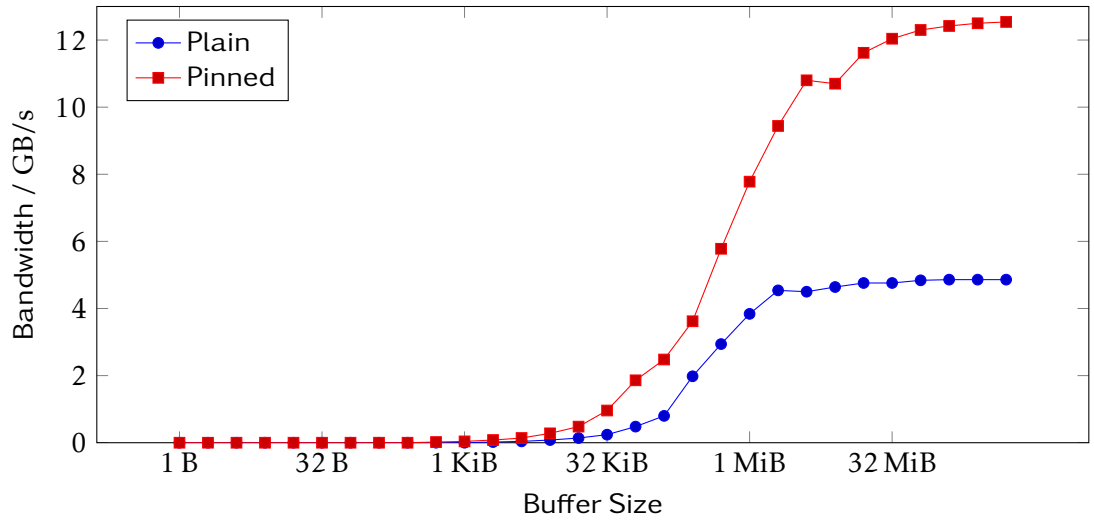

Figure 3.20.: Utilized bandwidth transferring data from GPU to CPU memory and back in a node of SANAM

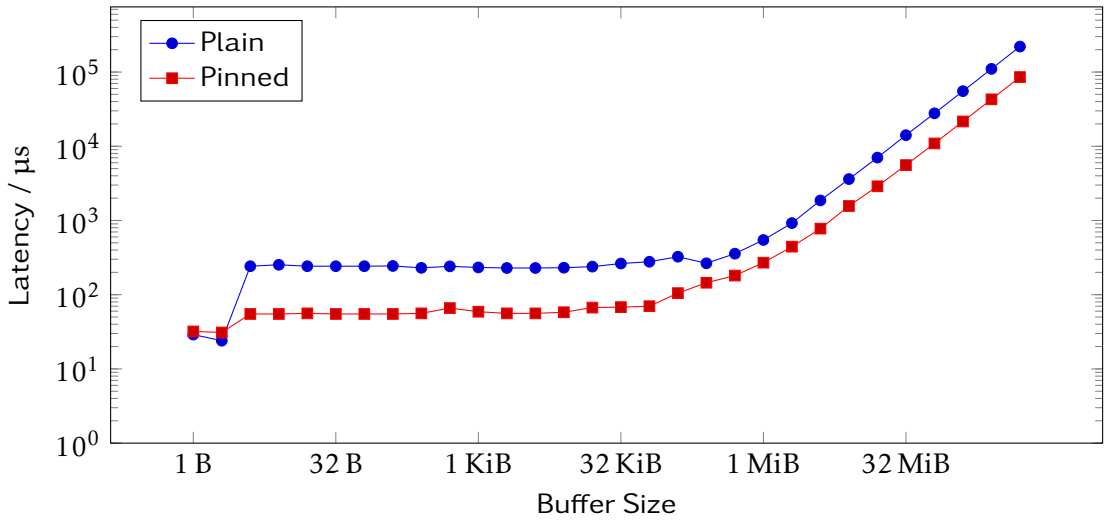

Figure 3.21.: Latency transferring data from GPU to CPU memory and back in a node of SANAM

the destination memory. This does not work for normal memory, as it is not necessarily even located in physical memory. For normal memory the GPU will send the data to a buffer allocated by the driver. The data is then copied into the destination memory by the CPU.

In current OpenCL implementations pinned memory can be acquired by creating a buffer that is backed by CPU memory and requesting a pointer to that memory from the OpenCL runtime. This is not specified by the standard but works on both, AMD's and NVIDIA's, implementations.

The performance for small buffers can be better seen in Figure 3.19 which shows the latency for transfers from GPU to CPU memory. For pinned memory the minimum latency of about 10 µs is kept for buffers up to a size of 64 KiB. Except for very small buffers the latency using plain memory is at least 50 µs.

For comparison, in 2010 NVIDIA has shown reduction kernels to sum up 2^{22} integers, which means a buffer of 16 MiB, in less then 268 µs [81]. This is only five times the minimum latency observed.

The performance for transfers from host to device is similar. But, when using plain memory, the latency for small buffer increases to about 200 µs while the latency for pinned memory stays at about 10 µs.

Figure 3.20 shows that the full performance of 12 GB/s is reached when repeatedly sending data from the GPU to the host and back. Compared to the single-direction case only larger transfers reach this limit. As Figure 3.21 shows, even for pinned memory the latency increases to about 55 µs.

This last benchmark does not represent the use case found in actual applications. But, it mimics a typical pattern in device-to-device communication, where the data is send to the memory of the CPU by one GPU and afterwards fetched from there by another. Thus, it provides an upper limit for such communication patterns. Contrary to an actual multi-GPU implementation it does not include the synchronization overhead. Of course, for comparison with actual device-to-device communication performance values, the data rate must be halved to get the net buffer-to-buffer copy performance.

3.4.2. Communication between Devices

In OpenCL there are multiple methods to transfer data from one device to the other. Figure 3.22 shows the performance for a variety of those methods when copying data between the memories of the two GPUs located on the same AMD FirePro S10000 in the SANAM cluster.

Maybe the most obvious variant to transfer the data between the two GPUs is to copy it from one GPU to the CPU memory and then send it on to the other GPU. Consistent with the single-GPU measurements shown in Figure 3.18 and Figure 3.20, the variant using plain memory performs worse than the variant using pinned memory. Therefore, it is excluded from Figure 3.22. Yet, even the variant using pinned memory peaks at 3 GB/s, and only reaches this performance for very specific buffer sizes.

Another set of options stems from the possibility to map buffers into the address space of the CPU. In the general case any OpenCL implementation can implement this

feature by copying the buffer to the CPU memory and copying it back to the GPU once it becomes unmapped. This is possible, as any operation on a buffer is undefined as long as that buffer has been mapped for access by the CPU.

The OpenCL implementation by AMD extends this feature to allow the CPU to directly access data stored on the GPU. For this an AMD specific flag must be passed when creating the buffers. This slightly reduces the portability of the code. But, as the flag can simply be skipped on other platforms functional portability remains.

Using these mapped buffers I have created three additional transfer variants. Mapping the buffers on both the source and the destination GPU to the CPU address space allows to implement a programmed input/output (PIO) method. In this case, once both buffers have been mapped, the buffer is transferred from the source to the destination via a classical `memcpy`. As the benchmark is implemented in Python, I utilized the corresponding functionality for copying Numpy arrays. Figure 3.23 shows that this is the fastest method for copying buffers in the range from 4 B to 512 KiB. In the plateau for small buffers it shows a latency of about 60 µs.

A second variant utilizing mapped buffers is to only map the buffer of the source GPU into the CPU address space. Passing `clWriteBuffer`, the normal OpenCL function for sending data from host to device, a pointer to the mapped memory results in a successful transfer. For the OpenCL runtime the mapped buffer seems to work like any plain memory. Even the performance curve is very similar to that shown for plain memory in Figure 3.18. The transfer peaks at about 4.5 GB/s and is the fastest variant for buffers larger than 64 MiB.

The inverse variant, mapping the destination buffer and writing to it from the source GPU performs much worse. It shows similar performance to the two variants using an explicit cache in CPU memory. Therefore, it is excluded from the plots.

A completely different variant to perform the transfer utilizes the fact that buffers in OpenCL are not explicitly bound to a device. Simply issuing a buffer copy on the destination device—passing a buffer on another device as the source—leaves it up to the OpenCL implementation how to actually implement the transfer. As Figure 3.22 shows this is the fastest method for copying buffers of a few MiB on the AMD FirePro S10000, peaking at 5 GiB/s. However, the implementation in AMD's OpenCL seems to be broken for buffers larger than 64 MiB where its performance falls below that of the custom implementations.

The two AMD FirePro S10000 in a node of SANAM are connected to different PCIe root complexes located in separate CPUs. Transfers between two GPUs located on distinct AMD FirePro S10000s, therefore, are forced to pass through the Quick Path Interconnect (QPI) connecting the two CPUs. As Figure 3.24 shows this results in reduced performance for the variant using a cache of pinned memory and the one where the second GPU reads from the mapped memory of the first. The buffer copy variant, however, still peaks at 5 GB/s.

For small buffers the performance is identical to that observed transferring the data between the two GPUs on a single AMD FirePro S10000. This is shown by Figure 3.25.

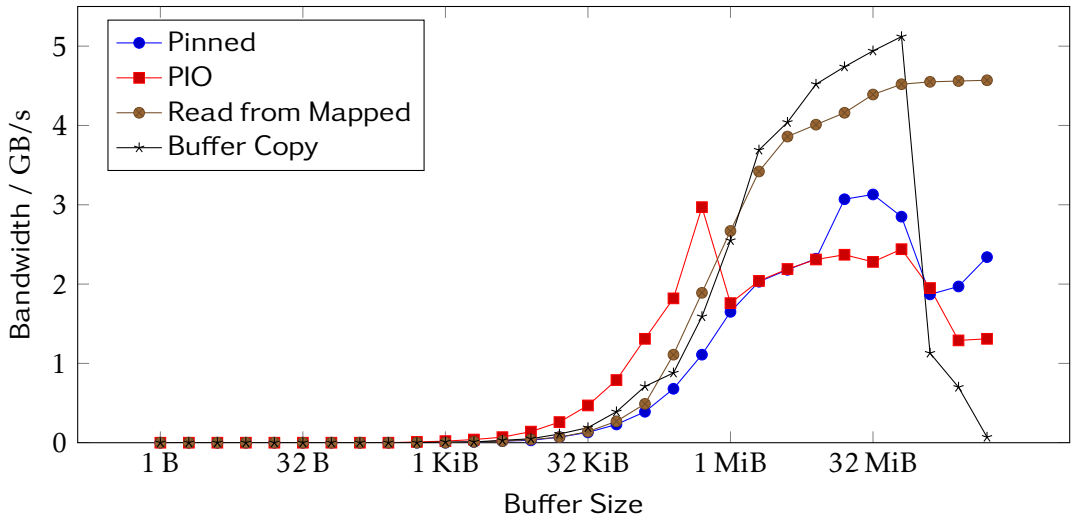

Figure 3.22.: Utilized Bandwidth transferring data between the two GPUs of a single AMD FirePro S10000 in a node of SANAM.

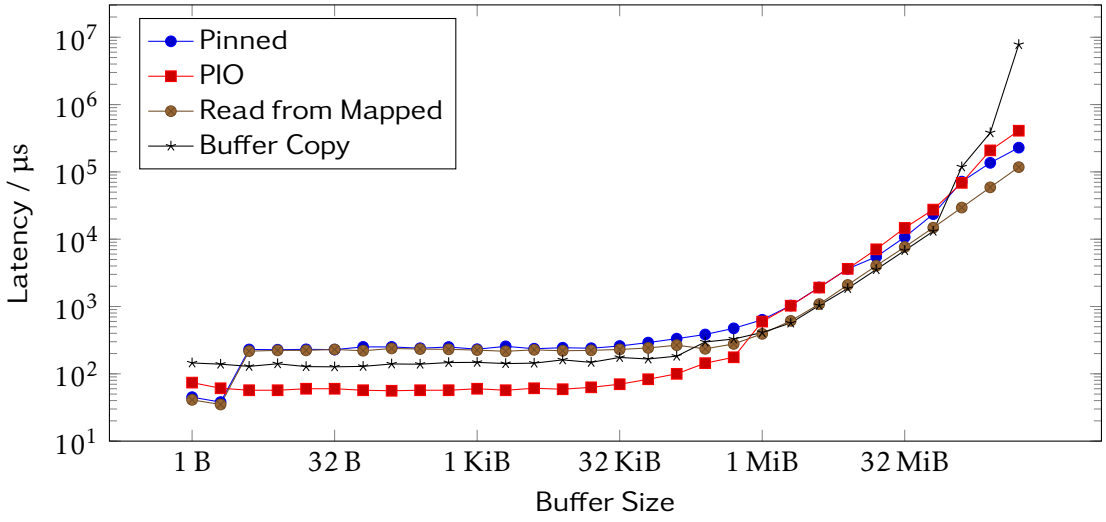

Figure 3.23.: Latency transferring data between the two GPUs of a single AMD FirePro S10000 in a node of SANAM

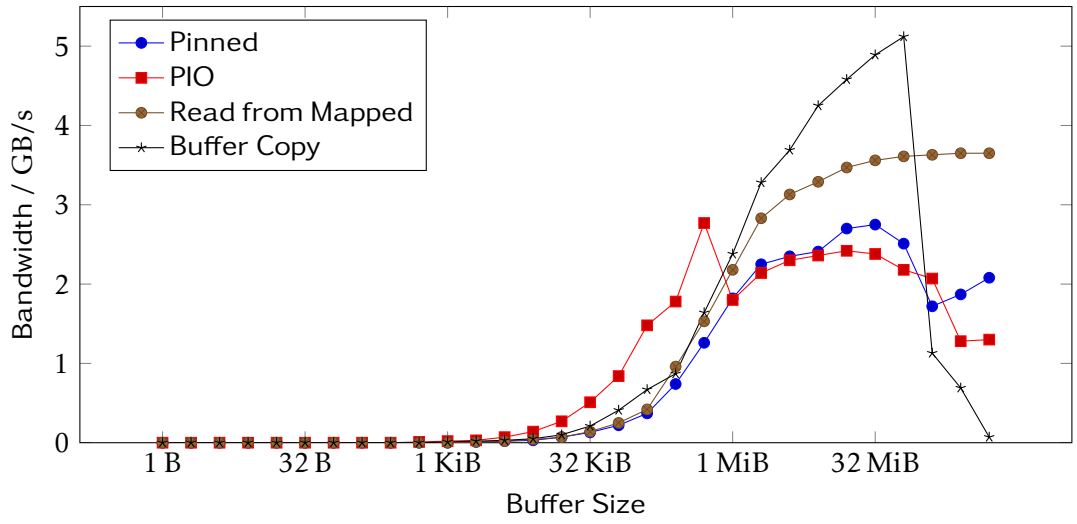

Figure 3.24.: Utilized Bandwidth transferring data between two GPUs of separate AMD FirePro S10000s in a node of SANAM.

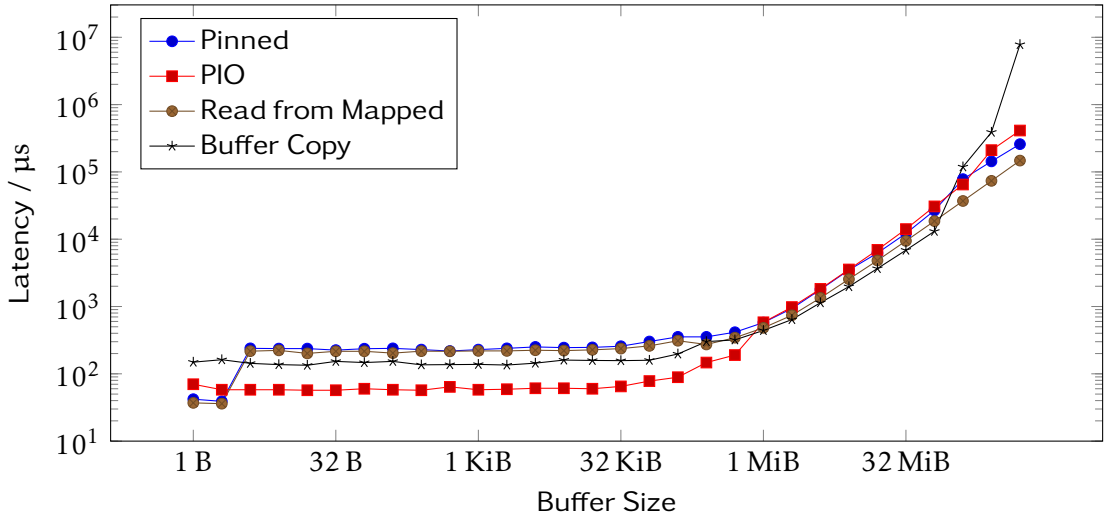

Figure 3.25.: Latency transferring data between two GPUs of separate AMD FirePro S10000s in a node of SANAM.

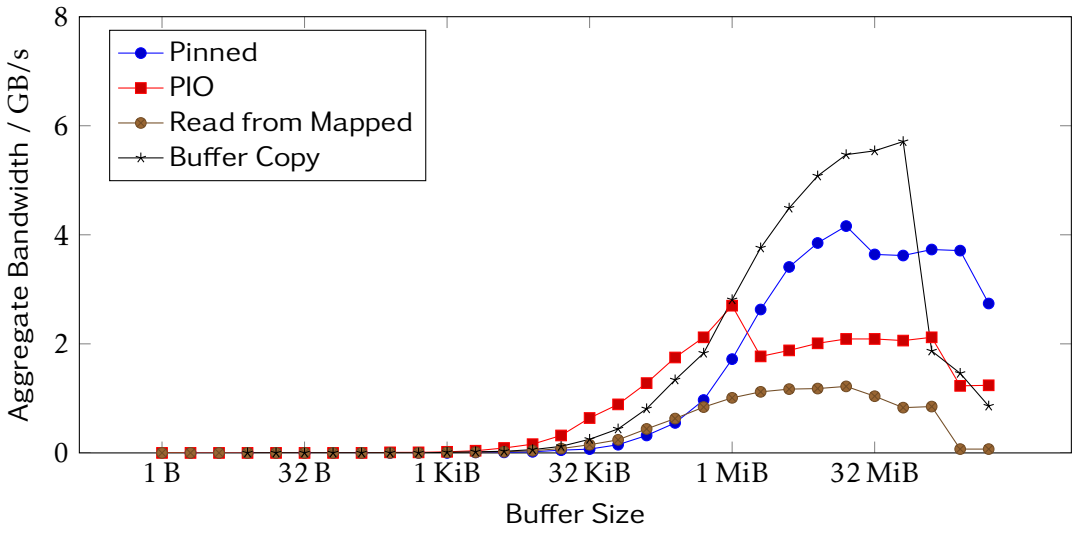

Figure 3.26.: Utilized Bandwidth swapping buffer contents between the two GPUs of an AMD FirePro S10000 in a node of SANAM

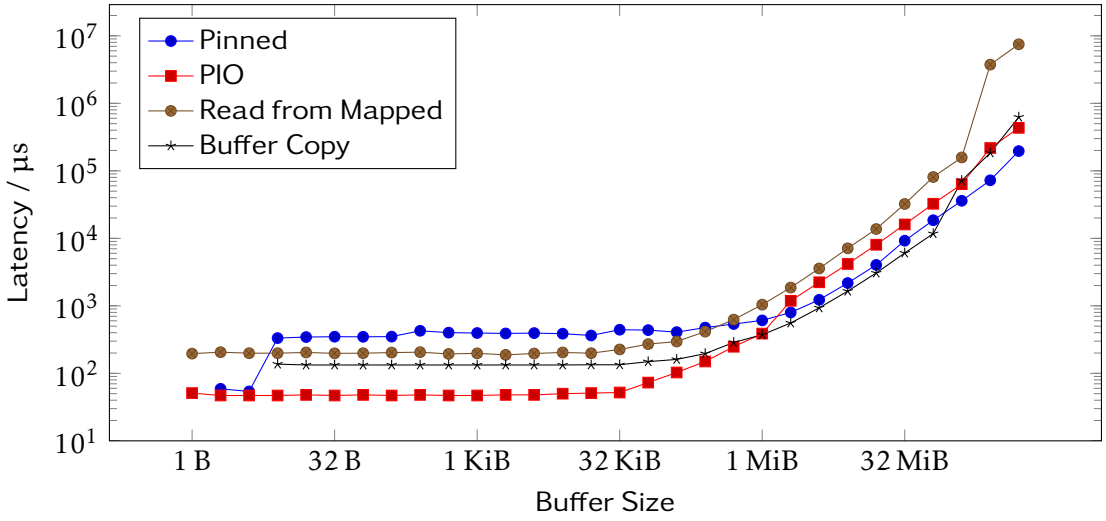

Figure 3.27.: Latency swapping buffer contents between the two GPUs of an AMD FirePro S10000 in a node of SANAM

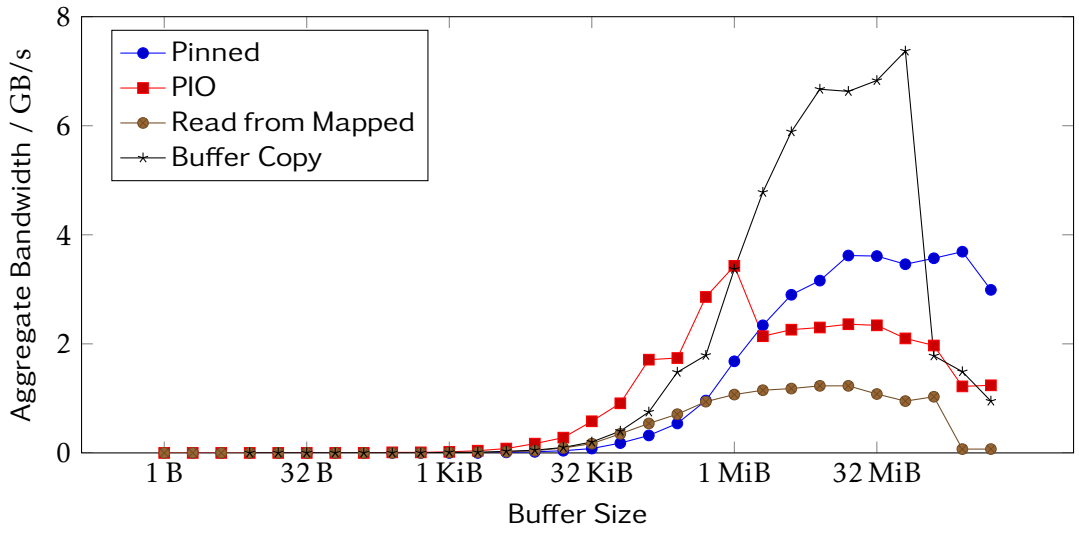

Figure 3.28.: Utilized Bandwidth swapping buffer contents between two GPUs of separate AMD FirePro S10000s in a node of SANAM.

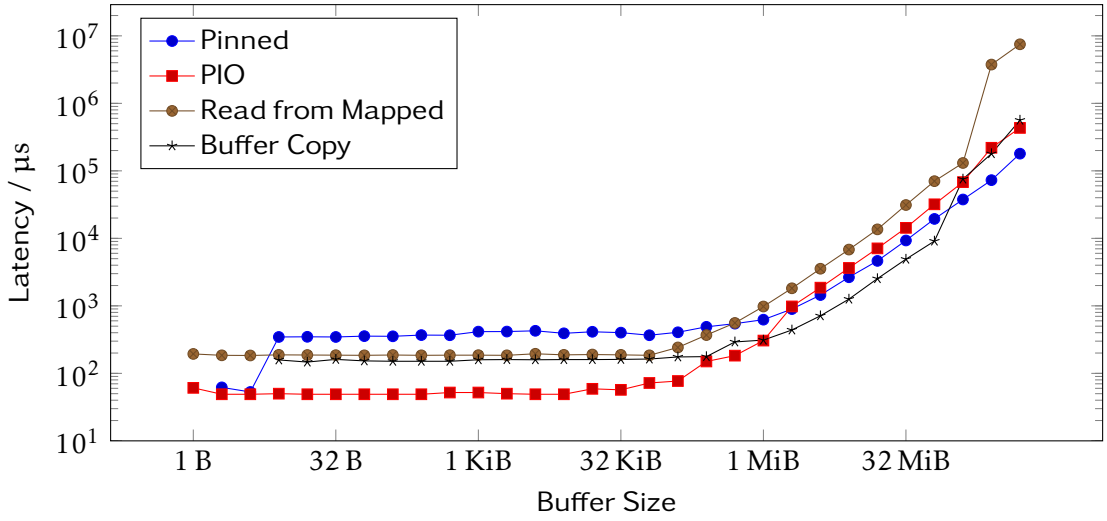

Figure 3.29.: Latency swapping buffer contents between two GPUs of separate AMD FirePro S10000s in a node of SANAM.

3.4.3. Bidirectional Communication between Devices

In a typical multi-device usage pattern, each device requires some data from other devices. A typical example is the halo or ghost cell update in domain-decomposed problems. In this case, the aggregate bandwidth and latency over all potentially overlapping transfers is important.

As before, I implemented a simple benchmark in *clPCI* [80] to evaluate multiple communication patterns between the two GPUs of an AMD FirePro S10000 in a node of SANAM. The benchmark allocates the buffer on each device and splits it in half. The upper half of the buffer is sent from the first device to the second. The lower half is sent from the second device to the first.

Figure 3.26 and Figure 3.27 show the best performing methods for a variety of buffer sizes. The simple approach of copying via a cache of pinned memory has a high latency of 300 µs to 400 µs for small buffers. Nevertheless, it is the only method that allows for an aggregate bandwidth of more than 3 GB/s for buffers of more than 64 MiB, which means a transfer of 32 MiB in each direction. Still, it peaks at less than 4 GB/s showing the same performance as a simple unidirectional device-to-device transfer. For best performance with this method separate cache buffers for each direction must be used on the host.

The PIO variant shows the best latency for small buffers. It stays at 50 µs up to a buffer size of 32 KiB. As this variant utilizes only a single thread, high performance for larger buffers is not to be expected. Yet, even a multi-threaded variant peaks at 3 GB/s and shows worse performance than the pinned-memory cache variant.

The best peak performance is reached by the variant leaving the transfer details up to the OpenCL implementation by simply requesting a buffer copy. For best performance the buffer objects may not be accessed by two transfers at the same time, even if the transfers are non-overlapping. This is ensured by first locally copying the data to be sent into a cache buffer on the same device. Given the 200 GB/s bandwidth that can be reached for on-device copies, the copy to the cache buffer is irrelevant for the transfer duration. The speed-up from the cache buffer is 0.5 GB/s to 1 GB/s in the buffer size range from 1 MiB to 64 MiB. The peak aggregate bandwidth is 6 GB/s, which is only 1 GB/s more than the unidirectional device-to-device performance.

When swapping buffer contents between the two GPUs on the AMD FirePro S10000, the classical methods are limited by the shared PCIe connection to the CPU memory. A single GPU can send up to 12 GB/s to the CPU memory. Two GPUs sending data at the same time can each only use half this bandwidth, limiting the transfer to 6 GB/s per GPU. While the aggregated bandwidth is still 12 GB/s, transferring the data back to the GPUs is again performed at only 6 GB/s per GPU. This results in a net transfer rate of only 3 GB/s per transfer direction. Therefore, any scheme swapping the buffer contents via the host is always limited to these 6 GB/s. Given this limitation the buffer copy variant shows optimal performance.

This scheme could be improved upon by utilizing the full-duplex property of PCIe. Splitting the transfer into multiple smaller transfers would allow to overlap the transfers from and to the GPU. This could allow large transfers to reach up to 12 GB/s.

Swapping buffer contents between GPUs on separate AMD FirePro S10000s is not effected by this limitation. Figure 3.28 shows that the buffer copy variant, in this case, peaks at 7.4 GB/s. This is about 75 % of the performance that would be reached if two uni-directional transfers could be performed without influencing each other. Due to the set-up in SANAM, the data must pass through the QPI interconnect in this case. Two AMD FirePro S10000 attached to the same PCIe root complex might provide even better aggregate bandwidth. As the curve for the PIO variant in Figure 3.29 shows, the latency is not influenced negatively.

3.4.4. DirectGMA

In Subsection 3.4.2 I have shown a variety of methods for transferring data between GPUs. However, they all failed to fully utilize the bandwidth provided by PCIe. This was caused by the detour via CPU memory that all of these methods used. Especially when performing bidirectional transfers between the two GPUs on an AMD FirePro S10000—as in Subsection 3.4.3—the achieved aggregate bandwidth is far below the theoretical peak.

To fully utilize the PCIe bandwidth the data needs to be directly transferred from one GPU to the other. NVIDIA's solution for this problem is called GPUDirect. However, it is limited to the NVIDIA CUDA framework, and, therefore, cannot be used with OpenCL.

AMD provides DirectGMA for this task. While not available on all GPUs, it can be used from within OpenCL, allowing an application to gracefully fall back to another transfer method on systems that do not support it.

As DirectGMA is not yet supported by PyOpenCL, *clPCI* [80] could not be extended to benchmark the performance of DirectGMA. Therefore, I rewrote it as *CLPCI2* [82] in C++11.

Figure 3.30 shows the performance reached when sending data from one GPU to the other. For reference the performance using the classical buffer copy method is also included. Between the two GPUs of an AMD FirePro S10000 the DirectGMA based transfer is by far the best performing option, reaching up to 8.4 GB/s. However, transfers that must pass through the QPI link perform worse than any of the classical methods discussed in Subsection 3.4.2. The same behaviour can be observed on NVIDIA GPUs when using GPUDirect [83].

Contrary to the methods discussed in Subsection 3.4.3, the DirectGMA based transfer method is able to utilize the duplex capabilities of PCIe. As Figure 3.31 shows, the DirectGMA based transfer method reaches up to 15 GB/s when swapping buffer contents between the two GPUs of an AMD FirePro S10000.

The implementation using DirectGMA does not expose its full potential, yet. DirectGMA also enables GPUs to signal each other. This allows to synchronize the execution between GPUs without involvement of the CPU. However, this feature was not available in the used preview-version of the GPU driver.

Another limitation of DirectGMA is the maximum transfer size. All buffers utilized for DirectGMA transfers must fit within an area of 96 MB. However, the size of this

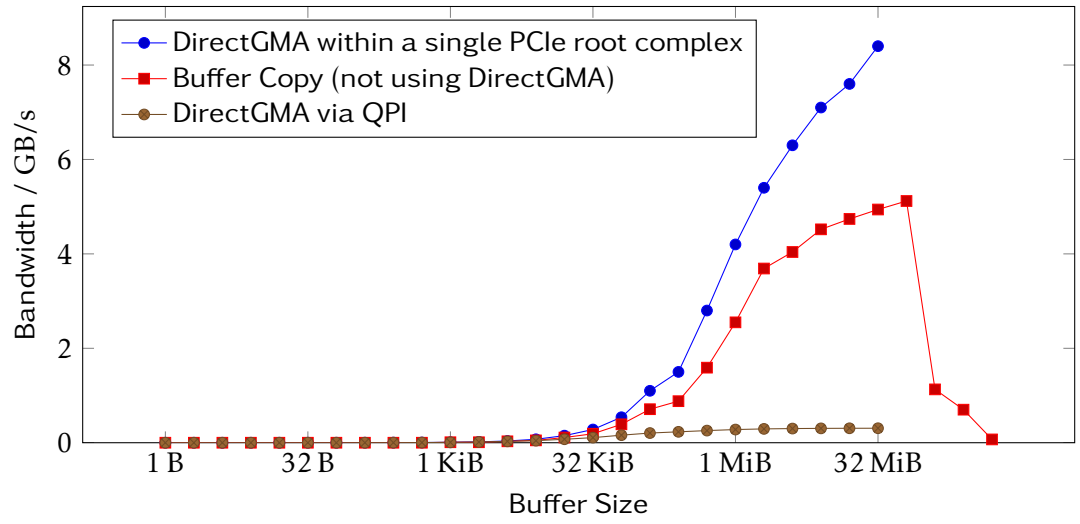

Figure 3.30.: Utilized bandwidth transferring data between two GPUs using DirectGMA on SANAM.

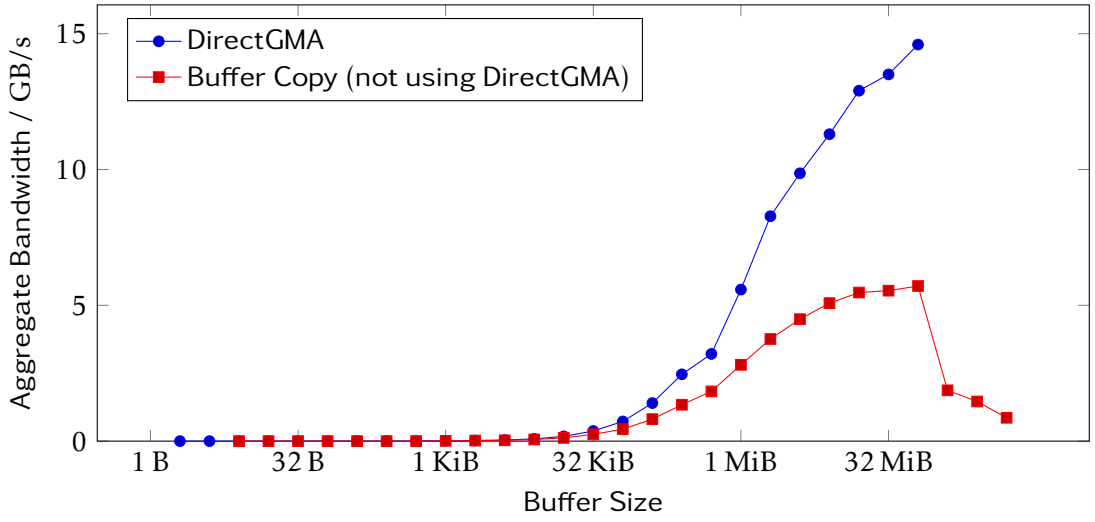

Figure 3.31.: Utilized Bandwidth swapping buffer contents between the two GPUs of an AMD FirePro S10000 in a node of SANAM using DirectGMA.

area can be increased via configuration parameters. Alternatively, larger transfers can also be split into multiple transfers utilizing smaller transfer buffers.

3.4.5. Summary

Communication between GPU and CPU memory does not require any optimization but pinned memory to reach peak performance. However, if communication between the devices is required, the situation becomes more complex. In this case, the most portable solution is to simply rely on OpenCL's native buffer copying functionality. It provides good performance for all transfer sizes and works on any platform.

On the AMD platform two further optimizations are possible. When transferring small buffers, utilizing PIO can reduce the latencies. This can especially be of advantage in situations where some processing will have to be performed on the transferred data. An example for this are reduction implementations. In this case, the CPU can perform the transfer and the processing in one step. As the only proprietary part of this optimization is a special flag during buffer creation, it can gracefully degrade on other implementations, where mapping the buffers will then become an implicit copy.

For large buffers DirectGMA provides a faster alternative to OpenCL's native buffer copying functionality. However, this does not work well if the data must pass through a QPI link. Therefore, any application relying on maximum transfer performance will have to benchmark the DirectGMA performance on start-up and fall back to the basic functionality if that performs better.

An interesting alternative is the utilization of Accelerated Processing Units (APUs). As those share the same memory for CPU and GPU, no transfer is required in between the two, minimizing latencies. For transfers between multiple APUs classical CPU data transfer methods can be used. This avoids the QPI bottleneck observed in the DirectGMA case and the detour via CPU memory in the traditional methods to transfer data between two GPUs.

Chapter 4.

CL^{2}QCD

The physical problems currently investigated at the Johann Wolfgang Goethe-Universität Frankfurt am Main include the quark gluon plasma (QGP) and the thermal transition of QCD with dynamical fermions [84–86] as well as in PGT. Previously those calculations mostly relied on the *tmlqcd* program suite [3] and an application based on *QGP++*. Neither of these applications had previous support for OpenCL. Also, none of the existing GPU codes supported the twisted mass variant of LQCD. To account for both the concepts of OpenCL and the LQCD variant used in Frankfurt, a new application was developed from scratch.

In 'LatticeQCD using OpenCL' [19] and 'Lattice QCD based on OpenCL' [20] some details on CL^{2}QCD have already been published. While the other authors focused on the initial implementation and the implementation of all physical features, my work was focused on performance optimizations and the implementation of technical features.

In this chapter I will first motivate my work by collecting a list of requirements for the application by looking at some of the challenges faced in LQCD computations. Then I will give an overview over the architecture of the application and finally show how I applied the techniques introduced in Chapter 3 to optimize the performance of CL^{2}QCD.

4.1. Application Requirements

While excellent single-GPU performance is one of the design goals of CL^{2}QCD, it is not always sufficient. Modern supercomputers like SANAM provide hundreds of GPUs. On SANAM, this allows to run 600 application instances with different parameters and pseudo-random number generator (PRNG) seeds at the same time. In this mode the application allows to increase the statistics of a measurement at a tremendous rate. This is extremely useful for parameter-sweeps, where each set of parameters requires an own instance of the HMC, as well as for the measurement stage, where the analysis can be performed by a separate application instance for each configuration.

However, to sufficiently sample phase transitions, the length of the HMC chain is important. Its length should exceed 2000 steps of the HMC algorithm. Using CL^{2}QCD, on SANAM this will mean a calculation duration of 1500 h to 2000 h—which means two to three months—to study the thermal transition of QCD with dynamical fermions on a $32^3 \times 12$ lattice at a pion mass of approximately 270 MeV. After that time one will have

very good statistics, as one can have completed 600 chains of that length. However, obviously it is of advantage to reduce that wall time of such calculation even if that means some drawbacks in the obtained statistics.

In addition, the memory of the GPU is a limiting factor regarding the problem size. An HMC on a $32^3 \times 12$ lattice can currently not be performed on GPUs that have only 1 GiB of memory. While this is not an issue on SANAM, the next problem size to study would be $48^3 \times 16$. This is too large even for the 3 GiB of memory available on the AMD FirePro S10000.

When setting up a new chain, this chain first has to thermalize. The number of HMC steps required for this process is of the order of 1000. If creation of many chains is desired this can be avoided by using a pre-generated chain, which has already been thermalized. From this chain new chains are forked. Of course, the correlation length in the original chain must be taken into account when selecting the new start configurations. Otherwise the new chains start in a correlated state. To quickly start data production, however, it is of advantage if one could directly start into new chains. For this, single chain performance must be sufficient to quickly run through the thermalization process.

There are additional requirements that put limits on the optimizations that can be performed. A correct HMC algorithm requires the molecular-dynamics step to be reversible [87]. Due to the finite precision of the calculations, this reversibility is inevitably lost. To keep the algorithm as close to reversibility as possible, as few rounding errors as possible should be introduced. Thus, it is usually not sufficient to perform these calculations using SP. Any utilization of less-precision calculations inside the molecular-dynamics step must, therefore, ensure to not reduce the precision of the overall calculation.

In addition, it is desirable to be able to utilize different discretization schemes. Thus, optimizations should not prevent the efficient implementation of other discretization schemes inside CL^{2}QCD.

To summarize, these are the key requirements given for the implementation:

- Support for twisted mass LQCD
- Keep the application flexible enough to support other variants in the future.
- High performance on a single GPU for the following use cases:
  - Analysis
  - Parameter range scans
- Support for multiple GPUs to support the following use cases:
  - Speed up long running chains.
  - Enable calculations too large to fit into the memory of a single GPU.
- Perform computations at DP accuracy.

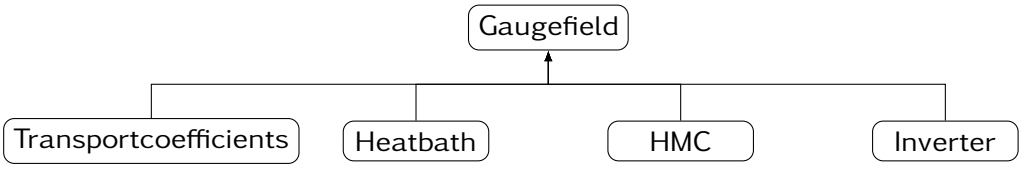

Figure 4.1.: Class hierarchy of the gauge-field classes in the old architecture of CL^{2}QCD [19]. The arrows depict generalization relationships.

4.2. Architecture

The architecture of the application developed historically in two stages. The first iteration of the architecture [19]—designed and implemented by Pinke *et al.* —was focused on hybrid systems, such as LOEWE-CSC. In the second iteration I refactored the architecture to ease modification of low-level routines without interfering with high-level application logic. This allows to keep changes required for efficient operation on specific devices rather local. In addition, it eased the implementation of features required on systems with a higher ratio of GPUs to CPUs, such as SANAM. Common to both architectures is that the host program is set up in C++11. The actual computation is performed by kernels written in OpenCL.

4.2.1. The Initial Architecture for Hybrid Systems

The initial architecture [19] was based around a central class called *Gaugefield*. This class provided import and export functionality for the gauge field and took care of initializing and tearing down OpenCL. As depicted in Figure 4.1, for each binary—e.g. the HMC and the inverter used in the analysis stage—the *Gaugefield* class was extended.

The execution of code on the devices was wrapped in *opencl_device* objects. Their base class provided basic functionality to read in and compile device code as well as memory management capabilities. The derived classes had three purposes: management of the required OpenCL buffers, management of OpenCL kernels, and implementation of full on-device algorithms like the solver. Distinct implementations existed for different functionalities. There was one for the pseudo-random number generator (PRNG), one to implement linear algebra on the spinor fields, one to provide HMC specific code, and so on. Those which required functionality of another class extended that one. Thus, the class providing the functionality for the fermionic fields extended the PRNG. This way it could access the PRNG state and code to generate random fields. The full class hierarchy of *opencl_device* classes is shown in Figure 4.2.

Each child of the *Gaugefield* used one or more instances of the *opencl_device* variants to implement their algorithm. For different tasks it would use separate objects, possibly using different devices. For example, the inverter would use on instance of the *Fermion* module to perform the solve on the GPU. Then it would copy the resulting spinor field into an instance of the *Correlator* module bound to the CPU which would perform the operator evaluation [19].

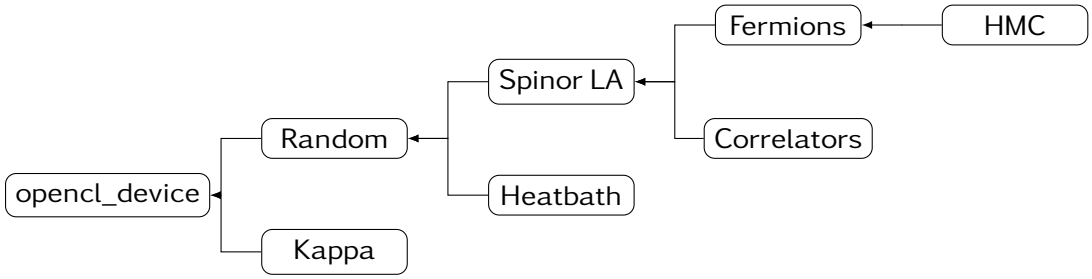

Figure 4.2.: Class hierarchy of the OpenCL modules in the old architecture of CL²QCD [19]. The arrows depict generalization relationships.

4.2.2. The Second Generation Architecture

Once we had shown a full HMC application, we wanted to extend the application in two directions. We wanted to optimize it for a larger variety of systems, both in the specific GPUs used as well as in their number. Trying to achieve this goal, the old architecture showed several limitations: Buffers were not type-safe. Thus, one could easily pass a buffer containing a gauge field to a kernel expecting a spinor field. In addition, concepts were not separated properly. Thus, modifications to the buffer layout affected a large part of the application. Therefore, I decided to design a new architecture for the application.

While the new architecture imposes a major refactoring it is an evolution from the original. It focuses on cleaner separation of concepts while retaining parts of the old architecture, like the OpenCL modules. Those, however, now only exist to generate and wrap the OpenCL kernels used by the other parts of the application. They are no longer responsible for managing the buffers used by the application and they no longer contain algorithms that operate on the data. These responsibilities have been moved to dedicated classes in the *Buffers* package and functions in the *Algorithms* package, respectively.

The code is now split into several packages—implemented as namespaces in C++. Each package represents a distinct level of abstraction. For reference, the most prominent packages and classes of the new architecture are shown in Figure 4.3.

How all those objects interact shall be explained using the example of the inverter. An application will first create an *Inputparameters* object. It will then instantiate *System*, which will require the *Inputparameters* object for its initialization. Afterwards the application can create instances of the classes *Gaugefield* and *Spinorfield*. These will again require the *System* instance for initialization. To perform the inversion the application will now invoke the inverter algorithm. Given instances of the classes *Gaugefield* and *Spinorfields* this will perform the inversion.

The previous paragraph describes the object interaction as it is observed from the top level of the application. The implementations of these objects add further interactions. Creation of objects from the *Lattices* package will cause the newly created objects to create *Buffer* instances on the devices described by the *Device* objects of the given

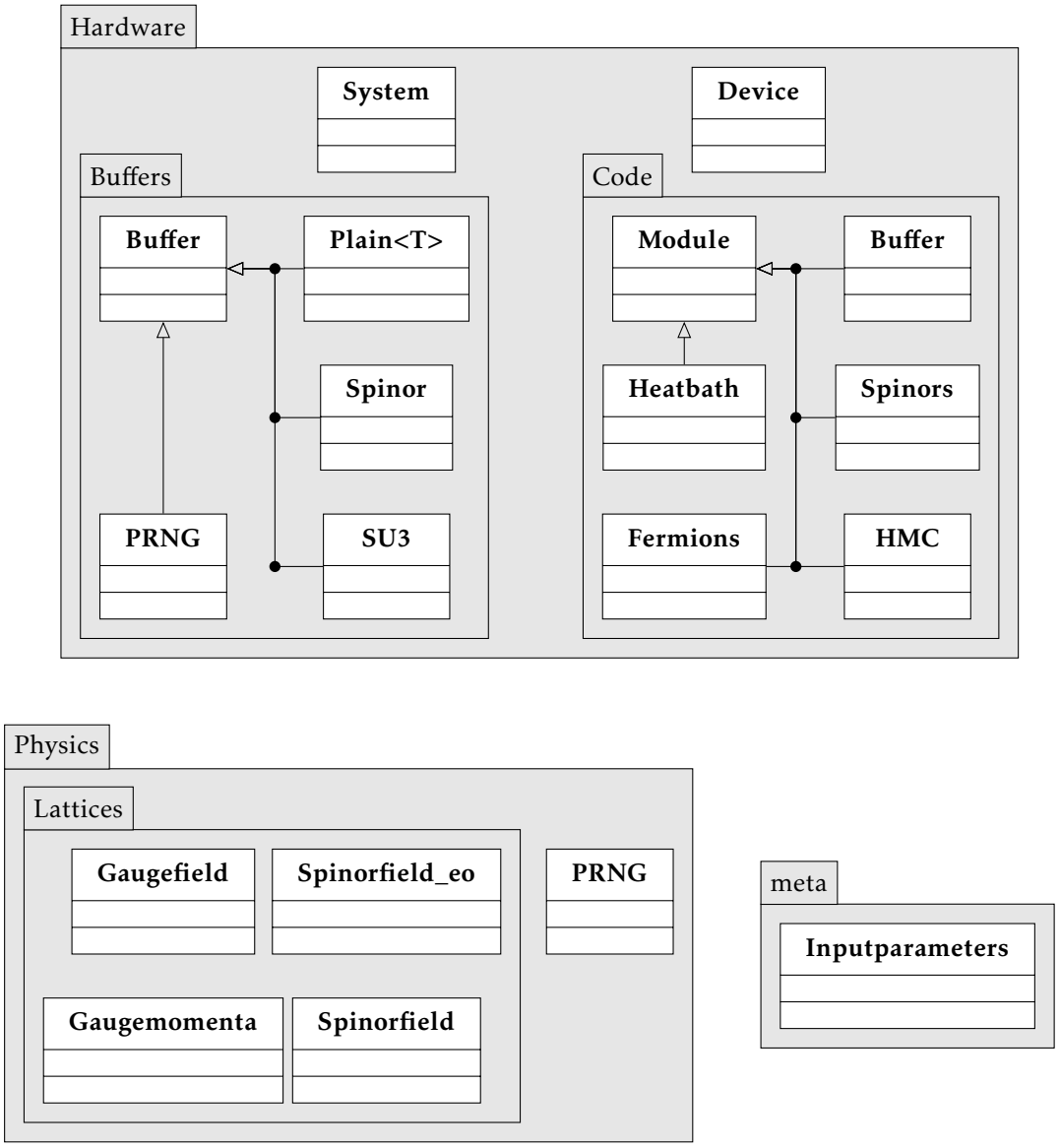

Figure 4.3.: Class diagram overview of the new architecture. To reduce complexity usage relations, members and operations are not shown.

System instance. In addition, creation and invocation of the *Inverter* object may cause the creation of additional objects from either the *Lattices* or *Buffer* package. Note however, the latter should ideally be wrapped by some class living in the *Physics* package. Further, all objects in the *Physics* package will utilize objects from the *Code* package to perform operations on a device.

The package *Physics* contains everything required to describe the program logic. It contains the sub packages *Lattices*, *Fermionmatrix*, and *Algorithms*. The first contains classes representing lattices of different type. One example is a complete field of spinors. The second contains the high-level implementation of the fermion matrix and related operators working on the fields. The third contains algorithms that work on these objects, e.g. an inverter.

A lattice class does not only wrap the buffers containing the data, but it also provides the operations on this data. To implement those operations it will use the OpenCL modules from the *Code* package. The advantage of this is, when modifying the way a certain lattice type is handled, algorithms using this type do not need to be changed. This allows, for example, to change the number of buffers a gauge field is stored in without having to change the inverter algorithm.

Not all classes in the *Lattices* package contain actual lattices. It also contains a special class *Scalar* which is used to represent scalars of all kinds. This is required for linear algebra operations, like scalar products or the scaling of lattices.

The members of *Algorithm* are not shown in Figure 4.3 as this package now only contains functions. These implement the high-level algorithms, like the inverters and the integrators, that are composed by chaining multiple computational kernels.

For layout reasons the same is true for the package *Fermionmatrix*, even though it contains objects. These, however, are simple callable classes, wrapping the functions implementing the fermion matrix and related operators. Those operate on the objects in the *Hardware* package, just like the *Algorithm* and *Lattices* package.

The package *Hardware* contains all code that is responsible for matching the logic of the application onto the actual hardware. Of this only the *System* class should be directly used by the application. Anything else is used to actually implement the types from the *Physics* package on the given system. Children of the *Buffer* class provide a type-safe alternative to conventional OpenCL buffers. Classes in the *Lattices* package use them to store their data on the given hardware in proper formats. Note that the mapping from lattices to buffers does not have to be one to one. A lattice class might store its data in multiple buffers, e.g. on multiple devices. The OpenCL modules still wrap the kernels, but contrary to the previous architecture the types of the kernel arguments are now checked. In addition, the OpenCL modules no longer use inheritance to access each others features. The objects are instantiated as per-device singletons and access each other in case they need each others features. Usually, this only occurs in the kernel building stage. For example, the *Fermion* module requests the source files and required build parameters to operate on spinors from the *Spinor* class.

The package *Meta* contains things that neither fit well into the *Physics* not into the *Hardware* package. For example the parsing and representation of input parameters are located here.

This new architecture throws overboard an important feature of the previous architecture. It is currently not possible to run different algorithms on different devices. This is caused by the lattices being initialized using a *System* instance. I chose this limitation to simplify the refactoring process. A further evolution of the architecture can solve this by using a set of devices instead of the whole system for these initializations. This will then allow an even more flexible distribution of tasks to devices than the original architecture did.

4.2.3. Common Architectural Features

There are, of course, architectural features that are common to both architecture iterations. Typical simulation parameters, like the dimension of the lattice or the lattice spacing, which are variables in the host part of the application, are passed as compile time constants to OpenCL. This reduces the number of arguments that must be passed to the compute kernels on invocation and allows the compiler to optimize the code for the specific problem at hand.

The OpenCL source is laid out such that a single compilation unit only includes a single—or at least only few—kernels. This has multiple advantages: If the compiler fails to build the source and also fails to provide a helpful error message, the size of code that can have caused the error is less. Also, if only code of a single kernel was modified, the compilation time for a specific kernel is less.

Application initialization time is a large issue during development of an OpenCL based application. Therefore, a custom cache for the compiled OpenCL code is included in CL^{2}QCD. Every time a kernel is generated the binary of that kernel is retrieved from the runtime and stored in a temporary file. Subsequent invocations of the application will try to reconstruct the kernel from this file and only recompile the sources if the compute device, the build options or the sources themselves have changed. Some OpenCL runtimes include such a cache on their own. On those the custom cache does not cause a significant overhead.

After clearing the cache, running the suite of automatic tests included with CL^{2}QCD takes about 58 min on our development system gpu-dev04. This time already profits from the cache, as multiple tests utilize the same kernels. A second run, starting with an up-to-date cache, completes in only 12 min.

In the kernel code all data types are implemented as structures, with all the required operations defined for them. This might in some cases require more registers than simply operating on arrays of scalars stored in main memory. However, in Section 3.2 I have shown that it is not required to optimize for minimal register usage as long as register spilling can be avoided. Therefore, we opted for this implementation strategy for its better code readability.

It is not trivial to estimate what the register overhead of the structure based implementation strategy is. The exact register requirements highly depend on the optimizer. The possibly higher register requirements of the structure based approach, however, can easily be seen when looking at the addition of two structures of four floats. When operating on arrays of scalars there only has to be space for three floats in registers.

As addition is element-wise, only one float from each operand has to be loaded at the same time. Additionally there has to be room for one element of the result. Using actual structures requires four times this space, as the whole structure has to be stored for each operand and the result. In the case of spinors, register requirements would increase from three floats to 72. This is, however, a worst case situation as for most operations, e.g. multiplication of an $SU(3)$ matrix with a spinor, more than one element of each structure is required at the same time anyway. In addition, given the high latency of GPU memory it does not make sense to completely serialize handling each element in a structure. Therefore, the register usage should be higher even when performing all operations using scalar types, as the optimizer will use different registers for different elements to enable the exploitation of ILP.

CL^{2}QCD can collect statistics from the OpenCL compiler similar to *pyclKernelAnalyzer* [76]. This enables register optimizations without the use of external applications. This is important as register usage can vary with the choice of run-time parameters like the lattice size. Based on this feature CL^{2}QCD will also warn users of development builds if the compiler uses scratch registers.

The storage format of the data types on the device is abstracted as far as possible. Inside the kernel code the actual data format used is wrapped by object load and store functions. These are automatically chosen at compile time to match the storage format. The storage format itself is—at least in the new architecture—chosen by the buffer object and depends on the hardware used. The buffer object includes data import and export functions, which will automatically convert the data between the device specific format and the default AoS layout used in the host code. Thus, any developer not explicitly optimizing the storage format can be completely unaware of the optimizations performed on it.

As up till now they have not been relevant for the overall runtime, all required linear algebra operations have been implemented in a straightforward manner. The only optimization was implicitly given by the data type storage format. International Lattice Data Grid (ILDG) compatible I/O has been implemented as well as the Ranlux [88] PRNG, as it is the standard choice for LQCD simulations. We use the original implementation on the host while on the device we use RANLUXCL [89], an open-source OpenCL implementation of Ranlux. For testing purposes we have also implemented the generator from *Numerical Recipes 3rd Edition: The Art of Scientific Computing* [90], but it has not been used for any results reported in this thesis. Initialization of the random number generator follows the usual Ranlux rules. They are applied across the host and the device, where each OpenCL thread runs on its own Ranlux PRNG state.

Since on different GPU drivers we have observed multiple miscompilations of our code during development, we added regression tests for most of our OpenCL functionality. This allows us to quickly check new drivers for incorrect output. A special challenge is that the likelihood for compiler errors scales with code complexity. Thus, a function might work perfectly in a simple test case but will produce errors when integrated into a larger kernel. Therefore, it is important to not only test each building block for regressions but also repeatedly check whether they still work as expected when being used in larger kernels.

CL^{2}QCD also includes the capability to monitor the memory utilization of each device. This is important due to the size of the fields involved. In DP a gauge field requires 144 B per element, resulting in 216 MiB for a $32^3 \times 12$ lattice. Each element of a spinor field requires 192 B per element. This results in 36 MiB consumed memory for a $32^3 \times 12$ lattice using even-odd preconditioning. If not using even-odd preconditioning, twice as much memory is consumed by the spinor field. As some of the fields are not required at all times of the computation, a mechanism to swap them to host memory has been integrated.

4.2.4. Common Code for CPUs and GPUs

As OpenCL can be used both for CPU and GPU programming, we use a single source code for the CPU and the GPU implementation. To cater for the different architectures, we introduced some abstractions.

The optimal looping strategy differs in between the CPU and the GPU. Assuming no SMT, loops on the CPU perform best when each core works on its own consecutive block of memory. Thereby, it can best utilize its time-local cache to reduce the number of actual requests performed on the memory. On the GPU however, the best memory throughput is achieved if consecutive cores read consecutive elements from memory. Therefore, a loop should always move though the data using large strides. We use a macro called `PARALLEL_FOR` to implement these different looping strategies transparently. Note, that Smelyanskiy *et al.* report this pattern to also be of advantage if SMT is used on CPUs [41].

A simplified version of the `PARALLEL_FOR` macro is shown in Listing 4.1. An invocation has the form `PARALLEL_FOR(id,max_value)`. This will iterate the value of `id` for each thread such that combined all values from 0 to `max_value` are covered. If `_USE_BLOCKED_LOOPS_` is defined, the work is distributed into equal-sized blocks. Each thread is given a block and will linearly iterate over the contained indices. Otherwise, the work will be split into blocks of a size equal to the number of threads. Then, each thread will be given that index from each block which matches its thread identifier.

Another important difference between CPU and GPU is that the GPU prefers a full SoA pattern. On CPUs AoS patterns are traditionally used. Partial SoA patterns have been shown to reach high performance on the CPU, but full SoA patterns are problematic [41]. Therefore, we encapsulated all memory accesses into separate functions which transparently perform the SoA conversion if required. When moving data onto a device or back to the host, the same automatism occurs. This ensures the best memory access patterns are used on all devices.

While OpenCL provides vector data types, which the AMD platform uses for vectorization on the CPU, we did not use those in our code. Besides complication of the source code, they would increase the amount of registers required on the GPU which are already a sparse resource. The Intel compiler is able of performing implicit vectorization by combining the work of multiple OpenCL threads into SIMD instructions.

Listing 4.1: A macro to utilize strided loops on GPUs and blocked loops on CPUs. Line-continuation characters have been removed to improve readability.

```
#ifdef _USE_BLOCKED_LOOPS_
#define PARALLEL_FOR(VAR, LIMIT)
size_t _block_size =
   (LIMIT + get_global_size(0) - 1)
   / get_global_size(0);
for(size_t VAR = get_global_id(0) * _block_size;
    VAR < (get_global_id(0) + 1) * _block_size && VAR < LIMIT;
    ++VAR)
#else /* _USE_BLOCKED_LOOPS_ */
#define PARALLEL_FOR(VAR, LIMIT)
for(size_t VAR = get_global_id(0);
    VAR < LIMIT;
    VAR += get_global_size(0))
#endif /* _USE_BLOCKED_LOOPS_ */
```

4.2.5. Utilizing Multiple Devices

CL²QCD currently only supports GPUs within a single host system. All operations are performed from a single host thread and only OpenCL is required to implement parallelism. To utilize multiple GPUs the lattice is split into multiple parts. The GPUs each perform the same operations, each on their part of the lattice.

All objects in the *Physics* package always operate on the whole lattice. Thus, each object from the *Lattice* package contains multiple buffers. One for each of the parts into which the lattice has been split. Methods and classes in the *Physics* package invoke the kernels for each of the buffers and ensure to combine the results, if required.

Volume Splitting

The lattice on which CL²QCD operates is the discretization of a four-dimensional volume. Some fields do actually have more dimensions, but these are not relevant for volume splitting as they are not common to all fields. Examples are the gauge fields and the gauge momenta fields, which have an additional direction dimension. Splitting in these dimensions would not affect all fields and reduce the profit in terms of a reduction in memory requirements.

Even more, in $\not{D}$ the calculation of one entry of the target spinor field requires neighbours in all directions. Therefore, a splitting in the extra dimension direction is not possible in a reasonable way. However, only direct neighbours are required in the four space-time dimensions. Other kernels, like the force calculation, might require second neighbours in space-time. Still the depth to which kernels reach into the neighbouring volume is always limited. Therefore, splitting is performed in space-time.

Currently the mapping from space-time to to the index i_{mem}, specifying the storage location in memory, is performed in the following order: position in x direction, position in y direction, position in z direction, position in time, where the left-most is the fastest running index. This basically gives the following formula for the index. x, y, z, and t specify the position in the corresponding direction. N_x, N_y and N_z specify the extent of the lattice in the corresponding direction. Usually those are all equal, so they can be replaced by N_{space}.

$$i_{\text{mem}} = x + y \cdot N_x + z \cdot N_x \cdot N_y + t \cdot N_x \cdot N_y \cdot N_z \tag{4.1}$$

Usually the slowest running index is used for splitting. However, for lattices with $T >> 0$ this is also the smallest dimension. Therefore, they give the worst boundary to volume ratio and put the lowest limit on the maximum number of GPUs used. One solution to mitigate this problem would be to switch the order of the time and the z direction in this case. Another solution is to split the volume in more than one direction. This does, however, complicate boundary handling. Therefore, it is not done in the initial implementation. This allows to solve the basic implementation problems without having to deal with the additional complexity.

The basic concept does not get changed by this, so the following conceptual work assumes splitting in time direction. This is also the variant implemented currently. Splitting the volume modifies the mapping of indices to memory, as there are now two variants of the variable t. t_{global} indicates the position of a site in the global lattice. t_{local} indicates the position of a site within the part of the problem local to the device. What before was the memory index i_{mem} now becomes the global index i_{global}. The local index i_{local} gives the position within the local problem.

$$i_{\text{global}} = x + y \cdot N_x + z \cdot N_x \cdot N_y + t_{\text{global}} \cdot N_x \cdot N_y \cdot N_z \tag{4.2}$$

$$i_{\text{local}} = x + y \cdot N_x + z \cdot N_x \cdot N_y + t_{\text{local}} \cdot N_x \cdot N_y \cdot N_z \tag{4.3}$$

Given N_{devices} devices, the values of the local indices follow from the global values as follows:

$$N_{t,\text{local}} = N_{t,global}/N_{\text{device}}, \tag{4.4}$$

$$t_{\text{local}} = t_{\text{global}} \mod N_{t,\text{local}}. \tag{4.5}$$

The device on which a specific node is stored is given as follows:

$$i_{\text{device}} = \lfloor t_{\text{global}}/N_{t,\text{local}} \rfloor. \tag{4.6}$$

The inverse of course also can be computed:

$$t_{\text{global}} = i_{\text{device}} \cdot N_{t,\text{local}} + t_{\text{local}}. \tag{4.7}$$

The indices of the other directions—x, y, z—remain unchanged.

There is an important limitation in this concept. $N_{t,\text{local}}$ is the same on all devices. Therefore, the size of the lattice in time direction must be a multiple of the number of devices.

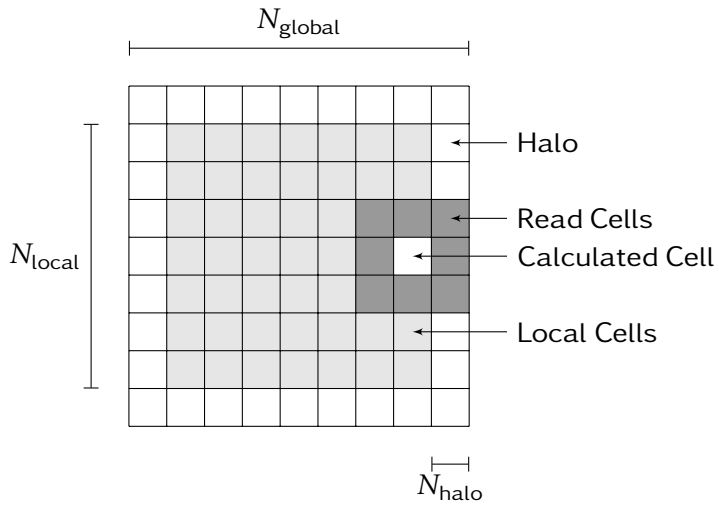

Figure 4.4.: A blur filter reads neighbouring pixels. Thus, a halo of width 1 is required.

Boundary Handling

An additional issue that comes up in multi-GPU computing using volume splitting, is the boundary handling. In an example not related to LQCD this can easily be explained using a blur filter. A blur filter—applied to an image—for each pixel averages the values of the neighbouring pixels. Therefore, if the image is split into multiple local volumes, the PE needs some way to get the neighbours of the outermost pixels of the local volume. Thus, at the boundary an additional area with a width of one element is required to process the image. This is visualized in Figure 4.4. In the following I will call the additional area halo. Its width is given as N_{halo}. The total width of the stored volume is given as N_{global} and the width excluding the halo as N_{local}.

The LQCD calculation utilizes periodic boundary conditions[1]. Therefore, even those sides of the local volume that are on the outside of the global volume require halo cells.

There are many kernels in CL²QCD that do not require any neighbouring sites. This includes reduction kernels. They do require all sites, but for them the results of multiple local calculations can trivially be combined.

$\not{D}$ requires the direct neighbours for calculation of a single result element. The calculation of rectangles and improved fermion forces require the second-nearest neighbours for calculation of a single result. Thus, CL²QCD uses a halo width of two sites by default. The halo width is actually a compile time constant in CL²QCD. Should a later version of the code require third or fourth neighbours, it can easily be changed.

There are two major approaches to store the halo sites. One is to increase the size of the lattice storing the local sites to also include the halo sites. A major advantage of

[1] The time direction is actually anti-periodic. However, in CL²QCD a phase of π/N_t is applied at each node. This evenly distributes the phase over all nodes. Therefore, the implementation is technically periodic.

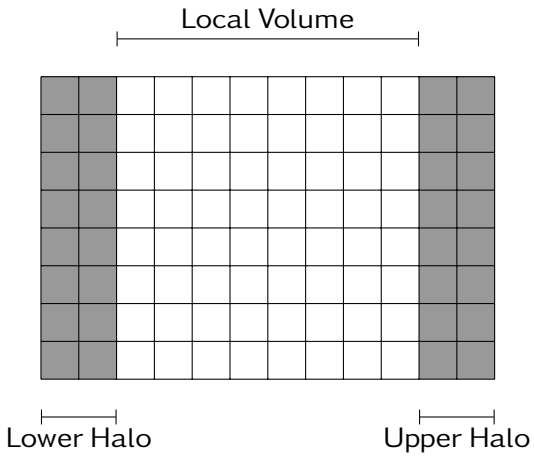

Figure 4.5.: Storage of the halo by extending the lattice. Accesses to the local volume must be offset to account for the halo cells.

this approach is that the number of buffers handled does not change. Also the kernels do not require any changes to their arguments. Only the calculation of the indices into memory changes. It now must take into account the increase in lattice size and potential offsets due to the halo cells.

Given periodic boundary conditions there are two variants of this approach. For one, the part of the halo containing lower neighbours can be stored below the volume local to the device. This is sketched in Figure 4.5. In that case each computation needs to use an offset into the buffer when accessing the local volume. In the memory of the device the halo cells form an actual halo around the local data.

CL^{2}QCD uses the other variant. The lower neighbours are wrapped around and stored above the upper neighbours. This way the indices of the local nodes start at the first element. A major advantage is that high level code has to care less about the halo and only index calculation if affected.

An alternative approach would have been to add buffers containing the halo sites. This way transferring the data is very easy to implement. However, for each access to a site a check whether it is a halo site is required. This causes modifications to the code in more places. In addition, it can lead to branch serialization, which can cause major performance issues on GPUs.

The boundary handling slightly modifies the index calculation for the sites. However, as t is the outermost index this is limited to fields which have more than the space and time dimensions and to neighbour index calculations. In these the actual size of t in memory $N_{t,\text{mem}}$ must be used instead of $N_{t,\text{local}}$.

$$N_{t,\text{mem}} = N_{t,\text{local}} + 2\,N_{\text{halo}} \tag{4.8}$$

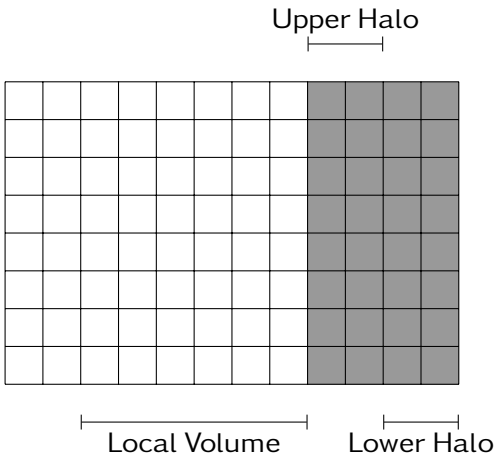

Figure 4.6.: Storage of the halo by extending the lattice and taking advantage of the periodic nature. Accesses to the local volume can completely ignore the volume. Reading the lower halo part must wrap around to the top, which is hidden in the index calculation.

Halo Transfer

After each calculation that updates the sites the halo must be refreshed from the neighbouring devices. As long as the lattice is only split in the major dimension, this is not too complicated even if the offset halo scheme is used. Each device transfers its lower boundary to its lower neighbour, and its upper boundary to its upper neighbour. For two devices this is shown in Figure 4.7. As t is the slowest running index, this is in principle continuous memory. However, things get more complicated if SoA, even-odd preconditioning, and additional dimensions are involved.

In the SoA case, given n storage lanes, n copies are performed. This is sketched in Figure 4.8. Conceptually it looks as if multiple AoS lattices were stored in a single buffer. Utilizing OpenCL's functionality for multi-dimensional memory copies allows to queue the copy using a single command. The padding added for best SoA performance causes the stride between subsequent blocks to not be equal to the volume of local and halo sites.

Even-odd preconditioning is only a special case of an additional dimension. The one additional effect it causes is that only half the sites are actually stored. Therefore, all indices and volumes are halved when accessing the memory. For spinor fields the size of this dimension is one. For gauge field and gauge momenta the size of this dimension is two.

Additional dimensions can have two effects. If they are faster running than the space and time coordinates, they simply cause the sites to be larger. However, in CL^{2}QCD additional dimensions are always slower running. Therefore, for each additional dimension the amount of copies required must be multiplied with the size of this dimension. This is very similar to the effect of the SoA lanes, just within them.

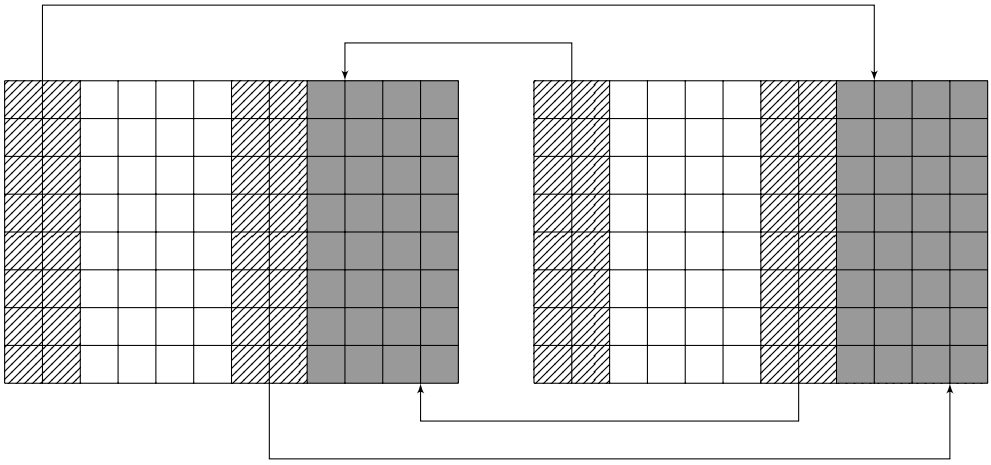

Figure 4.7.: The halo exchange pattern using two devices and AoS storage.

4.3. Optimization

In this section I will show how I applied the optimization techniques described in Chapter 3 to CL^{2}QCD. As $\not{D}$ is the kernel dominating overall performance, I give the most detailed analysis for this kernel. Before, however, I will discuss how data types are stored in memory since this optimization affects all parts of the application. The performance implications of this optimization will then be discussed using the example of the $\not{D}$ kernel.

Optimizing an LQCD code, the net memory bandwidth achieved is the most important metric because LQCD is completely memory bound. This can easily be seen for the $\not{D}$ kernel, which for each site performs 2880 B of memory I/O, while only performing 1632 FLOPS. Thus, the arithmetic density is only about 0.57 FLOP/B while—as Table 2.1 shows—GPUs provide more FLOPS than bandwidth.

4.3.1. Global Memory Storage Formats

As I have shown in Section 3.1 the way data is laid out in memory has significant effects on the achievable memory bandwidth. In Subsection 3.1.2 I have shown that large data types perform significantly worse than smaller ones. The typical upper limit for the size of a well-performing data type is 16 B. The larger types shown in Figure 3.3 show 30 % to 60 % less performance. However, in DP the important types $SU(3)$ and spinor have a size of 144 B and 192 B, respectively. Therefore, those types must be stored in a SoA format.

Table 4.1 shows how such types can be mapped to simple types for SoA storage. The copy benchmark—also used in Section 3.1—can be used to check which type performs best using the corresponding number of SoA lanes required for the QCD types. On the AMD Radeon HD 5870 this are the types of size 16 B. These reach about 90 GB/s. For comparison, using an AoS layout only 55 GB/s can be reached for the $SU(3)$ type.

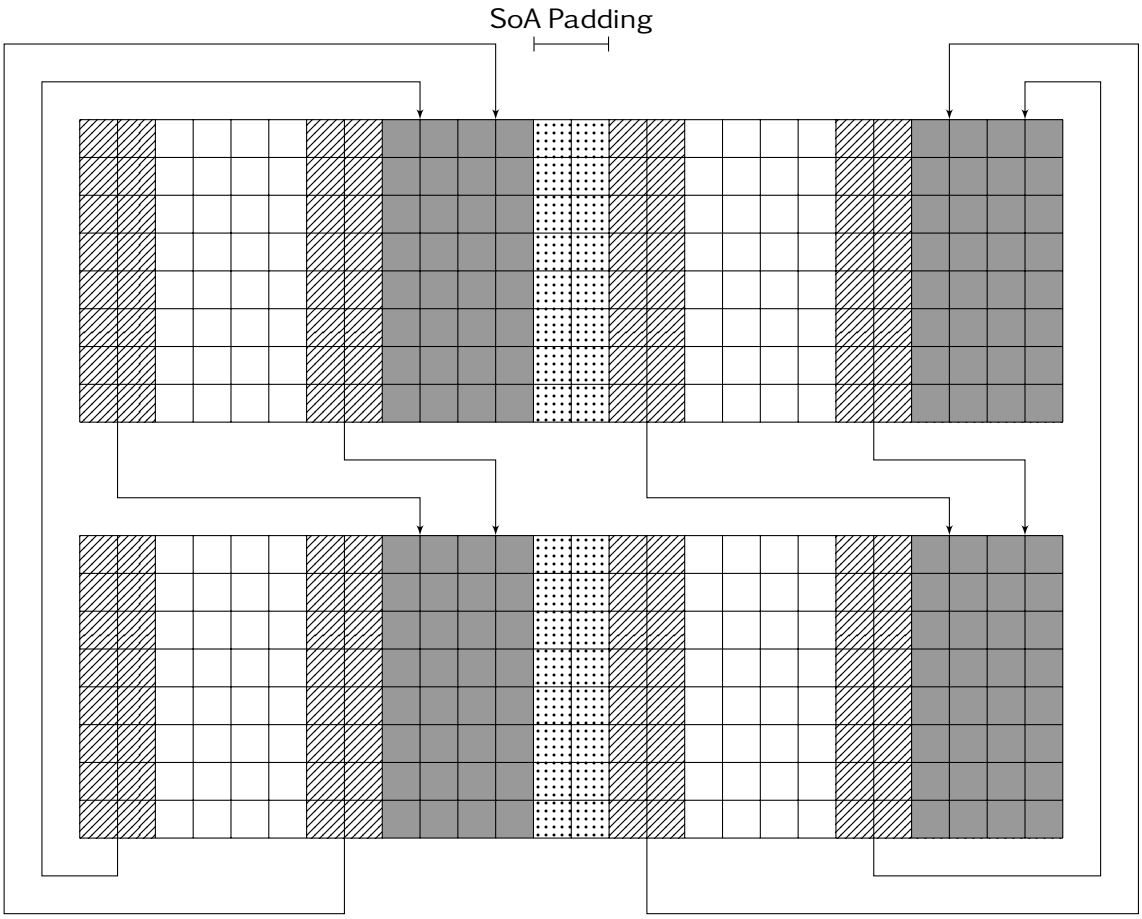

Figure 4.8.: The halo exchange pattern using two devices and SoA storage with two lanes. Contrary to Figure 4.7, the devices are sketched below each other.

Table 4.1.: Possible mappings of LQCD types to SoA storage. The table shows how many SoA lanes are required to store the LQCD type using a given basic type for storage.

	QCD type	
Storage Type	$SU(3)$	spinor
float	36	48
float2	18	24
float4	9	12
double	18	24
double2	9	12
double4	N/A	6

Of course all SoA stride optimizations from Subsection 3.1.5 have been applied. The NVIDIA GPUs show the same characteristic, reaching about 130 GB/s for types of size of 16 B. On the AMD Radeon HD 7970 all types from Table 4.1 show about 160 GB/s when run with the corresponding number of SoA lanes.

The $SU(3)$ and spinor types are internally built from a DP complex type. This type is equivalent to the double2 type and has a size of 16 B. As this size showed to be the best size for a SoA storage type, this type is the best possible base type to store the QCD types on all currently used GPUs. This type does not only provide the maximum memory bandwidth, but it has the additional benefit that the QCD types can be directly constructed from this type. Therefore, it is not required to perform any reinterpreting type conversions, reducing the complexity of the code given to the compiler.

4.3.2. $\not{D}$ Operator

The high-performance $\not{D}$ kernel started from a very naïve implementation that was basically identical to a generic scalar CPU implementation. The major difference was that the loop over the lattice sites was parallelized by spreading it over all threads. Figure 4.9 shows how the net utilized bandwidth of the GPU changed with various versions of the $\not{D}$ kernel during development. The details of each version will be described in the remainder of this section.

Memory Bandwidth Optimizations

The original version of the $\not{D}$ kernel is Version 1 in Figure 4.9. It still used an AoS storage format for the gauge and spinor fields and utilized only about 22 GB/s of memory bandwidth. This is only 14 % of the peak memory bandwidth of the AMD Radeon HD 5870.

Before showing the effect of the bandwidth optimizations suggested in Section 3.1, I want to show the effect of some minor modifications, which are also suggested by the *AMD Accelerated Parallel Processing OpenCL™ Programming Guide (v2.8)* [69] and its

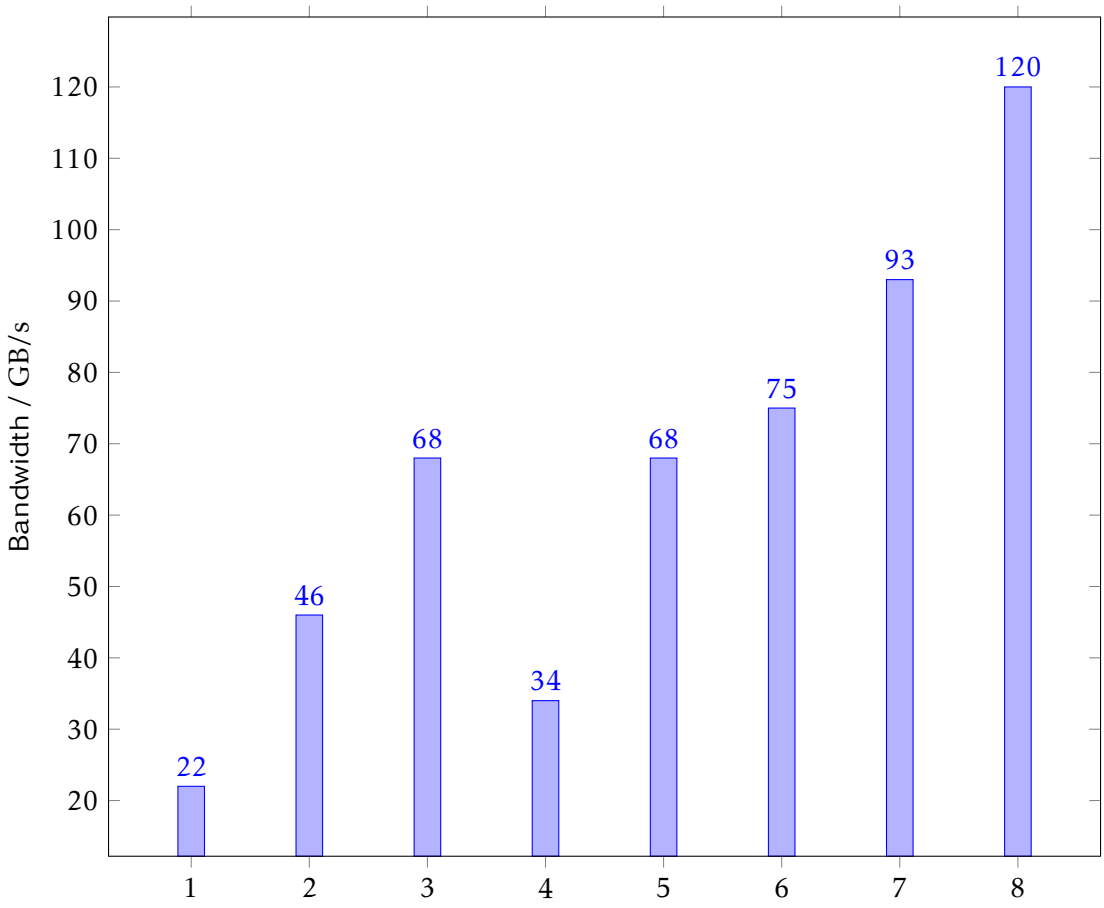

Figure 4.9.: Utilized bandwidth for multiple versions of the D kernel on the AMD Radeon HD 5870.

predecessors. The advantage of these optimizations is that they can be performed with minimal code modification. All the benchmarks for memory bandwidth optimization of the $\not{D}$ kernel on the AMD Radeon HD 5870 were performed using Catalyst 10.7 and 11.11.

Version 2 is a minor improvement over the initial version. Utilizing the texture cache—which on hardware by AMD can be done with minimal code modification by declaring all pointers as `const restrict`—provides a significant speed-up. However, the achieved 46 GB/s are still far from the AMD Radeon HD 5870's theoretical bandwidth limit of 155 GB/s.

As Version 3 shows, another speed-up can be reached by specifying a proper alignment for the AoS data types. The key is to use the largest possible alignment that is still a divisor of the data type's size. If no alignment is specified the compiler will use the alignment of the smallest contained data type, resulting in superfluous memory fetches. Without any additional specifications the $SU(3)$ type would only be aligned to 8 B. This is the size of double, from which the complex type is built. The $SU(3)$ type itself is then constructed from this complex type and inherits the 8 B alignment. Therefore, the complex type should be explicitly aligned to 16 B. This is also the best alignment for the $SU(3)$ type, as its size cannot be divided by 32 B without a remainder. While the number of fetches in some cases could be reduced even further by using an alignment that is larger than the data type, it turns out that the additional bandwidth required overcompensates the benefit. Using proper alignment of all types the code reaches 68 GB/s.

Utilizing the texture cache and properly specifying the alignment for all types is insufficient to optimize the $\not{D}$ kernel. It does, however, only require minimal code modification and should, therefore, always be applied. In codes with higher arithmetic density it might already provide all the speed-up required to be no longer limited by available memory bandwidth.

As shown in Subsection 4.3.1, to achieve maximum performance an SoA layout is required. But, when simply applying the SoA storage to the $\not{D}$ kernel, performance collapses to the 34 GB/s shown for Version 4.

The SoA performance issue is caused by the way the gauge fields sites are mapped to the memory. Figure 4.10 shows the implications of multiple implementation variants. It focuses on a segment of 256 B of memory, as 256 B are the maximum amount of data the AMD Radeon HD 5870 can read from memory in one request. The grey-coloured memory contains the data requested by multiple lock-stepped threads. Any white-coloured memory read cannot be used to fulfil a data request by the current group of threads and is discarded.

The first row shows the original layout. Sites are stored in an AoS fashion. The four links originating at each site are stored next to each other. Given proper caching, in the AoS case this does not cause a major problem. As each $SU(3)$ has a size of 144 B, a large portion of a 256 B access to memory can be used by the kernel and little bandwidth is wasted, even if neighbouring threads do not read neighbouring elements. The latter is the case in the $\not{D}$ kernel. Neighbouring threads will always read neighbouring sites, and of those they will read the links pointing in the same direction. This is why the

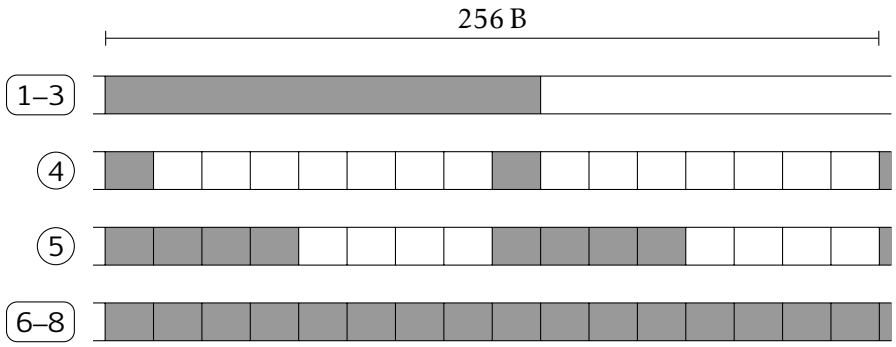

Figure 4.10.: The graph shows multiple layout variants for storing the gauge field. Each row represents a different storage variant. The label shows the kernel version in Figure 4.9 and the text using the storage variant shown in this row. The grid spacing denotes the size of elements used for access. Elements that are accessed by lock-stepped threads in the $\not{D}$ kernel are given a darker colour. Thus, white elements would be fetched when reading 256 B from memory but discarded afterwards.

second part of the segment is not used by the neighbouring thread. It does not contain the neighbouring site but a link into a different direction originating from the same site.

The second row of Figure 4.10 shows why the performance collapses if SoA is naïvely added to the implementation. Now each thread will only request 16 B, and there is a stride of eight elements, or 128 B between the accesses by neighbouring threads. The stride of eight elements stems from even-odd preconditioning. If neighbouring threads would actually read neighbouring sites, then the stride would be four elements as there are four dimensions resulting in four links per site. However, using even-odd preconditioning, only even or odd sites are taken into account which doubles the stride. Therefore, if the memory controller requests a block of 256 B from memory, only 32 B will actually be used, wasting nearly 90 % of the available bandwidth.

Row 3 shows the effect of making μ—the direction in which a link is pointing—the outermost index. Now neighbouring threads nearly access neighbouring sites. The stride is only 32 B. That means, half the memory read will actually be used. This gives a performance of 68 GB/s for Version 5 in Figure 4.9. To completely solve the issue, in Version 6 I made the even-oddness an own index, removing the interleaving of even and odd sites. It is not possible to simply leave out half of the sites, as all sites will be read by the $\not{D}$ kernel. This leads to row 4, where neighbouring threads read neighbouring links, providing maximum memory bandwidth utilization. Of the 256 B fetched via one reading access to memory, all 256 B are used.

The 75 GB/s of memory bandwidth utilized by Version 6 of the $\not{D}$ kernel are still far from the peak memory bandwidth of the AMD Radeon HD 5870. The difference to peak performance is, however, not caused by the way the memory is accessed but by

register spilling. Combined with the register optimizations shown in Section 4.3.2, the optimizations shown in this section allow to utilize more than 120 GB/s on the AMD Radeon HD 5870, which is more than 70 % of the peak. On the AMD Radeon HD 7970 close to 200 GB/s are utilized, which is nearly 80 % of the peak.

Register Optimizations

In Section 4.3.2 I showed the optimizations that are required to efficiently utilize the bandwidth provided by the GPU memory. But, as the $\mathbb{D}$ kernel was spilling registers it did not reach the full performance of the AMD Radeon HD 5870. As with the bandwidth optimization, I performed all register optimizations for the AMD Radeon HD 5870 using Catalyst 11.11.

The copy benchmark in Section 3.2 shows that full bandwidth can be achieved using only 64 threads per CU when copying float4 values. However, the float type requires at least 256 threads per CU for best performance. In addition, most floating-point operations have a latency of four cycles on the AMD Radeon HD 5870 [69]. Therefore, I aimed at running at least 256 concurrent threads per CU. This imposes a limit of 62 registers per thread.

Allowing the compiler to utilize scratch registers makes it possible to run 256 concurrent threads per CU. I implemented this in Version 7, which uses 62 normal registers and 10 scratch registers. This results in a bandwidth utilization of 93 GB/s.

When investigating the register usage I noted that the amount of registers used by a kernel is often more related to code complexity than to the actual working set size. Version 6 of the $\mathbb{D}$ kernel required 73 registers per thread. Adding more scopes to the kernel, to explicitly release variables no longer used, actually increased the register usage by 13 registers per thread. On the other hand, always directly using `get_global_id` instead of storing it in a variable at the beginning of the kernel reduced the number of used registers by 8.

The $\mathbb{D}$ kernel performs the same operation once in each dimension, summing up the results in the destination site. This allows to study the register requirements of partial implementations, e.g. only operating in one or two directions. In theory, all registers used to perform the operation in one direction can be reused to perform the operation for the next direction. Only when going from a single to multiple directions the registers to keep the intermediate result should cause some increase in register usage. In reality though, while the kernel for a single direction requires only 50 registers and a two direction kernel requires 67, the kernels implementing three and four directions require 69 and 73 registers, respectively.

The obvious solution is to make sure that the implementation uses exactly the same code and the same registers for each direction. This can be achieved by using a loop over the directions which may not be unrolled. However, each direction requires multiplication with a different Gamma matrix. Therefore, each direction was originally implemented as a separate function, in which the matrix multiplication was coded in.

As the Gamma matrices are the same for all threads one option is to store them in constant memory and conventionally multiply the matrix with the intermediate result.

It shows, however, that this approach actually increases the register requirements and performs worse than the original code.

A second approach is to create one function that unifies the four separate functions with hard-coded matrices. The unified function ensures each variable used is only declared once for all directions. To implement the different matrices it branches for the hard-coded matrix multiplication. But, as the branch condition will always be the same for neighbouring threads it should not cause performance issues. This is Version 8 of the $\not{D}$ kernel. It requires only 54 registers and operates at 120 GB/s, nearly 80 % of the peak bandwidth.

The 120 GB/s reached show that it is not necessary to further optimize the registers to enable more than 256 threads per compute unit. But it also shows that for the $\not{D}$ kernel less threads per CU are not sufficient, just as it was for the float copy kernel.

The AMD Radeon HD 7970 does not require any specific optimization to achieve high performance in the $\not{D}$ kernel. However, it did take AMD multiple driver versions until their compiler was capable of properly handling the high complexity of the code implementing the $\not{D}$. For this purpose I created a standalone version of the $\not{D}$ kernel. This version contains all kernel code in a single file. In addition it provides a Python script that can properly configure the code for multiple lattice sizes, benchmark it and test the result for correctness. This version was provided to AMD to enable them to test their compiler with a real code of this complexity.

Figure 4.11 shows how the performance of the code developed as Catalyst was updated to newer versions. On Catalyst 12.3 the $\not{D}$ achieves 165 GFLOPS/s. Ironically that performance is only achieved in the classic implementation of the $\not{D}$ kernel, which shows register spilling on the AMD Radeon HD 5870. The other variant, where all directions are merged into a single branching function, only achieves about 115 GB/s. This is slower than the much older AMD Radeon HD 5870. The major drawback of this old Catalyst version was that it did not officially support the AMD Radeon HD 7970 and failed on some other kernels required for a full HMC.

Catalyst 12.4 was the first to officially support the AMD Radeon HD 7970. Using this driver both kernel variants run into massive register spilling. This limits the performance of the $\not{D}$ to 20 GB/s, a loss in performance of 80 %. Less than 10 % of the peak bandwidth are utilized.

The first official driver to fix the regression was Catalyst 13.6.[2] Using this driver the $\not{D}$ kernel is able to utilize 225 GB/s of memory bandwidth on the AMD Radeon HD 7970. This even exceeds the performance of the benchmarks in Chapter 3. There are two effects which cause this. Firstly, the benchmarks in Chapter 3 measured copy performance while the $\not{D}$ kernel reads much more data than it writes. Secondly, the larger working set of the $\not{D}$ kernel can result a larger number of memory transactions that are in flight concurrently. Despite the major architectural change from the AMD Radeon HD 5870 to the AMD Radeon HD 7970, this does not require any additional

[2] AMD has continually been providing drivers containing a fix for the performance regression since November 2012. But, until Catalyst 13.6 none of these drivers was officially released and publicly available.

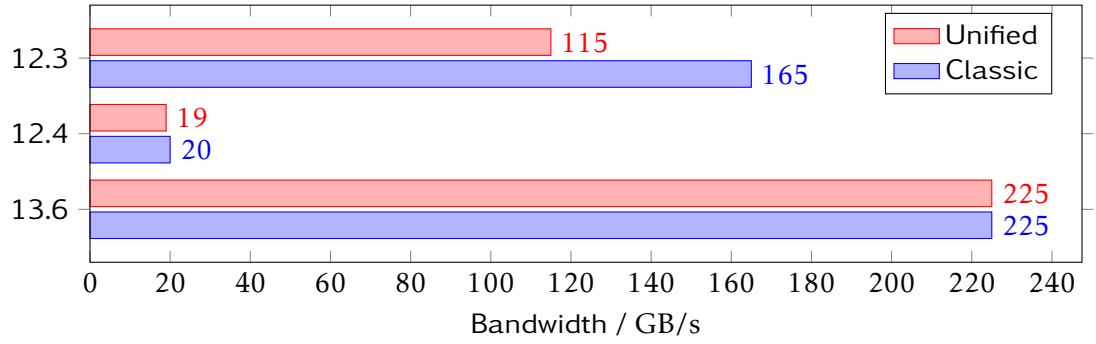

Figure 4.11.: $\not{D}$ kernel performance for a $24^3 \times 12$ lattice on the AMD Radeon HD 7970 using multiple versions of the Catalyst driver. The classic kernel uses distinct functions to implement the calculation for each direction. The unified kernel uses a loop over the directions, using branching inside the unified function to implement the direction-specific parts of the calculation. Driver versions in between 12.4 and 13.6 provided the same performance as Version 12.4. Catalyst 13.8 provides the same performance as 13.6. For the AMD Radeon HD 5870 and the AMD Radeon HD 6970 no performance variations were observed.

optimizations. As a matter of fact, the optimization of merging the different directions into a single function is no longer required. It no longer provides better performance than the classic implementation.

Small Lattice Optimizations

When my colleague Christopher Pinke used CL^{2}QCD for studies based on small lattices [91], he quickly noted $\not{D}$ to perform very badly for these. At that point in time Catalyst 12.4 was up to date. As the legacy code curve in Figure 4.12 shows, the $\not{D}$ kernel is hardly able to utilize any of the AMD Radeon HD 5870's bandwidth. It is unable to reach even 10 GB/s, except for lattices with a spacial extent of 16 sites.

Using Catalyst versions past 12.4, the $\not{D}$ kernel requires 865 scratch registers if the spacial extent of the lattice is not a multiple of 16. Lattices with a spatial extent of 16 sites do not require any scratch registers. All these lattices are using the same code. Only the lattice size is given as a compile time constant, seemingly triggering the compiler to use different optimizations depending on its value.

I was able to solve the performance issue by a minor modification to the way the loop over all sites is implemented. The legacy $\not{D}$ kernel uses the following code to implement the loop:

```
1 size_t _global_size = get_global_size(0);
2 for(size_t VAR = get_global_id(0);
3     VAR < LIMIT;
4     VAR += _global_size)
```

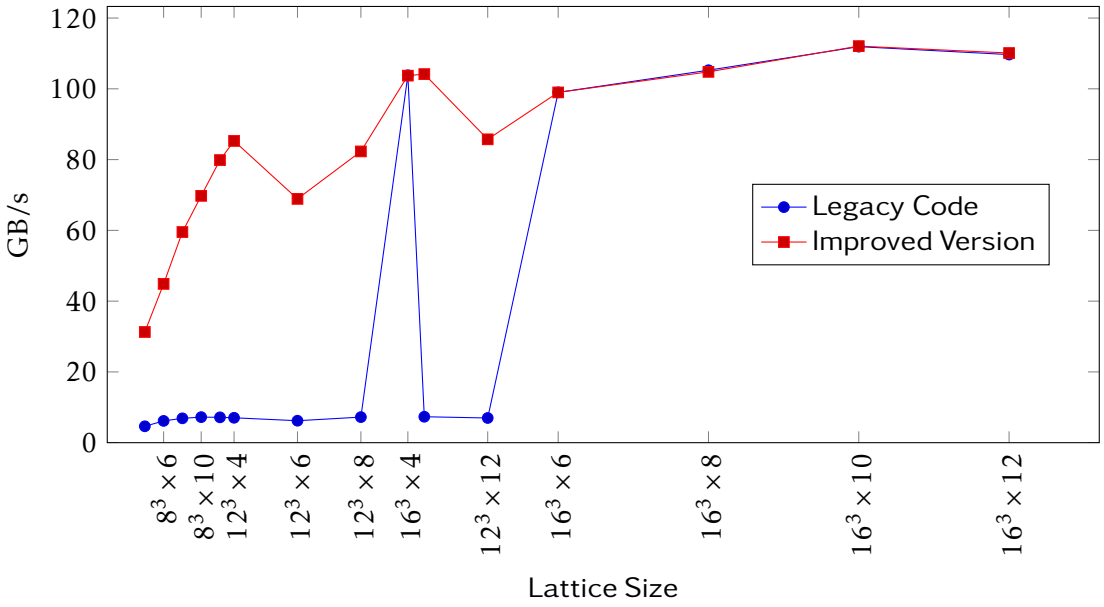

Figure 4.12.: Bandwidth utilization of the 𝕀𝔻 on the AMD Radeon HD 5870 using Catalyst 12.4 for small lattices.

The performance issue is solved by moving the call to `get_global_size(0)` inside the declaration of the loop. Using the following definition, the 𝕀𝔻 kernel does not require any scratch registers on small lattices, despite the fact that it only reduces the number of declared variables by one:

```
for(size_t VAR = get_global_id(0);
    VAR < LIMIT;
    VAR += get_global_size(0))
```

Note, that the function `get_global_size` is defined by the OpenCL standard. As the size of a grid is constant once a kernel has been launched it is basically a constant expression. Therefore, these two code variants specify exactly the same behaviour. In theory, a compiler should even be able to transform between these two variants by itself, using some cost estimation to choose the better one.

It turns out that using the second code variant will cause performance issues if the spatial extent of the lattice is a multiple of 16. Therefore, 𝕀𝔻 now uses a different formulation of the loop depending on the lattice site. The complete definition of the macro used to implement parallel loops on GPUs is given in Listing 4.2. In Figure 4.12 this improved version shows much better bandwidth utilization. It reaches 80 GB/s even for small lattices.

Listing 4.2: The macro used to implement parallel for loops on GPUs. This macro replaces the GPU part of the macro shown in Listing 4.1. Contrary to Listing 4.1 it does not differntiate between device types but between problem sizes. Again, line-continuation characters have been removed.

```
#if (NSPACE / 16 ) * 16 == NSPACE
#define PARALLEL_FOR(VAR, LIMIT)
size_t _global_size = get_global_size(0);
for(size_t VAR = get_global_id(0);
    VAR < LIMIT;
    VAR += _global_size)
#else /* NSPACE % 16 == 0 */
#define PARALLEL_FOR(VAR, LIMIT)
for(size_t VAR = get_global_id(0);
    VAR < LIMIT; \
    VAR += get_global_size(0))
#endif /* NSPACE % 16 == 0 */
```

4.3.3. Inverter

The inversion of the fermion matrix is the computational hotspot of both operator evaluation and the HMC algorithm. Inside the inversion $\not{D}$ is the most expensive operation. However, as we know from Amdahl's Law [92] we cannot expect the speed-up in $\not{D}$ to completely transfer to a speed-up in the inverter. The optimized $\not{D}$ provides about 70 GFLOPS on the AMD Radeon HD 5870. This translates into a performance of about 38 GFLOPS for both inverters, the CG and the BiCGSTAB. This is slightly about half of the $\not{D}$ performance.

Effects of Generic Optimizations

The inverter performance of 38 GFLOPS is reached without any specific optimizations, but some generic optimization guidelines have to be followed. Obviously, no data layout conversions may be performed inside the iterative loop of the inverter. The $\not{D}$ kernel accesses 2880 B of memory for each site. Any conversion of a spinor field reads and writes one spinor per site, resulting in 384 B of memory I/O. A conversion of the gauge field reads and writes four $SU(3)$ matrices per site, resulting in 576 B of memory I/O. That are 20 % of the I/O performed by the $\not{D}$ kernel. As both problems are limited by memory bandwidth, conversions would, therefore, take about 20 % of the execution time of the $\not{D}$ kernel.

Avoiding memory layout conversions automatically means that all other kernels use the same optimized memory layout as the $\not{D}$ kernel. As they are all of lower complexity than the $\not{D}$ kernel, register spilling is not an issue for them and on the AMD Radeon HD 5870 they can utilize memory bandwidth in the order of 100 GB/s.

As shown in Section 3.4, communication between the host and the device is an important optimization issue. The inverter gives an academic, but rather illustrative, example of this. It invokes $\not{D}$ twice in a row, once to update the even sites, and once to update the odd sites. Adding a call to `clFinish` will prevent the CPU from sending the second $\not{D}$ invocation to the GPU before it has been notified of the completion of the first invocation. In this case, sprofile shows the idle time of the GPU in between the $\not{D}$ kernel executions to be as long as the processing of the $\not{D}$ kernel takes for a lattice of $16^3 \times 12$ sites. Thus, for a lattice of this size a useless call to `clFinish` effectively halves the performance of the $\not{D}$ implementation. Calls to `clFinish` are not required in between multiple calls send to the same OpenCL command queue, as those are executed subsequently by the device.

Buffer Operations

The reduction kernels required by the inverters show a similar problem. As there is no global synchronization in OpenCL, reductions are implemented via two-pass kernels. Therefore, they require a temporary buffer to store the intermediate results. Dynamically creating this buffer every time a reduction is performed costs about 3 GFLOPS when inverting a $24^3 \times 8$ lattice on the AMD Radeon HD 5870. However, from a software-architectural point of view the temporary buffer should be encapsulated in the code implementing the proper invocation of the reduction kernels and should not be visible for other code. For CL²QCD I solved this problem by attaching the temporary buffer to the code object owning the reduction kernel. There its lifetime is scoped to be the same as the kernel objects. This has an additional advantage. If reductions are queued faster than they can be executed by the GPU, memory usage would creep up if each reduction allocated a separate temporary buffer.

The same issue exists with some compound operators used in the inverter. These require a temporary fermionic field to store intermediate results. Creating these temporary fields ad-hoc—every time the compound operator is called—costs 7 GFLOPS to 8 GFLOPS of performance when inverting a $24^3 \times 8$ lattice on the AMD Radeon HD 5870. As the temporary fields are rather large and the size of the GPU memory is limited, these buffers require a different treatment than the reduction buffers. Therefore, these operators were implemented as callable objects. Creating such an object when entering the solver will allocate the temporary fields. Invocation of the object will then execute the required kernels without any overhead for buffer management. This way, the costly buffer allocation on the GPU is only performed once, no longer affecting overall inverter performance. Finally, when the inversion finished and the object representing the compound operator goes out of scope, the buffer will be released and the memory can be reused for other data.

The inverter requires copying buffers containing only a single complex value within the GPU in each iteration. Using Catalyst 12.6, sprofile showed these copies to take about 800 µs on the AMD Radeon HD 5870, and even 5 ms on the AMD Radeon HD 7970. For comparison, on the AMD Radeon HD 5870 the $\not{D}$ kernels requires about 1400 µs to process a $24^3 \times 8$ lattice. A simple scalar division only takes about 10 µs. The

whole issue seems to be a profiling artefact. Yet, to avoid any potential performance issues CL^{2}QCD now automatically uses a simple kernel to perform any on-device copies for buffers containing exactly 16 B.

Residual Checking

The above mentioned performance penalty when synchronizing to the GPU does also impact the residual check required in any iterative solver. This residual check is required to determine whether the solver has reached a sufficient solution and the iteration can be stopped. Before evaluating the residual value the CPU must ensure that the GPU has stored it. Thus, the synchronization cannot be skipped.

One potential way to mitigate the performance penalty is to utilize an asynchronous transfer of the residual value from GPU to CPU memory. This way the GPU can already continue on the next iteration while the CPU waits for the completion of the transfer to be signalled. The only overhead imposed by this is an additional iteration of the solver, as the next iteration has already been queued on the GPU when the CPU evaluates the residual.

Using Catalyst 12.6, this approach does not show any speed-up on the AMD Radeon HD 5870 when performing the transfer in the same command queue as the kernel executions. As shown in Section 3.4 the actual transfer takes less then 10 µs. Therefore, I have not tried using a second command queue to perform the transfer in parallel to the computation. In that case an additional synchronization in between the command queues would be required. This should worsen and not lessen the problem.

Another approach is to only check whether the residual value is below the termination threshold every Nth iteration. This way the synchronization overhead is incurred less often. For realistic input values the inverter performs thousands of iterations to get to a solution. Therefore, even values such as $N = 50$ will only cause a compute time overhead in the single digit percentage range. Especially, as on average only $N/2$ additional iterations will be performed.

Figure 4.13 shows how this approach affects the performance of the CG inverter. On the AMD Radeon HD 5870 a speed-up of up to 15 % is observed. This overcompensates the overhead imposed by the additional number of iterations. For an inversion that requires 1001 iterations, the time-to-solution is reduced by 12 % even though 1050 iterations are performed.

On the AMD Radeon HD 7970 the effect is even larger, at least when using older drivers. Using the standard approach of checking the residual after each iteration, performance is worse than on the AMD Radeon HD 5870 despite the much higher performance in the individual kernels used. Checking the residual only every ten iterations improves the performance by 78 %. Performance doubles if the block size exceeds 50 iterations. Even in the hypothetical inversion of 1001 iterations the time-to-solution nearly halves. It is reduced by 48 %.

As the performance curves flatten for block sizes of more than ten iterations, CL^{2}QCD by default uses a block size of ten iterations. This ensures no slowdown occurs when performing inversions that require only a few iterations. For problems that are known

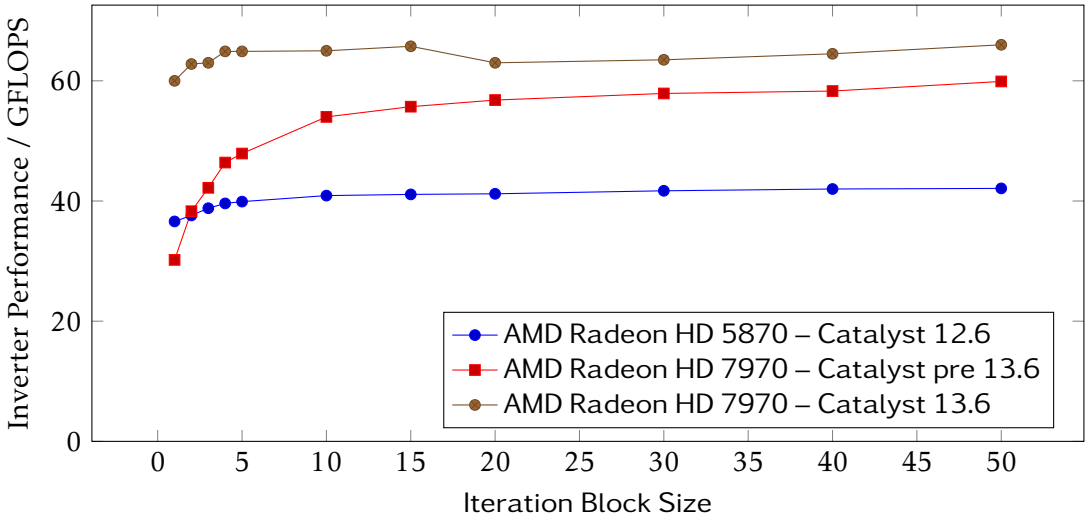

Figure 4.13.: Performance of the CG inverter for a $24^3 \times 8$ lattice when checking the residual only every N iterations.

to require a few thousand iterations the block size can be adjusted when invoking CL²QCD.

A further optimization would be to make the iteration block size depending on the residual value. Thus, the inverter would run at maximum speed for most of the iterations but not overshoot in the end. However, since Catalyst 13.6 the effect has mostly vanished, thus this optimization is now of less importance.

4.3.4. Hybrid Monte Carlo

The performance of the HMC algorithm is dominated by the inverter. For the Z12 set-up used in Section 5.1 the inverter makes up for about 90 % of the overall execution time. Given an average HMC step duration of about 50 min, only 5 minutes are spent in the remaining parts of the HMC.

The time spent in the inverter does depend on the pion mass and the gauge field used. For larger pion masses and for cold configurations the time spent in the inverter reduces while the remaining parts of the HMC stay constant in execution time. However, the physical interest is in low pion masses and thermalized configurations making the performance of such cases less relevant.

Of the approximately 5 min spent outside of the inverter about 80 % are spent calculating the gauge force. On Tahiti based GPUs this calculation originally performed much worse. Given its large contribution to the non-inverter runtime of the HMC, this directly affected overall performance.

The gauge force kernels have a similar structure as the $\not{D}$ kernels. They aggregate a result value by combining multiple ingredients that follow a common pattern, too. Again, the compiler fails to properly reuse the registers, leading to register spilling.

Note, that while the $\not{D}$ kernel works on the links and sites of the lattice, the gauge force operates only on the links. While the $\not{D}$ kernel emits one result per site, the gauge force calculation emits one result per link.

For the kernel *gauge_force*, which is the simpler of the two gauge force kernels, the effect is only minor. The largest effect is caused by proper coalescing of the memory accesses. In the original implementation neighbouring threads work on geometrically neighbouring links. As for the $\not{D}$ kernel the storage of the links has been reordered—storing even and odd sites separately—this leads to a suboptimal memory access pattern. This can be solved by reordering the association of threads to links, such that neighbouring threads operate on neighbouring sites in memory. Reordering the memory would not solve the problem, as in that case $\not{D}$ performance would collapse. This improves the performance on the AMD FirePro S10000 from 130 GB/s to 183 GB/s. This can, however, only be observed on lattices of a size of $32^3 \times 12$, as they are used in the Z12 set-up. On a smaller $16^3 \times 12$ lattice the kernel manages to utilize 180 GB/s on the AMD FirePro S10000, even in the original version.

While the original version requires 916 scratch registers, this does not have a huge impact on the utilized memory bandwidth. Dedicating an own thread for each link solves the problem. In that case no scratch registers are used. Originally the code looped over the links. Such, the number of threads did not have to match the number of links in the lattice. Utilizing this optimization the performance on the AMD FirePro S10000 improves slightly to 195 GB/s for a $32^3 \times 16$ lattice. Using this optimization, it is important to start the kernel with as many threads as there are links. Processing the links by invoking the kernel multiple time with the same number of threads used in the original version drops the utilized bandwidth to 111 GB/s.

The more complex kernel *gauge_force_tlsym* requires 5488 scratch registers in its original version. Exchanging the loop over the links for a dedicated thread per link as above is no sufficient solution for this kernel. This limits the utilized bandwidth to 77 GB/s for both lattice sizes.

Simplified versions of the kernel—containing only a subset of the six ingredients required for the full calculation—show that a kernel using two ingredients takes more than three times the execution time of a kernel using only one ingredient. From the two-ingredient kernel the code scales pretty straight to the three ingredient kernel, which takes about 50 % longer. The complete kernel using all ingredients takes about three times as long as the two ingredient kernel. While this is expected it is more than twice as long than executing kernels for each ingredient individually.

Therefore, I split the kernel into six smaller ones. Each performs the calculation of one of the ingredients. Despite the additional overhead of writing and reading intermediate results, this allows for a net bandwidth utilization of 171 GB/s for a $32^3 \times 12$ lattice on the AMD FirePro S10000. This reduces the execution time by 55 %.

Overall, in the Z12 set-up CL^{2}QCD spends a negligible 40 s per step of the HMC algorithm in the *gauge_force* kernel. The more complex *gauge_force_sym* calculation requires about 200 s, which is about 7 % of the overall HMC step duration. The optimization of this part of the HMC reduces the execution time by about 4 min of the typical 50 min an HMC step takes for this set-up.

4.3.5. Multi-Device

Utilizing multiple devices adds another dimension to the performance optimization problem. The $\not{D}$ kernel, which makes up most of the computation, requires an up to date halo. Therefore, the time required to update the halo affects the net $\not{D}$ performance.

First, I will estimate the effect of the halo transfer onto the $\not{D}$ performance. To simplify the formulas I use the following definitions:

$$V_{\text{space}} = \frac{1}{2} N_x \cdot N_y \cdot N_z, \tag{4.9}$$

$$V_{4d} = N_t \cdot V_{\text{space}}. \tag{4.10}$$

The factor one half in the spatial volume stems from the even-odd preconditioning.

The time t_2 to execute the $\not{D}$ kernel on two devices is given by:

$$t_2 = t_{\text{calc},2} + t_{\text{comm},2}. \tag{4.11}$$

Here $t_{\text{calc},2}$ is the time it takes to calculate $\not{D}$ on two devices. $t_{\text{comm},2}$ is the time it takes to communicate the halo in between two devices and communication and computation are not overlapped. Assuming a constant speed for $\not{D}$, the following can be assumed:

$$t_{\text{calc},N} = \frac{t_{\text{calc},1}}{N}. \tag{4.12}$$

The problem sizes are always in the plateau of the $\not{D}$ kernels performance, such that linear scaling can be assumed if the performance estimation is not expected to be exact to more then 10 %. This leads to the following formula for $t_{\text{calc},2}$:

$$t_{\text{calc},2} = \frac{V_{4\text{d}} \cdot 1362\,\text{FLOP}}{2 \cdot P_1}. \tag{4.13}$$

Here, P_1 is the performance on a single GPU and the 1362 FLOP are the number of floating point operations required for each site of the output volume. In the following $P_1 = 100\,\text{GFLOPS}$ is used.

The volume of each halo is given by:

$$V_{\text{halo}} = V_{\text{space}} \cdot N_{\text{halo}}. \tag{4.14}$$

This leads to the total volume transferred on N devices:

$$V_{\text{halo},N} = 2\, V_{\text{halo}} \cdot N. \tag{4.15}$$

Here the two stems from the fact that one halo needs to be sent to the next GPU, and one halo needs to be sent to the previous GPU. Using the aggregate bandwidth B_N leads to the time required to perform the communication.

$$t_{\text{comm},N} = \frac{V_{\text{halo},N} \cdot 192\,\text{B}}{B_N}. \tag{4.16}$$

The performance P_2 to expect on two devices can be estimated as follows:

$$P_2 = \frac{t_2}{t_1} P_1. \tag{4.17}$$

This formula can now be used to estimate the performance of a $24^3 \times 128$ lattice. For such a lattice the size of one halo is about 2.5 MiB. At this size the fastest copy method transfer I found in Subsection 3.4.3 was the standard buffer copy. It provides about 5 GiB/s of aggregate bandwidth. The single-GPU performance is approximately 100 GFLOPS. Using the initial implementation this results in an estimated $\not{D}$ performance of 155 GFLOPS. This matches perfectly with experimental result of 155 GFLOPS on SANAM.

DirectGMA provides about twice the bandwidth between devices. This could push the performance to about 174 GFLOPS. However, it will not work properly if using all four GPUs on SANAM.

As noted before, $\not{D}$ only requires a halo depth N_{halo} of 1. Halving the halo depth has the same effect on performance as doubling the bandwidth. Of the 174 GFLOPS expected, 168 GFLOPS are achieved on SANAM.

Therefore, CL^{2}QCD only updates the part of the halo required by the next kernel. For this, the update of the halo is not triggered by each kernel that modifies data. Whenever a kernel modifies data the corresponding object is marked as dirty. Then, a kernel will require a halo update for this object before it will read halo data. In the inverter, the only kernel requiring halo cells is $\not{D}$. Since $\not{D}$ requires only the next neighbours, the entire inverter never performs a full halo update, but only next neighbours are updated on demand.

Another speed-up can be gained by executing $\not{D}$ on the inner cells—those cells which do not require data resident in the halo—while the halo update is performed. In theory this should provide a perfect speed-up to 200 GFLOPS even employing the conventional transfer mechanism. However, on SANAM only 180 GFLOPS are observed. There are two things limiting performance in this case. The synchronization in between the different command queues has some latency. In addition, the boundary data must be copied to temporary buffers before the transfer is performed. Only then, the $\not{D}$ kernel can start to process the inner cells. Once the transfer has been performed, the transferred data must be copied to the halo buffer. After that the boundary cells can be computed. While the on-device copy performs at more than 200 GB/s, combined with the synchronization overhead it causes enough delay to limit the speed-up to 80 % instead of the desired 100 %.

The same calculations also show that a $32^3 \times 12$ lattice cannot receive significant speed-up using parallelization in time direction. However, parallelization in space direction promises perfect speed-up. Therefore, this kind of parallelization is a prime candidate for future work.

To summarize, on a $24^3 \times 128$ lattice the initial code version achieves 155 GFLOPS using two GPUs and 224 GFLOPS using four GPUs for $\not{D}$. Reducing the halo size—using $N_{\text{halo}} = 1$—provides 168 GFLOPS on two GPUs and 263 GFLOPS on four GPUs. Over-

lapping communication and computation—still using $N_{\text{halo}} = 1$—provides 180 GFLOPS on two GPUs and 331 GFLOPS on four GPUs.

Adding DirectGMA improves $\not{D}$ performance to 193 GFLOPS on two devices. Then, even the $32^3 \times 12$ lattice can slightly profit from a second GPU, achieving 130 GFLOPS. The four-GPU case involves QPI. As a result, the performance of the $24^3 \times 128$ lattice on four GPUs drops to 161 GFLOPS. This is less than half the performance achieved using the conventional method.

As DirectGMA does not perform well through QPI, CL^{2}QCD benchmarks the transfer between devices on start-up and chooses the faster method. This results in a hybrid halo update method which uses DirectGMA for transfers within a single AMD FirePro S10000 and the conventional method for transfers in between the two AMD FirePro S10000. Using DirectGMA the data of the transfers within the AMD FirePro S10000 does not leave the GPU. This reduces the load on the shared uplink to CPU memory, leaving additional available bandwidth for the transfers in between the two AMD FirePro S10000 transfers. However, there seems to be some interference in between the two transfer methods. On a $32^3 \times 64$ lattice the hybrid method only provides about two thirds of the performance of the conventional overlapped method. However, on lattices with a sufficiently large size in time dimension the hybrid method provides a speed-up of about 5 %. The performance of the $24^3 \times 128$ lattice improves to 356 GFLOPS. This is a speed-up of 59 % over the initial code version with $N_{\text{halo}} = 2$, which achieved 224 GFLOPS.

Given a fast multi-GPU $\not{D}$ implementation, the reductions become a performance problem in the inverter. In the single-GPU code the reduction writes its result to GPU memory, from where the next kernel can use it without any host-side synchronization. In the multi-GPU code, however, the results from each GPUs must be collected to get the global result. Afterwards the result is written back to all GPU, such that the next kernel can use it.

In Section 3.4 I showed that for small buffers less latency is incurred if the memory is mapped from the GPU into the CPU address space. Thus, in multi-GPU mode the *Scalar* class will attempt to allocate its buffers such that they can be mapped to the CPU. The final reduction will then be performed using PIO, minimizing latencies.

A further optimization is to completely eliminate the latency introduced by writing back the reduction result to the GPU. For this I implemented a special multi-GPU variant of the CG solver which allows the reduction to leave the GPU side copy of the reduction result in an inconsistent state. Instead of writing back the result, all further scalar calculations are performed on the host and the scalar values are passed to the kernels via kernel arguments. This does not cause any additional latency but the normal kernel launch latency.

The optimizations of the handling of scalar values improve the inverter performance by about 20 %. Utilizing them, the inverter provides 250 GFLOPS for a $24^3 \times 128$ lattice using the four GPUs on two AMD FirePro S10000.

Overall, the bandwidth in between the devices is essential for multi-device performance. Minimizing the size of the halo and overlapping the halo update with the $\not{D}$ calculation is essential for good performance. Yet, it cannot completely hide the cost

of transferring the data. This shows especially if QPI is involved. Therefore, the performance using four GPUs would probably improve if both AMD FirePro S10000 were attached to the same PCIe tree.

Chapter 5.

Results

In this chapter I present the results of the work performed. I focus on the ramifications this work has on execution time and energy efficiency for the different use cases. In addition, I will verify the cost effectiveness of the solution and provide a quick look at the physics results that were enabled by this work.

5.1. Comparison to Existing Solutions

To evaluate the compute time and energy consumption values of CL^{2}QCD, I perform comparisons to existing solutions for different use cases. As far as possible this comparison is based on published results. Where no data has been published I have performed reference measurements using an existing solution myself.

5.1.1. Compute Time

An essential criterion for the applicability of a given implementation is the time to solution. Therefore, I have benchmarked the performance of the $\not{D}$, the heatbath algorithm, and the HMC for a variety of lattice sizes. Since the LQCD group in Frankfurt is primarily interested in thermal systems, I have focused on such lattices.

The measurements have been performed on LOEWE-CSC, SANAM and a set of development systems located at the FIAS. I have covered a wide range of GPUs. Measurements have been performed on the AMD Radeon HD 5870, the AMD Radeon HD 6970 and the AMD Radeon HD 7970, covering the last three generations of AMD GPUs. Each of these is the fastest single-chip gaming GPU produced by AMD in its generation. The AMD Radeon HD 5870 is also the GPU available in LOEWE-CSC. In addition I also benchmarked the AMD FirePro V7800, which is build on the same chip as the AMD Radeon HD 5870, and the AMD FirePro S10000, which is used in SANAM. Thus, I also covered the last two generations of professional GPUs manufactured by AMD. For the $\not{D}$ benchmark I have also evaluated NVIDIA GPUs, namely the NVIDIA GeForce GTX 480, NVIDIA GeForce GTX 580 and NVIDIA GeForce GTX 680, thus again covering the last three generations of gaming GPUs.

All the benchmarks on AMD GPUs were performed using a beta version of release 12.102 of the AMD FirePro variant of the Catalyst driver. This version was pre-released to me as it contained register allocation fixes required for the $\not{D}$ kernel. The first publicly available release to contain these fixes is Catalyst 13.6. The measurements on

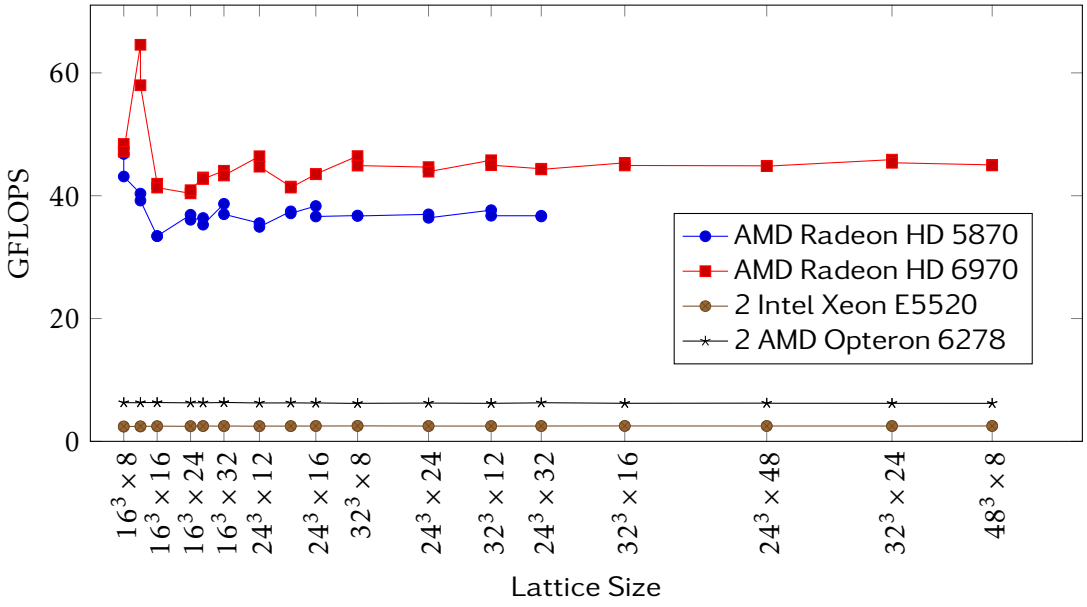

Figure 5.1.: Performance of the SP heatbath kernel.

NVIDIA GPUs used version 295.41 of the proprietary GPU driver by NVIDIA. Details on the systems used for the measurements can be found in Appendix C. Details on LOEWE-CSC and SANAM are presented in Appendix A and Appendix B, respectively.

Heatbath

Although it was never in the focus of the performance optimizations performed, I also report on the performance of the heatbath algorithm. While not showing the full potential of the hardware, the implementation of the heatbath algorithm gives an indication on the performance that can be reached with limited development effort based on the generic optimizations provided by the common parts of the implementation.

The performance of the heatbath algorithm is dominated by two kernels. That of the heatbath kernel is presented in Figure 5.1. This kernel makes large-scale use of the pseudo-random number generator (PRNG) and does not saturate the available bandwidth. Figure 5.2 shows the performance of the overrelaxation kernel. This kernel does not require the PRNG and shows much better performance.

Both kernels show four times the performance on GPUs compared to CPU. This is a common performance ratio for many applications. However, the CPU code still has room for performance improvements as it currently does not make use of vectorization. The performance difference in between the CPUs can be explained by the fact that the AMD system available for testing was both much newer and equipped with more cores and memory controllers than the Intel system.

The DP performance of the current implementation is only in the order of 10 GFLOPS. This is caused by register spilling, which is caused by the large working set of these

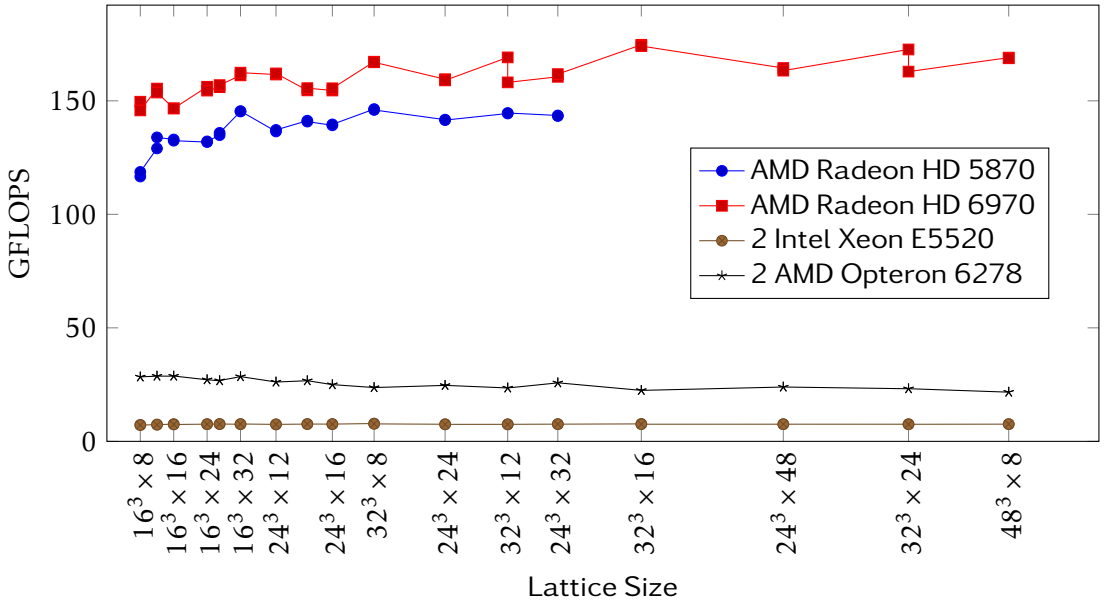

Figure 5.2.: Performance of the SP over-relaxation kernel.

kernels. It could probably be avoided with an optimized version of this kernel. In the heatbath algorithm there are no summations over the whole lattice, therefore, precision is less of an issue and the SP is currently sufficient.

A major issue for the heatbath algorithm is the size of the GPU memory, which limits the size of the studied lattices. This is especially an issue on the AMD Radeon HD 5870 and the AMD Radeon HD 6970. Their memory size is only 1 GB and 2 GB. In addition, usage of more than half of the memory by an OpenCL application is not supported officially. An unsupported way to utilize nearly all of the GPU memory is documented in the *AMD Developer Knowledge Base* [93], though. The OpenCL standard is not too specific as to what happens if a device runs out of physical memory. While current implementations all seem to report an error in this case, it would also be completely legal for an implementation to swap buffers or use host memory in this case. Both of these variants would obviously degrade performance.

Cardoso and Bicudo published numbers for an implementation based on NVIDIA CUDA [55]. They used the NVIDIA GeForce GTX 295 and the NVIDIA GeForce GTX 580. Compared to their numbers, the heatbath kernel only provides about half the performance on the AMD Radeon HD 5870 and the AMD Radeon HD 6970. The over-relaxation kernel shows similar performance even though the current implementation does not make use of the bandwidth reducing REC12 technique. Contrary to the implementation in CL^{2}QCD, the NVIDIA CUDA implementation cannot handle lattices that have a larger number of sites than the number of threads that can be scheduled on the GPU. CL^{2}QCD can process any lattice that fits into the memory of the GPU.

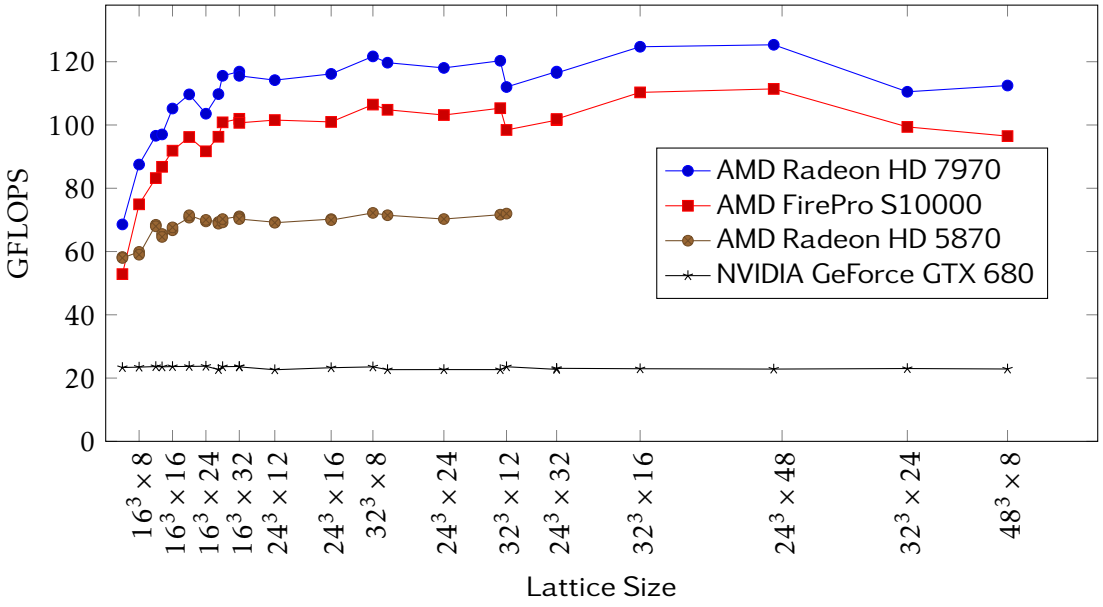

Figure 5.3.: Performance of the DP $\not{D}$ kernel.

Overall, all of the GPUs can be used efficiently for the heatbath algorithm. However, there is still some room for performance improvements on the existing hardware.

$\not{D}$ Operator

The inversion of the fermion matrix is the most compute intensive operation in the HMC and during operator evaluation. In terms of compute time $\not{D}$ is the most expensive operation in the solvers implementing this inversion [17]. Therefore, the performance of the $\not{D}$ implementation is of highest importance for the overall application performance. Figure 5.3 shows the performance of the $\not{D}$ kernel on three different GPUs for a variety of lattice sizes.

The elderly AMD Radeon HD 5870 performs at about 70 GFLOPS for a wide variety of lattice sizes. For smaller lattices, the performance drops slightly but is still above 50 GFLOPS. It's successor, the AMD Radeon HD 6970 scales with the increased memory bandwidth, providing about 80 GFLOPS.

With its much higher memory bandwidth, the AMD FirePro S10000 achieves about 100 GFLOPS for a wide variety of lattice sizes. While it cannot provide the full speedup for small lattice sizes, for large lattice sizes the performance scales very well with the increased memory bandwidth. For some lattice sizes it even peaks up to 110 GFLOPS. The AMD Radeon HD 7970, equipped with slightly more memory bandwidth than the AMD FirePro S10000, achieves about 120 GFLOPS and peaks up to 125 GFLOPS.

On the NVIDIA GeForce GTX 680 the $\not{D}$ kernel does not perform as well. The register spilling which occurs on that GPU severely limits performance to only 25 GFLOPS.

This register spilling is caused by a major limitation of the OpenCL implementation by NVIDIA. Different than in NVIDIA CUDA it is not possible to request a larger number of registers per thread from the compiler. Even though this would mean less concurrent threads on the hardware, it should result in a significant boost in performance. In addition, the OpenCL API does not provide the possibility to reconfigure the ratio of L1 cache to local memory, which in NVIDIA CUDA is possible on the NVIDIA GeForce GTX 680. This would allow to cache some of the spilled registers, reducing register bandwidth consumed by register spilling and enabling better performance. Thus, currently the code does not provide competitive performance on NVIDIA GPUs. However, those have not been in the focus of the development, and the provided performance shows the principle portability of the current implementation.

In addition, the gaming GPUs produced by NVIDIA are pretty crippled in terms of DP performance. Still, the $\not{D}$ kernel does not reach their theoretical peak DP performance. Therefore, it should still be bandwidth limited. However, this low DP peak performance might have an impact on performance and performance might be better on the professional series of GPUs by NVIDIA.

There is a multitude of $\not{D}$ performances given in literature. However, none of the published values has been measured on the latest generation of GPUs.

Clark *et al.* have shown a performance of 40 GFLOPS on the NVIDIA GeForce GTX 280 [17]. They make use of the REC12 technique to reduce the bandwidth requirements of the kernel. In comparison, the AMD Radeon HD 5870 is about 75 % faster than this. However, the NVIDIA GeForce GTX 280 is one generation older than the AMD Radeon HD 5870, which make the latter comparison somewhat unfair. Yet, it's peak bandwidth of 142 GB/s nearly matches the 154 GB/s of the AMD Radeon HD 5870.

The NVIDIA GeForce GTX 480 used by Alexandru *et al.* is of the same generation as the AMD Radeon HD 5870. Their NVIDIA CUDA based code achieves about 50 GFLOPS in DP [57]. Thus, the AMD Radeon HD 5870 is about 40 % faster.

Since the QUDA community has moved towards mixed-precision solvers, only SP performance numbers are available for current NVIDIA GPUs. Lacking other numbers I will quote those here. Interpretation, however, requires to keep in mind that a mixed precision solver usually converges slower than a DP solver. On the latest generation NVIDIA Tesla K20, QUDA achieves about 250 GFLOPS [59]. Using a special cache-friendly streaming strategy, QUDA even manages to exceed 300 GFLOPS on the previous-generation NVIDIA Tesla M2090 [51]. However, this approach only seems to work for rather large lattices.

For the Intel Xeon Phi only SP performance numbers have been published so far, too. Joó *et al.* achieved a performance of 295 GFLOPS to 320 GFLOPS for SP $\not{D}$ on the Intel Xeon Phi B1PRQ-7110P [59].

For problems that fit into the last-level cache, the Intel Xeon X5680 has been shown to reach 75 GFLOPS [41]. However, that performance is only reached in SP. For larger problems, their performance drops to 42 GFLOPS in SP while the AMD FirePro S10000 stays constant at more than 100 GFLOPS in DP. The CPU performance can be improved to 53 GFLOPS by merging multiple $\not{D}$ invocations. In that case the CPU manages to exceed the bandwidth limit. The implementation which is also used on the Intel Xeon

Phi can reach 120 GFLOPS on two Intel Xeon E5-2680 [59]. This is approximately the performance of one of the GPUs on the AMD FirePro S10000. However, the Intel Xeon E5-2680 uses SP while the AMD FirePro S10000 provides DP results.

On a single node of a Blue Gene/Q about 70 GFLOPS can be reached in DP $\not{D}$ [49]. This is at least 30 % less than what CL^{2}QCD achieves on the AMD FirePro S10000. However, on a Blue Gene/Q one would usually not use a single node only. Interestingly, SP $\not{D}$ on the Blue Gene/Q only achieves up to 90 GFLOPS. For a bandwidth bound problem like the $\not{D}$ kernel the performance would be expected to double going from DP to SP.

Overall, in DP the $\not{D}$ implementation presented in this thesis provides better performance than published for other implementations. However, CL^{2}QCD running on NVIDIA GPUs is currently unable to compete with QUDA in terms of performance. But, even on the elderly AMD Radeon HD 5870 it already outperforms the DP $\not{D}$ performance numbers published for other GPUs and those for CPUs. On the AMD FirePro S10000 it shows excellent scaling to the higher peak performance of that GPU and outperforms the older GPU by more than 40 %. This makes it the fastest DP $\not{D}$ on a single GPU.

Inverter

The inverter—or actually the solver—is the next building block on top of the $\not{D}$ and dominates the performance of both the HMC and the analysis stage. Comparing solver performances is tricky. Different solvers operate at different convergence rates. Thus, a higher-performance solver might actually have a higher time to solution. In addition, optimizations—like the used of mixed-precision—can also influence the convergence rate of a given solver. Thus, a proper comparison should be based on the time to solution for a given set of problems with a defined solution precision. However, lacking the option to evaluate other implementations in this way, I give only the raw performance values for the CG solver.

On the AMD Radeon HD 5870 about 50 GFLOPS are reached in DP. The AMD FirePro S10000 provides about 73 GFLOPS and the slightly faster AMD Radeon HD 7970 solves at 75 GFLOPS.

As mentioned above, for the Intel Xeon Phi only SP performance number have been published. The QUDA library, which is the performance reference on NVIDIA GPUs relies on mixed-precision solvers. The high-end Intel Xeon Phi B1PRQ-7110P achieves about 230 GFLOPS. The Intel Xeon Phi 5110P and NVIDIA Tesla K20 both run at about 205 GFLOPS.

All of the CG implementations—ours and those quoted—achieve 70 % to 80 % of the corresponding $\not{D}$ performance. The notable exception is the AMD Radeon HD 7970 which is hardly faster than the AMD FirePro S10000 although it is 10 % to 20 % faster in $\not{D}$. This might be related to the driver or the system in which the measurements where performed. Gpu-dev04, in which the AMD Radeon HD 7970 was benchmarked, has a significantly worse PCIe performance than SANAM, in which the AMD FirePro S10000 was benchmarked.

Table 5.1.: Parameter values of the three set-ups for the HMC performance benchmark. The value of m_π is only approximate.

set-up	A	B	C
$a\mu$	0.0025	0.0035	0.1
m_π	260	310	520

Hybrid Monte Carlo

To avoid depending on other systems for configuration generation, the performance of the HMC is of major importance.

In 'Lattice QCD based on OpenCL' [20] we presented performance tests of the HMC performance under realistic conditions. Here I will cite those numbers and extend them to include the performance of the AMD FirePro S10000, which is based on a completely different architecture than the previously used GPUs.

The tests were performed for one heavy and two lighter pion masses, as shown in Table 5.1. To simulate at maximal twist we chose $\beta = 3.9$ and $\kappa = \kappa_c = 0.160856$ according to Baron *et al.* [94]. Measurements were always performed over ten steps of the HMC algorithm. For set-up C we used $\tau = 1$, in set-ups A and B $\tau = 0.1$ was used. The integration was performed using a 2MN integrator with ten integration steps for the separate timescales of the fermion and the gauge part. All inversions were performed using a CG solver. For the CPU performance values tmlqcd was executed on one node of LOEWE-CSC. As they were found to be below the percentage limit, we neglected statistical errors.

Figure 5.4 and Figure 5.5 show the performance of set-up C for different lattice sizes. The AMD FirePro V7800 is at least 70 percent faster than the CPUs and about twice as fast for most lattice sizes. The AMD Radeon HD 5870 could not be used for this benchmark, as its memory is to small to hold all of the used lattice sizes. For the AMD FirePro S10000 memory size is not a problem. Its superior $\not{D}$ performance scales well to HMC performance. For small lattices it is twice as fast as the AMD FirePro V7800 and four times as fast as the CPU node. For larger lattices it scales even better, showing a speed-up of 2.6 over the older AMD FirePro V7800.

Figure 5.6 shows the performance on a $24^3 \times 8$ lattice for the different set-ups. The runtime for the different configurations scales approximately the same on all systems. For the lighter pion masses the AMD Radeon HD 5870 is twice as fast as the CPUs, too. For those masses the inverter is more dominant on the runtime of the HMC benefiting the GPUs with their high-performance $\not{D}$ implementation. The AMD FirePro S10000 is another factor of two faster than the AMD Radeon HD 5870. This means, a single GPU of the AMD FirePro S10000 provides a speed-up of four over a full node of LOEWE-CSC running tmlqcd. Given that a node of SANAM is equipped with four GPUs, it provides eight times the throughput of a node of LOEWE-CSC utilizing its GPU. Compared to a node of LOEWE-CSC utilizing its CPUs, the throughput even increases by a factor of 16.

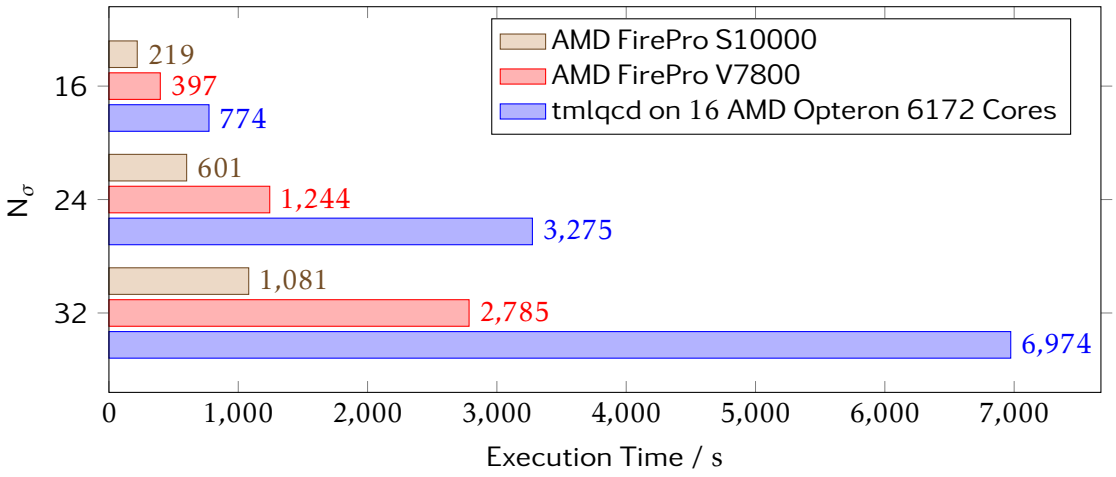

Figure 5.4.: HMC runtimes in seconds for set-up C for fixed $N_\tau = 8$

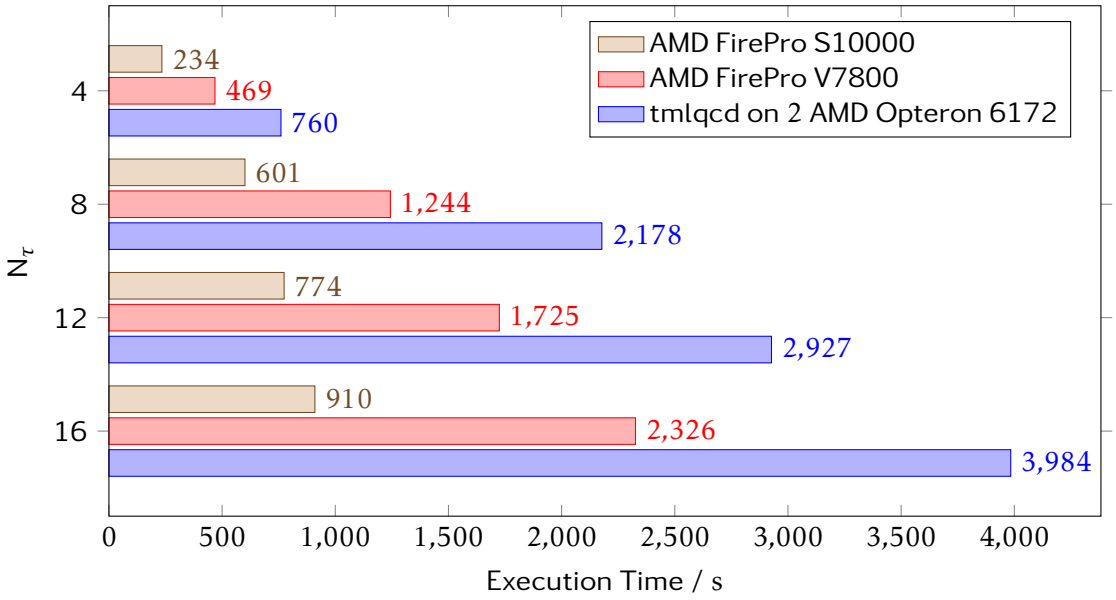

Figure 5.5.: HMC runtimes in seconds for set-up C for fixed $N_\sigma = 24$

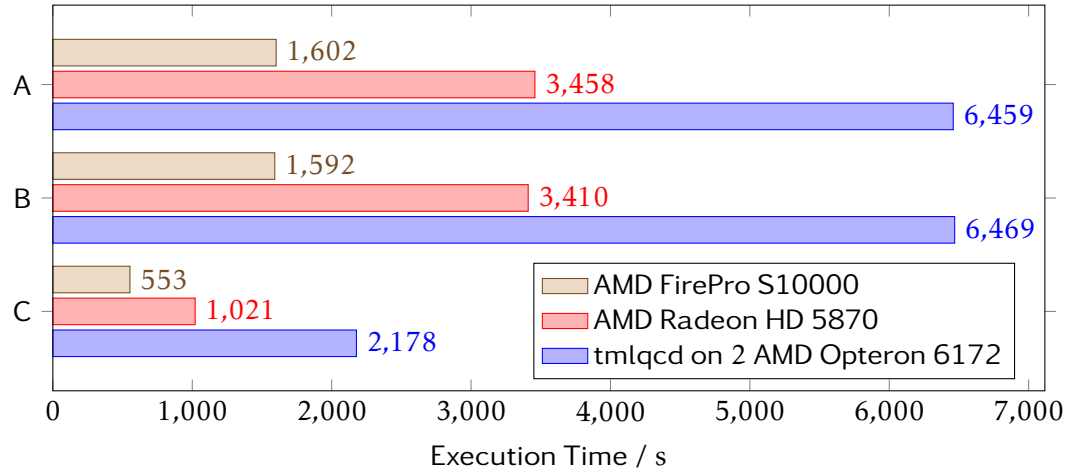

Figure 5.6.: HMC runtimes in seconds for set-ups A, B and C on a $24^3 \times 8$ lattice

Bonati *et al.* have benchmarked the HMC performance of the AMD Radeon HD 5870 and several NVIDIA GPUs using staggered fermions [15]. Due to the different discretization a direct comparison to our code is difficult. However, they found the AMD Radeon HD 5870 to be about 20 % faster than the NVIDIA Tesla S2050 running the OpenCL version of their code. Compared to the NVIDIA CUDA version, the AMD Radeon HD 5870 was 50 % slower, however. It is unclear what causes the big performance difference in between the OpenCL and the NVIDIA CUDA code running on the NVIDIA Tesla S2050. Bonati *et al.* also note that the small memory of the AMD Radeon HD 5870 limits the maximum problem size that can be studied.

While the AMD Radeon HD 5870 was severely limited by its small memory, the AMD FirePro S10000 can easily handle state-of-the-art lattice sizes like $32^3 \times 12$. Such lattices are currently used on SANAM to investigate the thermal QCD transition with two flavours of twisted mass fermions. The Z12 set-up used for this work follows up on the previously investigated set-ups A12, B12 and C12 [86]. For this set-up the HMC is configured to utilize mass preconditioning and a 2MN integrator on three timescales of eight, six and five steps. As Figure 5.7 shows, one GPU of the AMD FirePro S10000 provides a five times speed-up over a pure CPU system running tmlqcd. Therefore, a node with two AMD FirePro S10000, as it is the case in SANAM, provides 20 times the throughput of a pure CPU node.

The performance measurement of tmlqcd was performed on titanic. The system is equipped with two AMD Opteron 6220, providing a total of 16 Interlagos cores. While 4.3 times as fast as the tmlqcd run, the AMD Radeon HD 7970 is unable to match the speed-up of the AMD FirePro S10000. As the AMD Radeon HD 7970 does have the higher peak performance, this is probably an artefact of the system gpu-dev04 which hosted the AMD Radeon HD 7970.[1]

[1] As noted in Section 5.1.1, the AMD Radeon HD 7970 also already shows a strangely low inverter performance in that system.

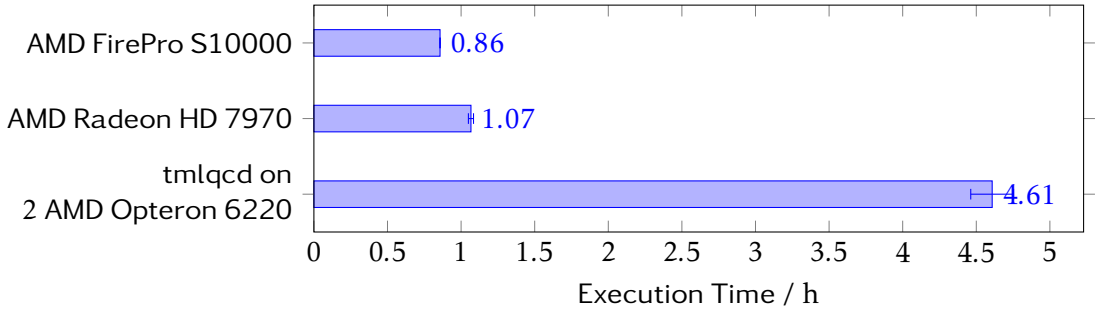

Figure 5.7.: Runtime for a single HMC step using the Z12 set-up with $\beta = 3.8175$ and $\kappa = 0.1634937$.

Table 5.2.: TCA per MFLOP for a variety of machines used for LQCD

Machine	Year	$/MFLOPS
QCDSP	1998	13.2
Blue Gene/L	2006	1.70
Kraken (x86)	2011	1.14
Sequoia (Blue Gene/Q)	2012	0.040
SANAM (x86 + GPU)	2012	0.033

5.1.2. Total Cost of Acquisition

Given the fact that budgets are usually limited, the TCA is an important metric for every system. There are hardly two systems that provide exactly the same performance. However, performance can—within limits—be scaled with the system size. Therefore, it is reasonable to compare the TCA normalized to the system performance in LQCD.

Table 5.2 gives an overview over the TCA of multiple machines that have been used for LQCD computations. In 1998 a Gordon Bell Prize was awarded for showing LQCD computations on QCDSP machines reaching a normalized TCA of $13.2/MFLOPS [45]. Since then the TCA per MFLOPS of performance has dropped by a factor of 400.

For the Blue Gene/L, Kraken, and Sequoia the normalized TCA has not been published. Therefore, I estimated the TCA of those systems based on publicly available data.

The Blue Gene/L delivers 12.2 TFLOPS on 32 Ki CPU cores[61]. Using a rack price of $1.3 million[95] I assume the TCA of the used system to be about $21 million.

Kraken is a Cray XT machine that is based on a grant providing $30 million for its construction [96]. It is based on AMD Opteron CPUs and uses Cray's proprietary interconnect technology. I estimated the normalized TCA by scaling it down to the 4096 cores that Babich *et al.* have shown to provide 952 GFLOPS [2]. With its normalized TCA of $1.14/MFLOPS this system seems to lag behind the development in LQCD performance shown by the less conventional systems.

Table 5.3.: TCA per MFLOP for hypothetical workstations. The performances for the systems not based on GPUs by AMD are taken from 'Lattice QCD on Intel Xeon Phi Coprocessors' [59].

Processor	$\not{D}$ \$/MFLOPS	CG \$/MFLOP	Precision
2 Intel Xeon E5-2680	0.058	0.073	single
4 Intel Xeon Phi 5110P	0.013	0.018	single
4 NVIDIA Tesla K20	0.020	0.024	single
4 AMD FirePro S10000	0.020	0.028	double

Finally, Sequoia—based on the current Blue Gene generation Blue Gene/Q—provides 6.18 PFLOPS [49] for an estimated TCA of $250 million [97]. This performance is matched by SANAM [98] with its AMD FirePro S10000 GPUs. However, Sequoia is still in the advantage of being capable of PFLOPS calculations which are out of SANAM's reach.

Given the limited availability of capacity systems, the cost of always-available workstation systems provide another interesting point for comparison. For this I created hypothetical workstation configurations based on a system by CADnetworks, which can be ordered with four NVIDIA Tesla K20 GPUs. Table 5.3 shows the normalized TCA for these systems.

Per flop, the pure CPU system is up to three times as expensive as an accelerator based system. The best normalized TCA is actually shown by the Intel Xeon Phi system. However, prices for the Intel Xeon Phi [99] are currently somewhat academic, as no store actually has it in stock. In the $\not{D}$ computation the AMD FirePro S10000 manages to match the NVIDIA Tesla K20, despite the NVIDIA Tesla K20's performance being provided for single precision computations. Given the bandwidth limited nature of LQCD this should approximately double the performance. In the inverter calculation the AMD FirePro S10000 falls behind the NVIDIA Tesla K20, which for the given performance values even utilizes half-precision computation. For a better comparison more recent DP precision values for the NVIDIA Tesla K20 and a mixed-precision code for the AMD FirePro S10000 would be required.

The usage of accelerators also allows the creation of truly low cost systems based on conventional consumer hardware. Table 5.4 provides the TCA for a set of hypothetical systems based on such hardware. The base system is the One Gamestar PC XL Core. It is shipped for a price of 1299 €[2] including an AMD Radeon HD 7970. The price of the latter has of course been deducted for the configurations using a different GPU.

For the Intel Xeon Phi 5110P[3] and the NVIDIA Tesla K20 the reduced system cost hardly effects normalized TCA. This is due to the high price of these accelerators which

[2] The price is taken from the One's web shop at `http://www.one.de` on 5 July 2013. It includes a VAT of 19 %.

[3] In fact the Intel Xeon Phi 5110P will not work in this system, due to its passive cooling solution.

Table 5.4.: TCA per MFLOP for hypothetical minimum acquisition cost systems based on a consumer system. The performances for the systems not based on GPUs by AMD are taken from 'Lattice QCD on Intel Xeon Phi Coprocessors' [59].

Processor	System Cost $	$\not{D}$ $/MFLOPS	CG $/MFLOP	Precision
Intel Xeon Phi 5110P	3672.01	0.013	0.018	single
NVIDIA Tesla K20	4807.27	0.019	0.023	single
AMD FirePro S10000	3987.46	0.017	0.027	double
AMD Radeon HD 7970	1400.30	0.012	0.019	double

in the four-accelerator configurations even dwarfed the workstation costs. The effect on normalized TCA is slightly larger for the AMD FirePro S10000, which still provides two GPUs in this minimal system.

AMD provides the same DP performance in its high-end consumer products as in its AMD FirePro line. Using the AMD Radeon HD 7970, that ships with the system, results in a normalized TCA of $0.012/MFLOPS to $0.019/MFLOPS, equalling that of the Intel Xeon Phi 5110P. However, using the AMD Radeon HD 7970 results in less than half of the system cost as when using the Intel Xeon Phi 5110P. In addition, the AMD Radeon HD 7970 provides DP calculations at the same normalized TCA as the Intel Xeon Phi 5110P provides single precision calculations.

The single node system is a factor 2 to 3 as price efficient as the big machines. Some of this can be attributed to the additional costs for high-performance interconnect and front-end nodes. However, due to their size these systems should compensate this by scaling effects in hardware prices. Therefore, the larger contribution is probably from the smaller system being tuned more heavily towards QCD computations. These systems do not contain the fastest CPUs and only the minimum amount of memory required for optimum memory speed. This reduces the system costs without negatively effecting LQCD performance.

In conclusion, CL^{2}QCD allows the computation of LQCD on systems using AMD GPUs at a TCA competitive with other accelerator based systems. Overall, accelerators match the Blue Gene systems in normalized TCA. Used in systems tuned for LQCD they can even outperform the Blue Gene systems by factor of 2 to 3.

5.1.3. Energy Consumption

Given today's energy costs and the cooling challenge, the energy required to solve a given problem is one of the key performance indicators. Therefore, I compared the energy consumption of running CL^{2}QCD with the energy consumption of running tmlqcd.

There are two metrics of interest. The average power consumption while running the application defines the required cooling. The total energy required to get to the result is important for the ecological impact of the system and the monetary costs. For

the purpose of this comparison I introduce an efficiency-metric which is defined as the number of steps the HMC algorithm can achieve per kWh.

The set-up used is that of the Z12 studies. The HMC algorithm is run on a given configuration of a $32^3 \times 12$ lattice using $\beta = 3.8025$ and $\beta = 3.8175$. I performed the comparison using two different systems: Tmlqcd runs on titanic, a system with two AMD Opteron 6220 and no GPUs. CL^{2}QCD runs on gpu-dev04, which is equipped with a varying number of AMD Radeon HD 7970 GPUs. In those measurements where more than one GPU was used I ran an own instance of CL^{2}QCD on each GPU. I removed unused GPUs from gpu-dev04 during the measurements. Such, the measurements represent a single-, dual-, or three-GPU system.

I measured the energy consumption of the whole systems at the wall outlet using an LMG95 [100]. All measurements include the whole application runtime. For CL^{2}QCD this includes the GPU code initialization from a pre-filled cache during application start-up.

The runtime of the HMC algorithm is influenced by the random numbers used. Therefore, all measurements were performed for ten random number seeds and the analysis was performed on the average result.

Figure 5.8 shows the power consumption of the systems averaged over the duration of one step of the HMC algorithm. Equipped with one GPU, the system gpu-dev04 requires 348 W running CL^{2}QCD. This is hardly more than the 343 W required by the pure CPU system titanic running tmlqcd. Adding the second GPU to run a second instance of the HMC increases the power consumption by about 235 W. The third GPU leads to a larger increase in power consumption of about 271 W to a total of 884 W. The exact value of β does not have any significant effect on the power consumption.

The power-consumption increase depending on the number of used GPUs allows to model the power consumption of the GPU computation and the offset of the base system. The following formula gives a simple model for this:

$$P_n = n \cdot P_{\text{GPU}} + P_{\text{offset}}. \tag{5.1}$$

In this model P_n is the power consumed by a system of n GPUs running n application instances. P_{GPU} is the power required by a GPU computation and P_{offset} is the system offset.

However, the power consumption values in Figure 5.8 cannot be fit using this simple model. As I will show in Subsection 5.2.1 the system gpu-dev04 shows exceptionally bad scaling behaviour, especially for the third GPU. Therefore, I will for now use a worst and a best case estimation derived from the three- and the two-GPU case, respectively. The worst case estimated from the three GPU case leads to:

$$P_{\text{GPU}} = 271\,\text{W}, \tag{5.2}$$

$$P_{\text{offset}} = 73.5\,\text{W}. \tag{5.3}$$

The best case leads to:

$$P_{\text{GPU}} = 235\,\text{W}, \tag{5.4}$$

$$P_{\text{offset}} = 113\,\text{W}. \tag{5.5}$$

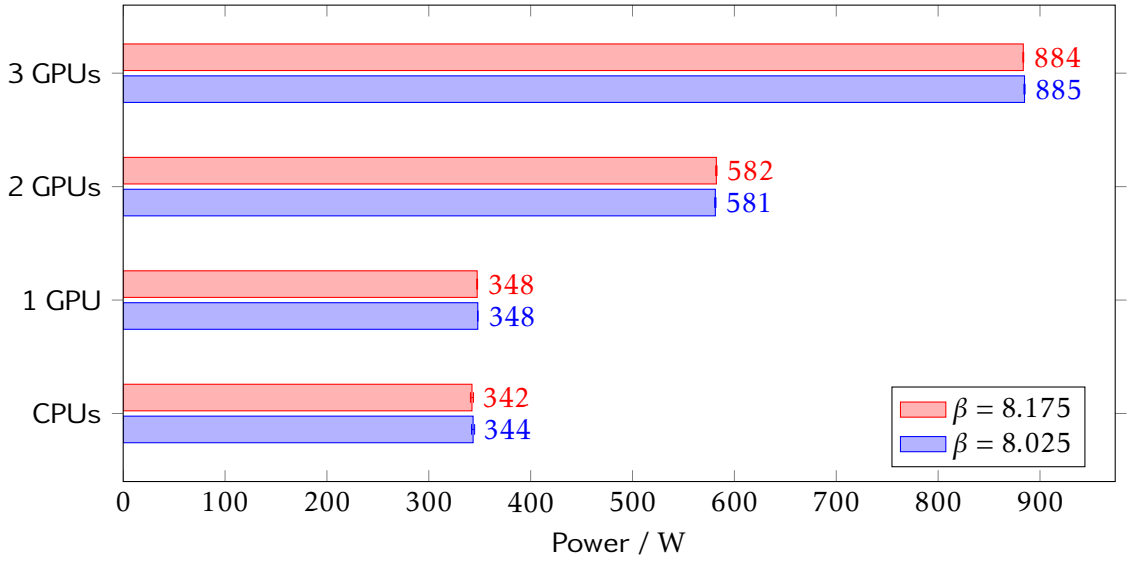

Figure 5.8.: Average power consumption of one HMC step

The specified typical power consumption of the AMD Radeon HD 7970 is 210 W. The specified maximum power consumption, which can be derived from its power connectors, is 225 W. This fits well with the observed values, which include the CPU and main memory power consumption caused by the host part of the application and the limited efficiency of the power supply.

Figure 5.9 shows the maximum power consumption during the calculation of a single HMC step. In this figure the maximum means the highest average power consumption in a 0.5 s sampling interval, not the highest infinitesimal peak consumption. While there is hardly any difference between the CPU and the single-GPU system when averaging the power consumption over the whole calculation, the maximum consumption of the single-GPU system shows a 23 W higher maximum consumption than the CPU system. The CPU system shows a maximum consumption of 364 W. This is 21 W more than the average. The single GPU consumes up to 387 W, surpassing its average by about 39 W. The CPU system has a more constant energy consumption than the GPU system.

The three-GPU system breaks the kW barrier, consuming up to 1009 W. Comparison of the single-, dual- and three-GPU systems shows an increase in maximum power consumption of 263 W to 358 W per GPU. This is 17 % to 60 % beyond the maximum energy consumption specified for a single AMD Radeon HD 7970. Even taking into account the power consumption due to the additional CPU load generated, the three-GPU system shows a disproportional increase in power consumption. In Subsection 5.2.1 I show that utilizing three GPUs hits the thermal limits of this system. This is probably the reason for this disproportional increase in power consumption and could be avoided by a system providing proper cooling as the SANAM nodes do.

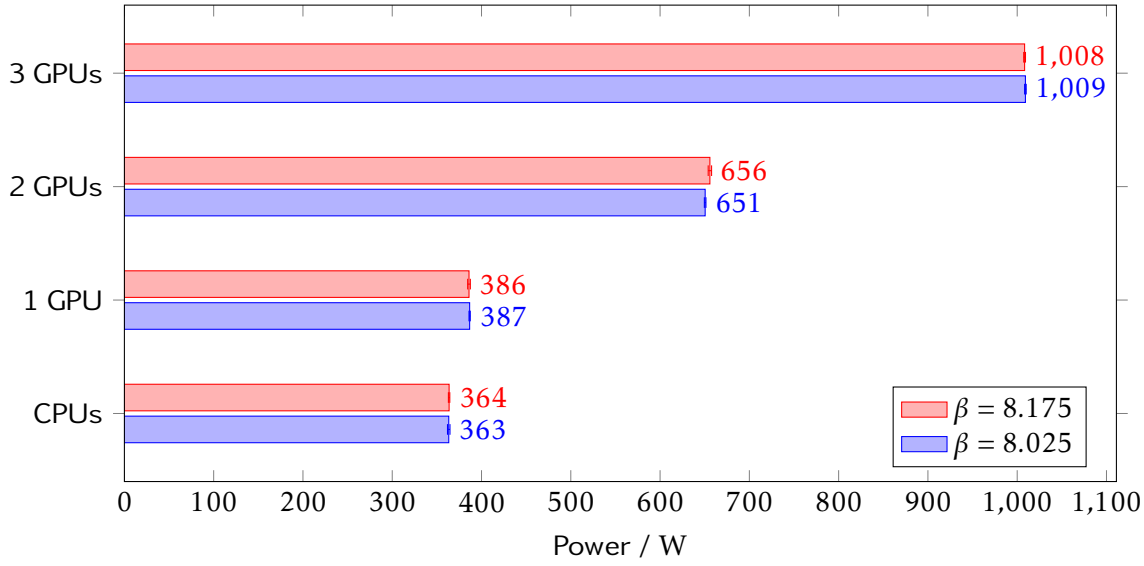

Figure 5.9.: Maximum power consumption during one HMC step

Figure 5.10 shows the energy efficiency of the different systems when performing HMC calculations. The efficiency is calculated as follows:

$$\text{eff} = n/E. \tag{5.6}$$

E is the energy consumed during the measurement. n is the number of HMC steps performed. For the single GPU and the CPU system n equals one. For the multi-GPU systems n is equal to the number of GPUs, as an own instance of CL^{2}QCD was executed on each GPU.

The CPU system performs about 0.63 HMC steps per kWh. The single-GPU system is 4.25 to 4.5 times as efficient. It can perform 2.7 to 2.85 HMC steps per kWh.

Contrary to the CPU system, the energy efficiency of the GPU systems shows a dependency on the value of β. The power consumption of the systems is the same for both values of β used. This difference in energy consumption stems only from the difference in execution time.

The same effect exists in between the CPU and the single-GPU system. Both have basically the same power consumption. Thus, the higher energy efficiency of the GPU system stems from its improved throughput.

The two-GPU system shows the best energy efficiency, performing 3.3 HMC steps per kWh for $\beta = 8.025$. This is a 5.25 times the energy efficiency of the CPU system.

The three-GPU system manages to calculate about 3 HMC steps per kWh. While this is more than the single-GPU system provides, it is less than the two-GPU system achieves. This comparatively low efficiency is a result of the disproportional increase in power consumption and an disproportional increase in compute time time observed for the three-GPU system.

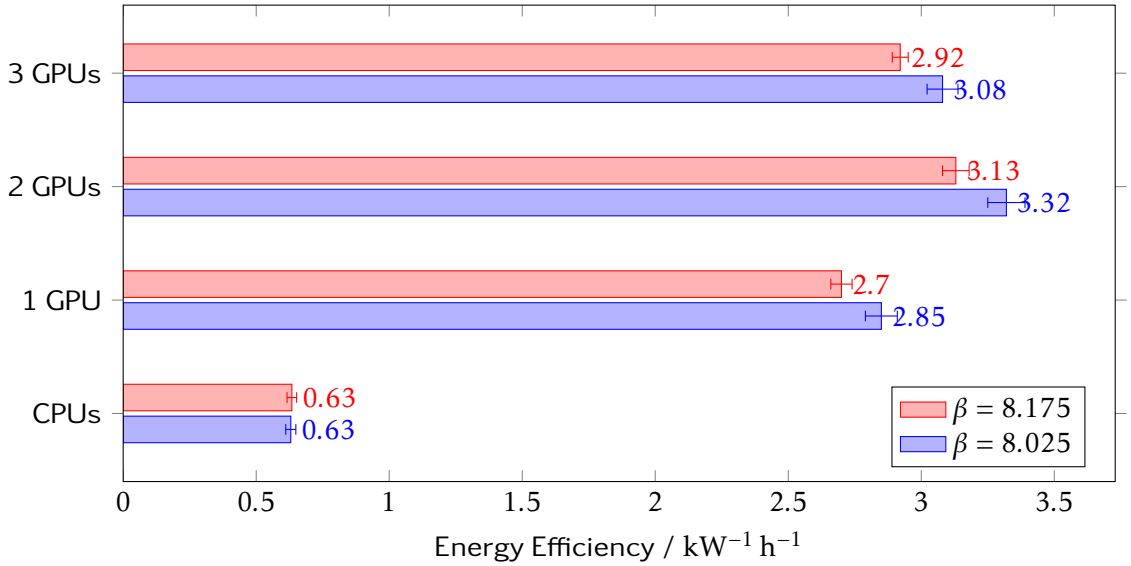

Figure 5.10.: Energy efficiency measured in HMC steps per energy consumed.

The system gpu-dev04 used for the measurements is not optimal from a power consumption perspective. As a development workstation it is equipped with 24 CPU cores. While those help in the development process by speeding up the compilation, only one is used by each instance of CL^{2}QCD. Therefore, the system overhead of the power consumption could be reduced by using a more lightweight system to host the GPUs. In addition, as I show in Subsection 5.2.1, gpu-dev04 is ill-suited to host multiple GPUs. The slowdown observed when running multiple instances of CL^{2}QCD is much smaller on SANAM. This should lead to a better energy-efficiency scaling on that system.

The following formula gives a simple model for the energy consumption E_n of a system with n GPUs. It assumes a linear slowdown s of each single-GPU application caused by additional GPUs in the system. In addition, the power consumption of each GPU is assumed to be independent of additional GPUs in the system. Thus, there is a small linear increase in runtime and a linear increase in power consumption with each GPU.

$$E_n = t_n P_n = (1 + (n-1)s)\, t_1 P_n \tag{5.7}$$

Running a separate application instance on each GPU there are n results produced using the Energy E_n. Thus, the efficiency eff_n of a system with n GPUs can be calculated as follows:

$$\mathrm{eff}_n = n/E_n. \tag{5.8}$$

Normalizing the efficiency to that of the single-GPU system makes it independent of

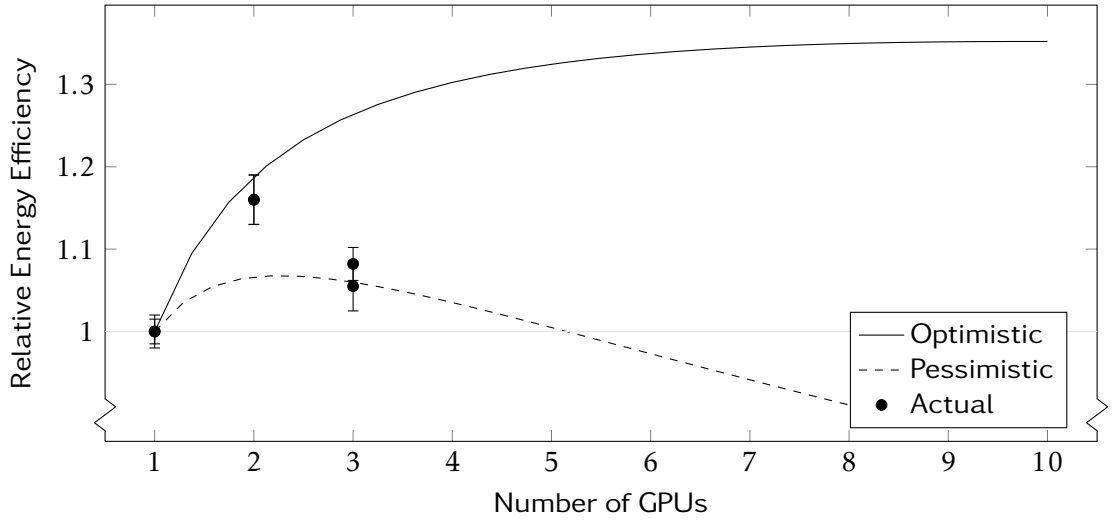

Figure 5.11.: Modelled energy efficiency of multi-GPU systems and actual efficiency on gpu-dev04 normalized to that of an identical single-GPU system.

the actual execution time:

$$\delta\mathrm{eff}_n = \frac{\mathrm{eff}_n}{\mathrm{eff}_1} = \frac{n \cdot E_1}{E_n} = \frac{n}{1 + (n-1)\,s}\frac{P_1}{P_n}, \tag{5.9}$$

$$\delta\mathrm{eff}_n = \frac{n}{1 + (n-1)\,s}\frac{P_{\mathrm{GPU}} + P_{\mathrm{Offset}}}{n P_{\mathrm{GPU}} + P_{\mathrm{offset}}}. \tag{5.10}$$

Figure 5.11 shows the result of this model for two sets of parameters. The optimistic curve uses the power values extracted from the measurements using two GPUs. The value of the slowdown parameter is only 0.5 % per additional GPU. This is similar to the value observed on SANAM. The pessimistic curve is modelled after the values observed for three GPUs in gpu-dev04. It uses a slowdown of 5 %.

The comparison to the actual values observed on gpu-dev04 shows that the simple model is unable to explain the behaviour of that system. However, assuming similar throughput scaling as on SANAM the system could scale to large numbers of GPUs. At about eight GPUs the curve flattens at an energy efficiency that is about one third better than that of the single-GPU system. However, given the bad throughput scaling observed for the three-GPU system, the energy falls below that of the single-GPU system if more than four GPUs are used. All of this happens at an already high level of energy efficiency, which, for the single-GPU system, is four times that of the CPU system.

Overall, the GPU systems show a much better energy efficiency than the CPU system. In the same energy budget they provide 4.25 to 5.25 times the throughput of the CPU system. The average power consumption of the GPUs fits well with the specifications given by the manufacturer. Yet, the observed maximum power consumption of the

GPU calculation exceeds the specified maximum power consumption of the GPU by up to 43 %. This shows that the power consumption of the host systems—and GPUs potentially exceeding their specified power consumption for short intervals—must not be neglected when estimating peak power consumption of a GPU. The bad scaling to three GPUs on gpu-dev04 also shows that the host system must be carefully chosen. Otherwise, the slowdown caused by utilizing multiple GPUs can overcompensate the benefit of dividing the host system energy consumption in between the GPUs. Still, even in this case energy-efficiency of the GPU system exceeds that of the CPU system by far.

5.2. Scaling to Multiple GPUs

There are three ways to utilize multiple GPUs. In the analysis stage and for parameter sweeps only throughput is of interest. But, if a large number of HMC steps are required for a single set of parameters, the wall time of a single HMC step becomes relevant. Thirdly, multiple devices become interesting if a single device is unable to process a lattice because of memory size constrains.

5.2.1. Throughput

In a system with n GPUs, running a separate instance of the HMC algorithm on each GPU should provide n times the throughput of a single GPU. However, host memory and PCIe are shared resources, which might cause some interference. On today's CPUs the host part of each instance can use an own CPU core. But, the GPU driver might internally require some serialization, imposing another possibility for interference in between the instances. All these effects might slow down each instance, reducing the overall throughput.

To measure the slowdown I ran n instances on n GPUs, varying n from one to the maximum number of GPUs in the system. Each instance performed a single step of the HMC algorithm using the Z12 set-up.

Figure 5.12 shows the time required to complete all n instances on gpu-dev04. Running two instances on two GPUs takes about 3 % longer than running a single instance on a single GPU. Running three instances on three GPUs takes about 10 % longer than running a single instance on a single GPU. This is a significant slowdown. Therefore, three GPUs provide only about 2.7 time the throughput of a single GPU in this system.

The situation is different on SANAM. The runtimes shown in Figure 5.13 show that on SANAM even running four instances on four GPUs only takes about 2.5 % longer than a single instance on a single GPU.

On SANAM there is also no significant slowdown when moving from two to three GPUs. The third and the fourth GPU are connected to the second CPU socket of that system. This probably explains why there is hardly any effect in between the third GPU and the first two.

On gpu-dev04 the second and the third GPU are attached to the same CPU socket. This might explain why the slowdown is much larger for the third GPU. However,

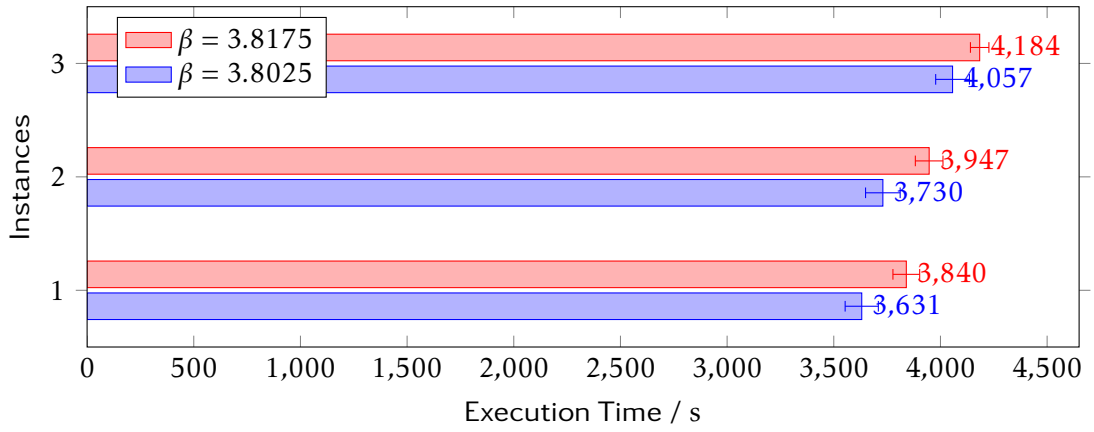

Figure 5.12.: Time for one step of the HMC algorithm on gpu-dev04 if multiple instances of CL^{2}QCD are running concurrently.

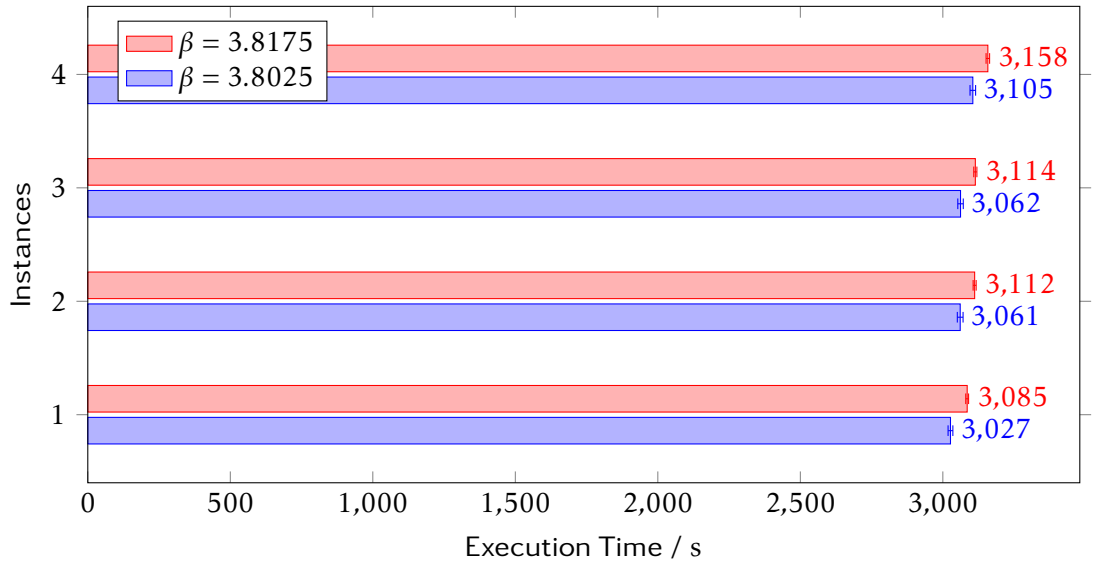

Figure 5.13.: Time for one step of the HMC algorithm on SANAM if multiple instances of CL^{2}QCD are running concurrently on the same node.

contrary to the situation on SANAM there is already a significant interference between the first two GPUs, despite them being attached to different CPUs.

The more likely explanation for the huge slowdown on gpu-dev04 is overheating. If the system only contains a single GPU, this GPU will never exceed 81 °C. Adding a second GPU will make both GPUs reach a reported temperature of 90 °C. On average they will be running at about 88 °C. Running three GPUs in gpu-dev04, the outermost GPUs reaches temperatures of about 92 °C. The middle GPU even reaches reported temperatures of up to 97 °C. On average it runs at about 95 °C, the temperature at which thermal throttling occurs for this GPU. Given that the reported temperatures are measured on the board of the GPU—not inside the actual GPU chip—even the outermost GPU might be affected by thermal throttling. On SANAM, where no such slowdown is observed, GPU temperatures stay in the 70 °C range even if all four GPUs are under load.

Obviously the choice of the host system has a huge impact on this type of scaling. On SANAM application throughput has been shown to scale up to four GPUs with minimal losses of only 2.5 %. Given a host system with good PCIe performance and sufficient cooling it should be possible to scale to even larger numbers of GPUs.

5.2.2. Latency

I have studied the scaling behaviour of CL^{2}QCD on SANAM. As the maximum number of GPUs in a node of SANAM is four, this limited the scaling analysis.

Figure 5.14 shows that for lattices with a sufficiently large time dimension a good scaling behaviour can be observed in $\not{D}$. For lattices that also fit onto a single GPUs the efficiency is 75 % to 95 % Here, I define efficiency as follows:

$$\text{Efficiency} = \frac{\text{Performance using } N \text{ GPUs}}{N \times \text{Performance of a single GPU}}. \tag{5.11}$$

On sufficiently large lattices like $32^3 \times 256$ even 419 GFLOPS are reached on four GPUs. Sadly, due to memory constrains this lattice cannot be used in the inverter. The $32^3 \times 12$ lattice shows about 50 % speed-up on two GPUs. To be sped up further, this kind of lattice will require parallelization in additional directions.

Figure 5.14 also shows the effect of DirectGMA, QPI, and GPU selection. The lower performance of the $32^3 \times 12$ lattice is achieved if two GPUs on separate AMD FirePro S10000 are used. In this case, QPI is involved and DirectGMA cannot be used. The effect also shows, though less, for the other lattices. For them the performance does not double when moving from two to four GPUs.

Figure 5.15 shows the strong scaling for the inverter. Due to memory constraints single-GPU data is only available for the $32^3 \times 12$ lattice. For the vacuum lattices, two GPUs can reach 140 GFLOPS, nearly twice the single-GPUs peak performance in the CG. Four GPUs reach 250 GFLOPS, an efficiency of 85 % versus the single GPU peak.

In the strong scaling case the local lattice size reduces with each added GPU. This can result in reduced performance of the $\not{D}$ kernel execution. Therefore, it is also interesting to observe the scaling behaviour given a fixed local lattice size. Figure 5.16

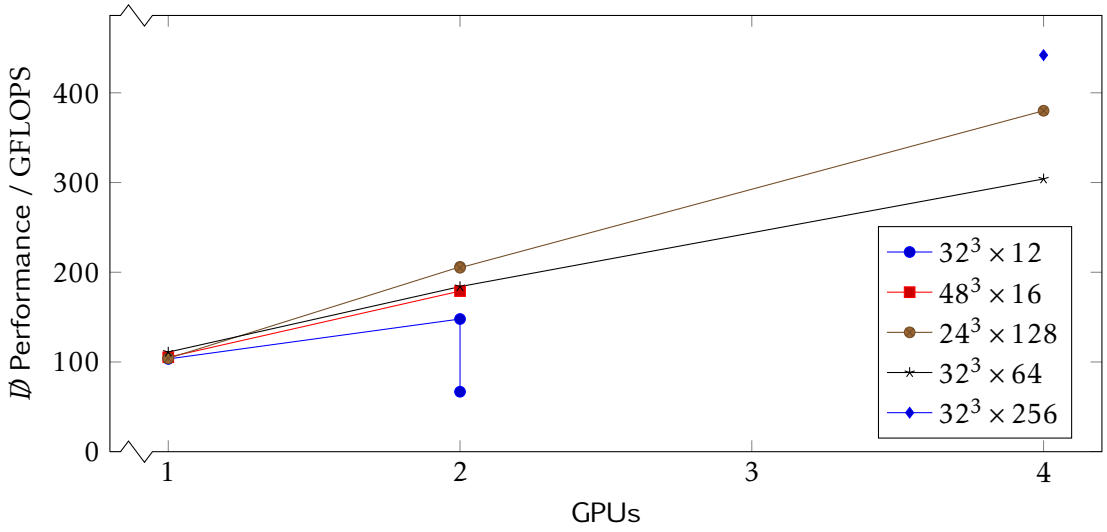

Figure 5.14.: Strong scaling of the $\not{D}$ operator on the AMD FirePro S10000 GPUs in SANAM for multiple lattice sizes. The jump at two GPUs is caused by communicating either between GPUs attached to the same CPU or GPUs attached to different CPUs involving QPI in the communication.

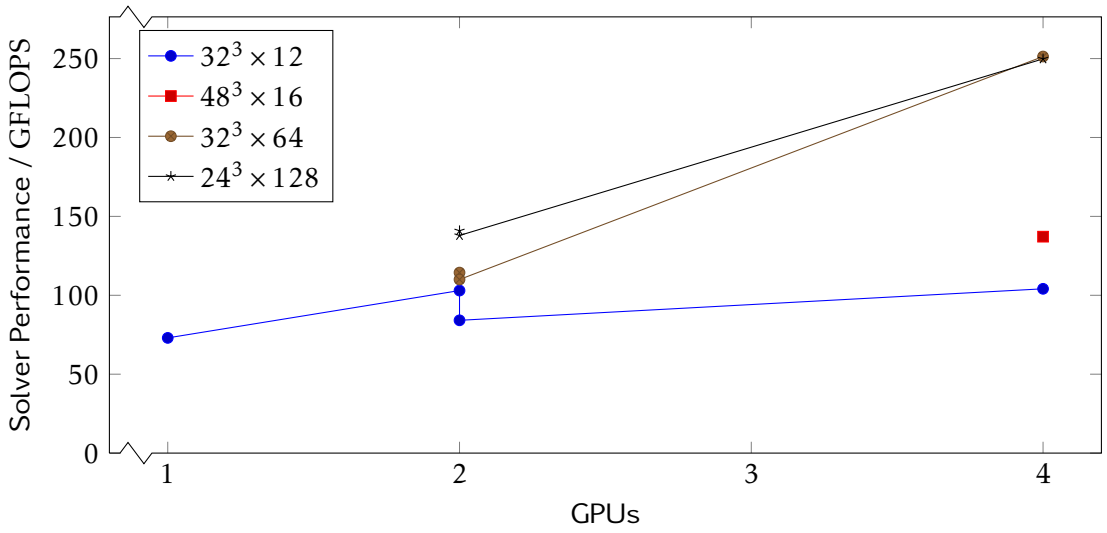

Figure 5.15.: Strong scaling of the CG solver on the AMD FirePro S10000 GPUs in SANAM for multiple lattice sizes. The jump at two GPUs is caused by communicating either between GPUs attached to the same CPU or GPUs attached to different CPUs involving QPI in the communication.

shows the $\not{D}$ performance in this case. Figure 5.17 does the same for the inverter. The results match that of the strong scaling case. Obviously the reduction in local lattice size is not an issue for the lattices analysed.

Weak scaling with a local lattice size of $32^3 \times 16$ very clearly shows the effect of using DirectGMA. Switching to the slower conventional transfer method for three GPUs leaves hardly any speed-up compared to two GPUs, where DirectGMA is used. The effect is smaller on the $24^3 \times 32$ lattice, with its much smaller halo. The inverter scales linearly for both local lattice sizes. In both cases, the efficiency is about 85 %.

Babich *et al.* have published their results of utilizing multiple NVIDIA GPUs using NVIDIA CUDA in 2010 [16]. Their implementation utilizes MPI instead of operating all GPUs from a single host process. They show weak scaling with a local lattice size of $24^3 \times 32$ to be linear up to 32 GPUs. In DP they reach about 32 GFLOPS per GPU in BiCGSTAB, while we reach about 63 GFLOPS in CG. However, their data is based on the much older NVIDIA GeForce GTX 285 and can utilize GPUs in multiple host systems. Normalizing the performance by the peak bandwidth of the used GPUs, CL^{2}QCD has a performance advantage of about 25 %. Most of this can probably be attributed to the more direct communication.

Using mixed precision in combination with a GCR-DD solver Babich *et al.* have shown to scale up to 256 GPUs [2]. However, their code is optimized to study much larger lattices, while the development of CL^{2}QCD was so far focused at lattices like $32^3 \times 12$ and similar sizes used for parameter sweeps. Still, this indicates two potential directions for further performance improvements in CL^{2}QCD.

The Intel Xeon Phi has been shown to provide about 122 GFLOPS per device for up to 32 devices [59]. However, no DP results are available for that platform. Given the high degree of optimization applied to the data transfer in CL^{2}QCD, the Intel Xeon Phi might currently provide better device-to-device communication capabilities than the AMD FirePro S10000. However, the 122 GFLOPS are only about half of the single-device performance. On the AMD FirePro S10000 CL^{2}QCD achieves about 85 % of the single-device performance when scaling to multiple devices. Unlike the Intel Xeon Phi implementation, CL^{2}QCD is not restricted to SP and operates at DP accuracy.

5.2.3. Problem Size

The multi-GPU capabilities increases the maximum problem size that can be studied using CL^{2}QCD. In Subsection 5.2.2 I showed the performance of the $24^3 \times 128$ and the $32^3 \times 64$ lattice. Both do not fit into a single GPU of an AMD FirePro S10000 but can be computed by using both GPUs. In addition, for these lattices CL^{2}QCD also shows nice performance scaling to additional GPUs.

The next interesting problem size to follow up on the Z12 studies is $48^3 \times 16$. It is also too large to fit a single GPU. Using all four GPUs of the two AMD FirePro S10000 in a SANAM node this lattice can be simulated. In that case the inverter performs at about 93 GFLOPS. This is equivalent to the performance of eight nodes in LOEWE-CSC running tmlqcd.

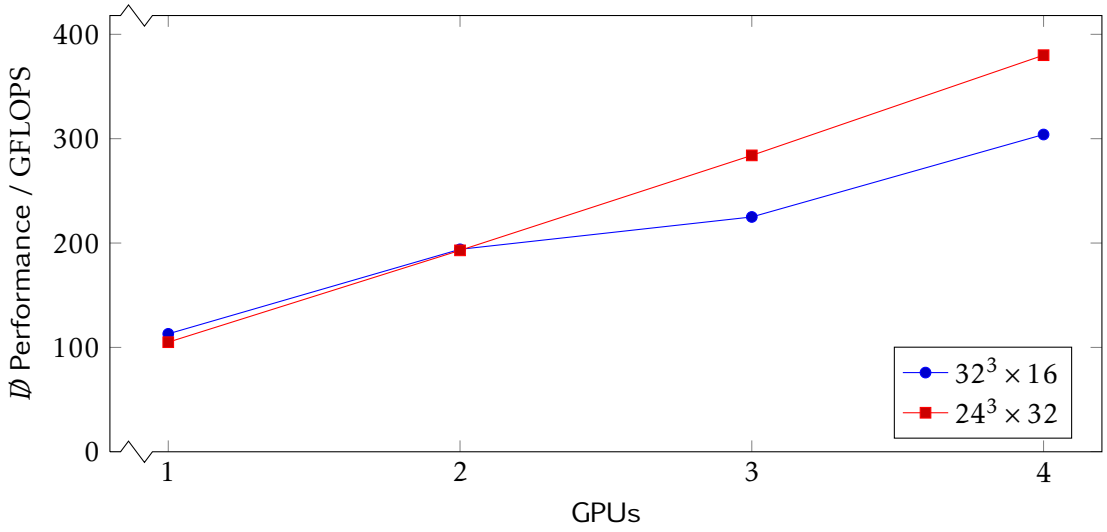

Figure 5.16.: Weak scaling of the $\not{D}$ operator on the AMD FirePro S10000 GPUs in SANAM for two different local lattice sizes.

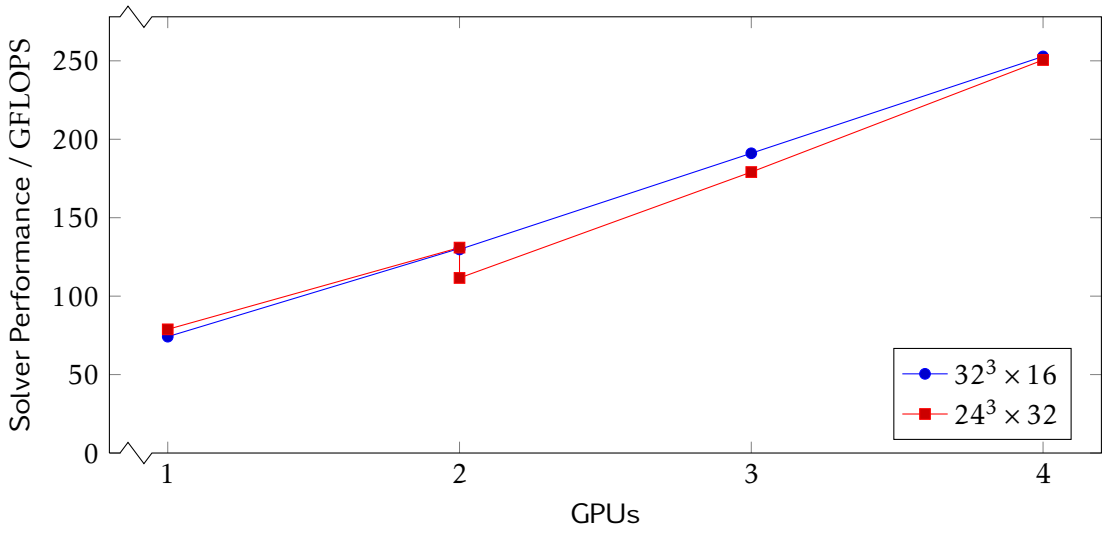

Figure 5.17.: Weak scaling of the CG solver on the AMD FirePro S10000 GPUs in SANAM for two different local lattice sizes. The jump at two GPUs is caused by communicating either between GPUs attached to the same CPU or GPUs attached to different CPUs involving QPI in the communication.

5.2.4. Conclusion

CL^{2}QCD can utilize multiple GPUs in all three variants—throughput improvement, latency reduction, and enabling larger lattices.

For thermal lattices, where parameter sweeps are commonly required, CL^{2}QCD can utilize multiple GPUs in a single node using separate application instances, achieving throughput scaling without any significant losses. However, care must be taken to choose a proper host systems, as insufficient cooling and PCIe issues have been observed to cause interference.

Vacuum lattices, which are typically somewhat larger than thermal lattices, can be sped up using multiple GPUs for a single application instance. Using extensive optimization of the device-to-device and the device-to-host communication, CL^{2}QCD manages to achieve linear scaling. In $\not{D}$ perfect scaling can be achieved for lattices of sufficient size. The CG solver achieves linear scaling with an efficiency of about 85 %.

Finally, utilizing multiple GPUs also enables the processing of lattices which do not fit into the memory of a single GPU. Examples for for this are $32^3 \times 128$ and $48^3 \times 16$ lattices. The latter would be the successor of the $32^3 \times 12$ lattices used in the Z12 studies.

5.3. Results obtained via CL^{2}QCD

The code has so far been used extensively to study the Z12 set-up, which has been mentioned multiple times in this chapter. However, no publications have been made on the results of this study, yet.

CL^{2}QCD has also been used for 'The nature of the Roberge-Weiss transition in N_f=2 QCD with Wilson fermions' [91]. There, the QCD phase diagram is studied using Wilson fermions in regions so far only studied using staggered fermions. For this, both LOEWE-CSC and SANAM were utilized. Following the success of this study, further investigations will be performed using smaller quark masses.

Chapter 6.

Conclusion

This thesis presents architecture and optimization of CL^{2}QCD—the first implementation of LQCD using OpenCL for Wilson fermions—as well as a performance, TCA, and energy-efficiency analysis. The optimizations are based on an in-depth analysis of optimization techniques for bandwidth-limited applications on GPUs. Based on OpenCL, CL^{2}QCD is not limited to accelerators by a single vendor and shows excellent performance on AMD GPUs. In addition, it also operates on CPUs.

For AMD GPUs the implementation manages to utilize more than 70 % of the theoretical peak device bandwidth in the $\not{D}$ kernel. Reaching 70 GFLOPS in DP on the AMD Radeon HD 5870 it outperforms published performances of both NVIDIA and CPU based codes. This performance scales with device bandwidth, reaching 100 GFLOPS on the newer AMD FirePro S10000 used in SANAM and 120 GFLOPS on the AMD Radeon HD 7970—a performance unmatched by other DP $\not{D}$ kernels.

Running the full HMC there is a speed-up of two when comparing the AMD Radeon HD 5870 with a CPU system containing two AMD Opteron 6172 CPUs. This speed-up increases to a factor of four when using the AMD FirePro S10000 or the AMD Radeon HD 7970.

The speed-up translates directly into energy-efficiency, as a single-GPU system operates on the same power budget as a pure CPU system. Utilizing multi-GPU systems further improves energy efficiency. Overall, GPU systems provide 4.25 to 5.25 times the energy-efficiency of a CPU system with two AMD Opteron 6220.

On such systems CL^{2}QCD shows excellent scaling properties. If multiple instances of the HMCs algorithm are executed concurrently—as convenient to perform parameter range scans for thermal lattices—throughput scales without any significant performance losses. Using multiple GPUs to reduce the wall time of processing a single vacuum lattice shows perfect scaling in $\not{D}$ and linear scaling with an efficiency of 85 % in CG.

In terms of TCA LQCD calculations on accelerators match conventionally-used capacity machines. Dedicated accelerator-based systems can even improve cost efficiency by a factor of 2 to 3. Here, CL^{2}QCD even enables DP LQCD computations at the lowest entry price by far.

To achieve these results, hardware characteristics have been analysed thoroughly. To address a performance issue of the AMD Radeon HD 5870 and similar GPUs, an optimized SoA pattern was developed. For QCD data types this provides a speed up of 50 % over using AoS or conventional SoA patterns.

High memory-bandwidth utilization can be achieved with a small number of threads.

Thus, a large number of registers per thread can be used by bandwidth-limited codes. Still, register usage has proven to be a major challenge to solve in order to achieve high performance on the given hardware.

Other challenges are latency and bandwidth of PCIe, which connects the GPUs and the CPU. Different methods of data transfer provide different characteristics. Thus, a variety of these methods is used. For compatibility reasons, an automatic selection depending on the hardware is performed. For inter-GPU communication, this selection is additionally based on start-up benchmarks, as the performance highly depends on the system layout and the GPUs involved.

On the AMD Radeon HD 5870 the combination of all the techniques presented in this thesis lead from a $\not{D}$ kernel utilizing 20 GB/s to one utilizing 120 GB/s—six times the previous performance.

OpenCL is a valuable tool for portable GPGPU computations. As this thesis shows, complex applications can be implemented using OpenCL and, on top of that, achieve excellent performance. Its approach of run-time code compilation to target architectures helped in the creation of versatile tools [68, 76] to study optimization techniques. However, implementation quirks by the vendors have at some points hindered optimization. In addition, at the feature level, NVIDIA does not provide the same level of support to OpenCL as it does to NVIDIA CUDA.

As presented in this thesis, CL^{2}QCD provides excellent performance on the AMD platform and can be used on all accelerator platforms and CPUs that support OpenCL. Its architecture has been modularized. Such, new algorithms can be implemented quickly, reusing components and profiting from all the given optimizations. This also allowed for collaborative development, where the co-authors of the other publications on CL^{2}QCD [19, 20] focused on the physics and I focused on the optimizations. Yet, as always, there is of course still room for further improvement.

Further application speed-up might be reached by further optimization of the inverter performance. One way to do this is using mixed precision solvers [17]. Those had originally been avoided because of the limited memory capacity of the AMD Radeon HD 5870. Initial prototypes on the AMD Radeon HD 7970 have been stashed because of compiler issues with the SP $\not{D}$ kernel. However, this GPU has already shown excellent performance in SP codes [101] and similar problems have already been solved for the DP kernel. Thus, future drivers should resolve the issue, making this a viable option.

Another potential for further performance improvements lies in bandwidth conserving techniques like REC12. Together with Pinke et al., I have already shown the principal feasibility of this approach in 'Lattice QCD based on OpenCL' [20], however—for a long time fighting with compiler problems regarding register usage—have not yet come around to completely integrate it into the main version of the application.

To reduce the wall time of the calculations it might also be worthwhile to utilize GPUs from multiple nodes. This requires to extend the application to use multiple host processes, extending the existing device-to-device communication methods to include MPI transfers. Given the performance limitations presented in Section 5.2, such an implementation would also need to use a more advanced inverter algorithm. One such

solver could be GCR-DD, which has already been used by Babich *et al.* on the Edge cluster at Lawrence Livermore National Laboratory (LLNL) [2].

In terms of hardware it would be interesting to evaluate the use of APUs and similar hybrid processors. Given the bandwidth requirements of LQCD, those would probably require on-board, GPU-like memory as found in the Sony Playstation 4. Combined with an interconnect this could lead to architectures similar to the QPACE system and avoid the QPI-PCIe interference observed in classical multi-socket, multi-GPU systems.

The work presented has already enabled new studies [91]. For these, calculations have been performed on the LOEWE-CSC and SANAM supercomputers. Now, work is under way to extend CL^{2}QCD's scope by adding further discretizations, namely staggered fermions. A prototype of the Langevin algorithm—an alternative method for ensemble generation—has been implemented in a day, reusing building blocks of the HMC algorithm. It shows the successful modularization of the architecture and profits from all the optimizations performed to speed up the $\not{D}$, the inverter and the HMC.

Appendix A.

LOEWE-CSC

LOEWE-CSC is a hybrid supercomputer at the Johann Wolfgang Goethe-Universität Frankfurt am Main. Built from commodity hardware, including AMD Radeon HD 5870 GPUs, and utilizing a highly optimized HPL [12] it can operate at 740 MFLOPS/W. Thereby it ranked eighth in the Green500 list of November 2010. It is hosted in a highly efficient data centre, which features an exceptionally low cooling overhead of less than 8 % [10].

As LOEWE-CSC is a university system, it had to be designed to cope with the diverse set of requirements arising from a large number of heterogeneous applications. Besides the LQCD computations, which are covered in detail in the main part of this thesis, there is also the simulation of heavy-ion collisions using Monte-Carlo transport simulations. While these simulations have recently been ported to GPU, they traditionally require a large number of single-threaded jobs [102]. Climate model calculations require petabytes of storage to store their highly resolved data. Yet, other applications have further requirements. This heterogeneous set of requirements enforced a general purpose machine.

LOEWE-CSC is implemented as a hybrid cluster of 826 compute nodes in 34 racks. For details see Table A.1. It has two node types: The details of the GPU nodes can be found in Table A.2. For jobs requiring large amounts of main memory there are also nodes without GPUs but more main memory and four instead of two CPUs. These are called quad nodes. Their details can be found in Table A.3. The resources of the machine are managed via the Simple Linux Utility for Resource Management (SLURM).

Table A.1.: Key data of the LOEWE-CSC supercomputer

Compute nodes	826
GPU nodes	786
CPU nodes	40
Total CPU cores	20768
Total GPUs	786
Total main memory	55.4 TiB
Peak performance (DP)	599 TFLOPS
Peak performance (SP)	2.45 PFLOPS
Shared storage	2 PB

Table A.2.: LOEWE-CSC GPU node data

Server	SuperServer 2022TG-HIBQRF
CPUs	2 AMD Opteron 6172
Total cores	24
CPU clock speed	2.1 GHz
GPUs	1 AMD Radeon HD 5870
Main memory	64 GiB
Interconnect	on-board quad data rate (QDR) IB

Table A.3.: LOEWE-CSC quad node data

Server	SuperServer 2042G-TRF
CPUs	4 AMD Opteron 6172
Total cores	48
CPU clock speed	2.1 GHz
Main memory	128 GiB
Interconnect	on-board QDR IB

Appendix B.

SANAM

SANAM, built by an international collaboration of research groups from FIAS and KACST, is the prototype of a general-purpose 10 PFLOPS supercomputer. Similar to LOEWE-CSC it is based on off-the-shelf components. However, contrary to that older system it features four GPUs in each node instead of just one. Utilizing a highly optimized HPL implementation [103] it can perform at 2351 MFLOPS/W, ranking second in the Green500 list of November 2012. Its overall HPL performance is 532 TFLOPS. The key data of the system can be found in Table B.1. Details about the nodes can be found in Table B.2. The resources of the machine are managed via SLURM. Details on how the GPU scheduling is implemented can be found in Appendix D.

Table B.1.: Key data of the SANAM supercomputer

Compute nodes	304
Total CPU cores	4468
Total GPUs	1216
Total main memory	38.9 TiB
Peak performance (DP)	1112.8 TFLOPS
Peak performance (SP)	4295.9 TFLOPS

Table B.2.: SANAM compute node data

Server	ASUS ESC4000/FDR G2
CPUs	2 Intel Xeon E5-2650
Total cores	16
CPU clock speed	2.0 GHz
GPUs	4 AMD FirePro S10000
Main memory	128 GiB
Interconnect	on-board fourteen data rate (FDR) IB

Appendix C.

Development and Test Systems

This appendix gives an overview over the technical specifications of the development systems used during the development of CL^{2}QCD. I report on their specifications as I have used them for some of the performance measurements in this thesis.

C.1. gpu-dev00

gpu-dev00 has been the primary development system for any NVIDIA targeted development. It is a consumer system that has been equipped with a variety of NVIDIA GPUs.

Table C.1.: gpu-dev00's specification

Motherboard ASUS P6T7 WS SUPERCOMPUTER

CPUs Intel Nehalem Core i7-930

CPU Family Nehalem

Total Cores 4

CPU clock speed 2.8 GHz (up to 3.06 GHz in turbo mode)

Memory 12 GiB DDR3 at 1333 MHz

Maximum Number of GPUs 3

Operating System Ubuntu Linux

C.2. gpu-dev01

gpu-dev01 is a server system similar to the nodes used in LOEWE-CSC, however, based on Intel instead of AMD CPUs. It has been used for a lot of development targeting Cypress GPUs. For a long while it has been equipped with an AMD FirePro V7800.

Table C.2.: gpu-dev01's specification

Server Supermicro 827H-R1400B

Motherboard Supermicro X8DTT-IBX

CPUs 2 Intel Xeon E5520

CPU Family Nehalem

Total Cores 8

CPU clock speed 2.27 GHz

Memory 24 GiB DDR3 at 1066 MHz

Maximum Number of GPUs 1

Operating System Ubuntu Linux

C.3. gpu-dev03

gpu-dev03 is a workstation that has mostly been equipped with a varying number of AMD Radeon HD 7970 GPUs.

Table C.3.: gpu-dev03's specification

Workstation Supermicro AS-4022G

Motherboard Supermicro H8DG6

CPUs 2 AMD Opteron 6278

CPU Family Piledriver

Total Cores 32

CPU clock speed 2.4 GHz (up to 3.3 GHz in turbo mode)

Memory 128 GB DDR3 at 1600 MHz

Maximum Number of GPUs 3

Operating System openSUSE 12.1

C.4. gpu-dev04

gpu-dev04 is a workstation that has been been equipped with a varying number of AMD Radeon HD 7970 GPUs, but also hosted AMD Radeon HD 6970 and AMD Radeon HD 5870 GPUs.

Table C.4.: gpu-dev04's specification

Workstation Supermicro AS-4022G

Motherboard Supermicro H8DG6

CPUs 2 AMD Opteron 6172

CPU Family K10 (Magny-Cours)

Total Cores 24

CPU clock speed 2.1 GHz

Memory 64 GB DDR3 at 1333 MHz

Maximum Number of GPUs 3

Operating System openSUSE 12.1

C.5. titanic

titanic is a server system that has not been equipped with GPUs, but was used as a reference system to execute CPU codes.

Table C.5.: titanic's specification

Server Supermicro AS-1022GG-TF

Motherboard Supermicro H8DGG-QF

CPUs 2 AMD Opteron 6220

CPU Family Bulldozer

Total Cores 16

CPU clock speed 3.0 GHz (up to 3.6 GHz in turbo mode)

Memory 64 GB DDR3 at 1333 MHz

Maximum Number of GPUs 2

Operating System openSUSE 12.1

Appendix D.

Scheduling GPUs with SLURM

In every system that allows the execution of multiple applications an important question arises: Which application may use which resource at a given point in time?

For CPUs this problem is solved by the OS and cluster resource managers like the Simple Linux Utility for Resource Management (SLURM). The cluster resource manager sends the individual jobs, in which applications are executed, to the individual nodes of the cluster and instructs the OS on which CPU cores to run the application, how much memory the application may consume, and so on. This way it avoids that two applications compete for the same resource and optimum throughput of the cluster is ensured.

Once GPUs come into play this becomes more difficult. An OS does not schedule GPUs. All applications running on a given node can access all of its GPUs. While two applications using the same GPU will observe increased latencies in GPU kernel completion, this is not the major problem. As there is no concept of swappable memory for GPUs, this can also lead to unexpected application crashes if two applications access the same GPU.

On systems like LOEWE-CSC, which only contain a single GPU, this problem can be avoided by always scheduling only a single job on each node. This way an application can always be sure that it has exclusive access to the GPU. On a system with more than one GPU per node such an approach could, however, lead to a rather suboptimal distribution of resources. Even a job that only requires a single GPU would be scheduled four GPUs.

Therefore, the cluster resource manager must also treat the GPUs as consumable resources, just as CPUs and system memory. The following sections describe how to achieve this using SLURM for GPUs by NVIDIA and AMD.

The solution presented here has been developed for SLURM version 2.4.4. Using this version it has been tested extensively on SANAM. It should also work on the current version 2.6.0 but has not been tested with that version.

D.1. Scheduling NVIDIA GPUs

SLURM provides an out-of-the-box support for scheduling NVIDIA GPUs. It is enabled by selecting the consumable resources scheduling plug-in and defining the GPUs as consumable resources. The entries required into the main configuration file `slurm.conf`

Listing D.1: Required entries in the SLURM configuration file to use four GPUs as consumable resources

```
GresTypes=gpu
NodeName=<list of nodes> Gres=gpu:4
SelectType=select/cons_res
```

Listing D.2: Content of `gres.conf` on a system with four NVIDIA GPUs

```
Name=gpu File=/dev/nvidia[0-3]
```

are shown in Listing D.1. The example is taken from SANAM and, therefore, assumes four GPUs per node.

Each node that provides a consumable resource requires an additional SLURM configuration file called `gres.conf`. Listing D.2 gives an example for a node with four GPUs. The purpose of specifying the device files is that SLURM will not use the node in case a device is missing.

As mentioned before, the operating system does not handle access of the applications to the GPUs. Therefore, SLURM utilizes a functionality of the NVIDIA device driver to restrict access to the GPUs. If the environment variable `CUDA_VISIBLE_DEVICES` is set to a comma separated list of device indices, the NVIDIA device driver will only display those devices to the application. When `CUDA_VISIBLE_DEVICES=2,3` is set, an application will only see the third and fourth GPU in the system. This way it will be unable to conflict with applications running on the first or second GPU.[1]

D.2. Scheduling AMD GPUs

SLURM does not provide out-of-the-box support for scheduling AMD GPUs. However, as the AMD device driver knows a functionality that works the same way as `CUDA_VISIBLE_DEVICES` works for NVIDIA GPUs, this functionality can easily be built on top of the NVIDIA GPU scheduling shown in Section D.1.

The file `gres.conf` should look as given in Listing D.3 according to SLURM's documentation on consumable resources [104]. However, in this case, SLURM will set `CUDA_VISIBLE_DEVICES` to nonsense values like `0,0,1,0`. Therefore, we have to make SLURM believe it is actually using NVIDIA GPUs and use the `gres.conf` given in Listing D.2.

As the NVIDIA device files do not exist in a system with AMD GPUs these have to be created manually. To keep the functionality of SLURM checking whether the devices are actually existing, it is useful to create the NVIDIA dummy device files based on the actual device files of the AMD GPUs. A possible implementation for this is shown

[1] Of course there can still be conflicts caused in other areas of the system, but not because the two applications are using the same GPU.

Listing D.3: Naïve version of `gres.conf` on a system with four AMD GPUs

```
Name=gpu File=/dev/ati/card[0-3]
```

Listing D.4: A script to create mockup NVIDIA device files based on the AMD GPUs in a system

```
#!/bin/bash
for FILE in /dev/ati/card*; do
  touch "/dev/nvidia${FILE#/dev/ati/card}";
done
```

in Listing D.4. This can be included into the SLURM initialization file right before the invocation of the actual SLURM daemon. The AMD device files are not existent until the AMD device driver is loaded, which is usually triggered by X. Therefore, the start-up of SLURM should also be made dependent on X.

Finally, the variable `CUDA_VISIBLE_DEVICES` needs to be transformed to its AMD equivalent `GPU_DEVICE_ORDINAL`. This can be achieved by the script shown in Listing D.5. It simply assigns the value of the NVIDIA specific variable to the AMD specific one. By setting it to an invalid value in case the NVIDIA variable is empty it will mask out all GPUs for jobs that did not demand any. This is required, as the AMD device driver will show all devices to an application if the variable `GPU_DEVICE_ORDINAL` is set to an empty value. The name of the script has to be given as the value of `TaskProlog` in the SLURM configuration file.

Note that it would of course have been possible to modify SLURM itself to fully support AMD GPUs. However, until that patch would have been merged back into the upstream version of SLURM, it would have to be ported every time a new version is released. With the purely configuration based solution presented here, updates can be applied without any additional overhead.

D.3. Known Issues of the Current Implementation

The current method of scheduling GPUs via SLURM is not perfect. Using this implementation for multiple months of data production I ran into several limitations.

SLURM sometimes gets confused as to which GPUs are already allocated and allocates the same GPU to multiple jobs. I have observed this behaviour in high load scenarios, that is, when starting many jobs at the same time. My workaround for the problem is to always add a delay of thirty seconds between launching of jobs.

The issue also affects returning nodes—or the whole cluster—into service after maintenance. When adding single nodes the above workaround still works. Thus, to return multiple nodes into service, I hold waiting jobs and release them with the same thirty second pause between two jobs. When restarting the whole cluster after maintenance,

Listing D.5: SLURM task prolog script for AMD GPUs

```
#!/bin/sh
#
# A TaskProlog script that will port the CUDA device
# restrictions to AMD.
#

if [ "X${CUDA_VISIBLE_DEVICES}" == "X" ]
then
  # mask all GPU devices
  echo "export GPU_DEVICE_ORDINAL=-1"
else
  echo "export COMPUTE=:0"
  echo "export GPU_DEVICE_ORDINAL=$CUDA_VISIBLE_DEVICES"
fi
```

however, holding the jobs does not help. To me it seems that SLURM in that case completely forgets to mark the GPUs as allocated. As our focus was on getting data from our simulations, I was unable to investigate this further but just created a script that kills all pending jobs and recreates them after maintenance.

While not an error, a feature that I would really love to see in SLURM is the capability to drain a single GPU instead of a whole node. Draining, in this context, means to prevent it from accepting further jobs. As we were already using the cluster during commissioning phase there were still some hardware issues. If a GPU started hanging up or produced calculation errors, I had to disable the whole node instead of the single GPU. This way we lost four GPUs instead of just one until the node was repaired.

Another feature lacking from the version of SLURM used is the capability to show the GPUs allocated to a certain job, or a list of jobs ordered by GPUs. This feature would have two major advantages: In the case of a failing GPU it would allow to quickly check which jobs are effected. When starting up nodes, it would allow to quickly check proper allocation of GPUs. As a workaround I always had each job print its allocated GPUs at the top of its output via a simple `echo $GPU_DEVICE_ORDINAL`.

While the other issues mostly effect error states, there is also a missing feature that effects normal job execution. The scheduler is unaware of the PCIe topology. Thus, if a mixture of single- and dual-GPU jobs is queued, the scheduler will also allocate GPUs on two different AMD FirePro S10000s to a dual-GPU job. As shown in Chapter 3, Chapter 4, and Chapter 5 this results in reduced performance. The current workaround is to only to use single or four GPU assignments.

Bibliography

[1] R. Gupta, *Introduction to Lattice QCD*. Jul. 1998, p. 150. arXiv: 9807028 [hep-lat] (cit. on pp. 1, 3, 4).

[2] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower and S. Gottlieb, 'Scaling Lattice QCD beyond 100 GPUs', *Sciences-New York*, p. 11, Sep. 2011. arXiv: 1109.2935 (cit. on pp. 1, 2, 16, 118, 130, 135, 167).

[3] K. Jansen and C. Urbach, 'tmLQCD: a program suite to simulate Wilson Twisted mass Lattice QCD', *Quantum*, no. May 2009, pp. 1–44, 2009 (cit. on pp. 1, 11, 75).

[4] D. Padua, Ed., *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4 (cit. on pp. 1, 17).

[5] C. Battista, S. Cabasino, F. Marzano, P. S. Paolucci, J. Pech, F. Rapuano, R. Sarno, G. M. Todesco, M. Torelli, W. Tross, P. Vicini, N. Cabibbo, E. Marinari, G. Parisi, G. Salina, F. del Rete, A. Lai, M. P. Lombardo, R. Tripiccione and A. Fucci, 'The APE-100 Computer: (I) The Architecture', *International Journal of High Speed Computing*, vol. 05, no. 04, pp. 637–656, Dec. 1993, ISSN: 0129-0533. DOI: 10.1142/S0129053393000268 (cit. on pp. 1, 186).

[6] P. Boyle, D. Chen, N. Christ, M. A. Clark, S. Cohen, A. Gara, L. Levkova, R. Mawhinney, S. Ohta, K. Petrov, T. Wettig, A. Yamaguchi and C. Cristian, 'QCDOC: A 10 Teraflops Computer for Tightly-Coupled Calculations', in *Proceedings of the ACM/IEEE SC2004 Conference*, vol. 00, IEEE, 2004, pp. 40–40, ISBN: 0-7695-2153-3. DOI: 10.1109/SC.2004.46 (cit. on p. 1).

[7] H. Baier, H. Boettiger, M. Drochner, N. Eicker, U. Fischer, Z. Fodor, A. Frommer, C. Gomez, G. Goldrian, S. Heybrock, D. Hierl, M. Hüsken, T. Huth, B. Krill, J. Lauritsen, T. Lippert, T. Maurer, B. Mendl, N. Meyer, A. Nobile, I. Ouda, M. Pivanti, D. Pleiter, M. Ries, A. Schäfer, H. Schick, F. Schifano, H. Simma, S. Solbrig, T. Streuer, K. .-.-H. Sulanke, R. Tripiccione, J. .-.-S. Vogt, T. Wettig and F. Winter, 'QPACE – a QCD parallel computer based on Cell processors', p. 21, Nov. 2009. arXiv: 0911.2174 (cit. on pp. 1, 14).

[8] P. Vranas, 'QCD and the BlueGene', *Journal of Physics: Conference Series*, vol. 78, p. 012080, Jul. 2007, ISSN: 1742-6588. DOI: 10.1088/1742-6596/78/1/012080 (cit. on pp. 1, 186).

[9] H. Meuer, E. Strohmaier, J. Dongarra and H. Simon, *Top 500 Supercomputing Sites*. [Online]. Available: http://top500.org (cit. on p. 1).

[10] M. Bach, J. de Cuveland, H. Ebermann, D. Eschweiler, J. Gerhard, S. Kalcher, M. Kretz, V. Lindenstruth, H.-J. Ludde, M. Pollok and D. Rohr, 'A Comprehensive Approach for a Power Efficient General Purpose Supercomputer', in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, IEEE, Feb. 2013, pp. 336–342, ISBN: 978-1-4673-5321-2. DOI: 10.1109/PDP.2013.55 (cit. on pp. 1, 137).

[11] W.-c. Feng and K. W. Cameron, *The Green500*. [Online]. Available: http://www.green500.org (cit. on p. 2).

[12] M. Bach, M. Kretz, V. Lindenstruth and D. Rohr, 'Optimized HPL for AMD GPU and multi-core CPU usage', *Computer Science - Research and Development*, vol. 26, no. 3-4, pp. 153–164, Apr. 2011, ISSN: 1865-2034. DOI: 10.1007/s00450-011-0161-5 (cit. on pp. 2, 137).

[13] D. Rohr, 'On Development, Feasibility, and Limits of Highly Efficient CPU and GPU Programs in Several Fields', PhD thesis, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 2013 (cit. on pp. 2, 57).

[14] P. Rogers, J. Macri and S. Marinkovic, *AMD Heterogeneous Uniform Memory Access*, 2013. [Online]. Available: http://www.slideshare.net/AMD/amd-heterogeneous-uniform-memory-access (cit. on p. 2).

[15] C. Bonati, G. Cossu, M. D'Elia and P. Incardona, 'QCD simulations with staggered fermions on GPUs', p. 22, Jun. 2011. DOI: 10.1016/j.cpc.2011.12.011. arXiv: 1106.5673 (cit. on pp. 2, 17, 28, 117).

[16] R. Babich, M. A. Clark and B. Joó, 'Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics', in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2010, pp. 1–11, ISBN: 978-1-4244-7557-5. DOI: 10.1109/SC.2010.40. arXiv: 1011.0024 (cit. on pp. 2, 16, 28, 130).

[17] M. A. Clark, R. Babich, K. Barros, R. Brower and C. Rebbi, 'Solving Lattice QCD systems of equations using mixed precision solvers on GPUs', *Computer Physics Communications*, p. 30, Nov. 2010, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2010.05.002. arXiv: 0911.3191 (cit. on pp. 2, 5, 16, 28, 35, 112, 113, 134).

[18] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi and K. K. Szabo, 'Lattice QCD as a video game', *Computer Physics Communications*, vol. 177, no. 8, p. 11, Nov. 2006, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2007.06.005. arXiv: 0611022 [hep-lat] (cit. on pp. 2, 15).

[19] C. Pinke, O. Philipsen, C. Schäfer, L. Zeidlewicz and M. Bach, 'LatticeQCD using OpenCL', Dec. 2011. arXiv: 1112.5280 (cit. on pp. 3, 75, 77, 78, 134, 188).

[20] M. Bach, V. Lindenstruth, O. Philipsen and C. Pinke, 'Lattice QCD based on OpenCL', *Computer Physics Communications*, p. 19, Mar. 2013, ISSN: 00104655. DOI: 10.1016/j.cpc.2013.03.020. arXiv: 1209.5942 (cit. on pp. 3, 35, 75, 115, 134, 188).

[21] A. Petitet, R. C. Whaley, J. Dongarra and A. Cleary, *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. [Online]. Available: http://www.netlib.org/benchmark/hpl/ (cit. on p. 3).

[22] G. P. Lepage, *Lattice QCD for Novices*, 1. 2005, pp. 1–42 (cit. on p. 4).

[23] M. Creutz, *Quarks, Gluons and Lattices*. Cambridge University Press, 1983, ISBN: 0521244056 (cit. on p. 4).

[24] ——, 'Monte Carlo study of quantized SU(2) gauge theory', *Physical Review D*, vol. 21, no. 8, pp. 2308–2315, Apr. 1980, ISSN: 0556-2821. DOI: 10.1103/PhysRevD.21.2308 (cit. on p. 6).

[25] N. Cabibbo and E. Marinari, 'A new method for updating SU(N) matrices in computer simulations of gauge theories', *Physics Letters B*, vol. 119, no. 4-6, pp. 387–390, Dec. 1982, ISSN: 03702693. DOI: 10.1016/0370-2693(82)90696-7 (cit. on p. 6).

[26] A. Kennedy and B. Pendleton, 'Improved heatbath method for Monte Carlo calculations in lattice gauge theories', *Physics Letters B*, vol. 156, no. 5-6, pp. 393–399, Jun. 1985, ISSN: 03702693. DOI: 10.1016/0370-2693(85)91632-6 (cit. on p. 6).

[27] S. Duane, A. Kennedy, B. J. Pendleton and D. Roweth, 'Hybrid Monte Carlo', *Physics Letters B*, vol. 195, no. 2, pp. 216–222, Sep. 1987, ISSN: 03702693. DOI: 10.1016/0370-2693(87)91197-X (cit. on p. 6).

[28] T. Takaishi and P. de Forcrand, 'Testing and tuning symplectic integrators for the hybrid Monte Carlo algorithm in lattice QCD', *Physical Review E*, vol. 73, no. 3, p. 036 706, Mar. 2006, ISSN: 1539-3755. DOI: 10.1103/PhysRevE.73.036706 (cit. on p. 6).

[29] A. Meister, *Numerik linearer Gleichungssysteme*. Wiesbaden: Vieweg, 2008, ISBN: 9783834815507. DOI: 10.1007/978-3-8348-8100-7 (cit. on p. 8).

[30] T. a. Degrand and P. Rossi, 'Conditioning techniques for dynamical fermions', *Computer Physics Communications*, vol. 60, no. 2, pp. 211–214, Sep. 1990, ISSN: 00104655. DOI: 10.1016/0010-4655(90)90006-M (cit. on p. 8).

[31] C. Urbach, K. Jansen, A. Shindler and U. Wenger, 'HMC algorithm with multiple time scale integration and mass preconditioning', *Computer Physics Communications*, vol. 174, no. 2, pp. 87–98, Jan. 2006, ISSN: 00104655. DOI: 10.1016/j.cpc.2005.08.006. arXiv: 0506011v2 [arXiv:hep-lat] (cit. on p. 9).

[32] K. Wilson, 'Confinement of quarks', *Physical Review D*, vol. 10, no. 8, pp. 2445–2459, Oct. 1974, ISSN: 0556-2821. DOI: 10.1103/PhysRevD.10.2445 (cit. on p. 9).

[33] L. Susskind, 'Lattice fermions', *Physical Review D*, vol. 16, no. 10, pp. 3031–3039, Nov. 1977, ISSN: 0556-2821. DOI: 10.1103/PhysRevD.16.3031 (cit. on p. 9).

[34] P. M. Vranas, 'Chiral symmetry restoration in the Schwinger model with domain wall fermions', *Physical Review D*, vol. 57, no. 3, pp. 1415–1432, Feb. 1998, ISSN: 0556-2821. DOI: 10.1103/PhysRevD.57.1415 (cit. on p. 9).

[35] H. Neuberger, 'Exactly massless quarks on the lattice', *Physics Letters B*, vol. 417, no. 1-2, pp. 141–144, Jan. 1998, ISSN: 03702693. DOI: 10.1016/S0370-2693(97)01368-3. arXiv: 9707022 [hep-lat] (cit. on p. 9).

[36] A. Shindler, 'Twisted mass lattice QCD', *Physics Reports*, vol. 461, no. 2-3, pp. 37–110, May 2008, ISSN: 03701573. DOI: 10.1016/j.physrep.2008.03.001 (cit. on p. 9).

[37] A. Ukawa, 'Computational cost of full QCD simulations experienced by CP-PACS and JLQCD Collaborations', *Nuclear Physics B - Proceedings Supplements*, vol. 106-107, no. 2, pp. 195–196, Mar. 2002, ISSN: 09205632. DOI: 10.1016/S0920-5632(01)01662-0 (cit. on p. 10).

[38] C. Bernard, T. Burch, T. DeGrand, C. DeTar, S. Gottlieb, U. Heller, J. Hetrick, L. Levkova, C. McNeile, K. Orginos, J. Osborn, K. Rummukainen, B. Sugar and D. Toussaint, *The MILC Code*, 2010. [Online]. Available: http://www.physics.utah.edu/~detar/milc/milcv7.pdf (cit. on p. 11).

[39] R. G. Edwards and B. Joo, 'The Chroma Software System for Lattice QCD', pp. 7–9, Sep. 2004. DOI: 10.1016/j.nuclphysbps.2004.11.254. arXiv: 0409003 [hep-lat] (cit. on p. 11).

[40] *The Columbia Physics System*, 2010. [Online]. Available: http://phys.columbia.edu/~cqft/physics%5C_sfw/physics%5C_sfw.htm (cit. on p. 11).

[41] M. Smelyanskiy, K. Vaidyanathan, J. Choi, B. Joó, J. Chhugani, M. A. Clark and P. Dubey, 'High-performance lattice QCD for multi-core based parallel systems using a cache-friendly hybrid threaded-MPI approach', *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, p. 1, 2011. DOI: 10.1145/2063384.2063477 (cit. on pp. 11, 60, 83, 113).

[42] N. Avico, P. Bacilieri, S. Cabasino, N. Cabibbo, L. Fernández, G. Fiorentini, A. Lai, M. Lombardo, E. Marinari, F. Marzano, P. Paolucci, G. Parisi, J. Pech, F. Rapuano, E. Remiddi, R. Sarno, G. Salina, A. Tarancón, G. Todesco, M. Torelli, R. Tripiccione and W. Tross, 'From APE to APE-100: From 1 to 100 gflops in lattice gauge theory simulations', *Computer Physics Communications*, vol. 57, no. 1-3, pp. 285–289, Dec. 1989, ISSN: 00104655. DOI: 10.1016/0010-4655(89)90229-4 (cit. on p. 11).

[43] N. Cabibbo, F. Rapuano and R. Tripiccione, *An Introduction to the APE100 Computer* (cit. on p. 11).

[44] F. Belletti, S. Schifano, R. Tripiccione, F. Bodin, P. Boucaud, J. Micheli, O. Pene, N. Cabibbo, S. de Luca, A. Lonardo, D. Rossetti, P. Vicini, M. Lukyanov, L. Morin, N. Paschedag, H. Simma, V. Morenas, D. Pleiter and F. Rapuano, 'Computing for LQCD: apeNEXT', *Computing in Science & Engineering*, vol. 8, no. 1, pp. 18–29, Jan. 2006, ISSN: 1521-9615. DOI: 10.1109/MCSE.2006.4 (cit. on p. 12).

[45] D. Chen, P. Chen, N. H. Christ, R. G. Edwards, G. Fleming, A. Gara, S. Hansen, C. Jung, A. Kahler, S. Kasow, A. D. Kennedy, G. Kilcup, Y. Luo, C. Malureanu, R. D. Mawhinney, J. Parsons, C. Sui, P. Vranas and Y. Zhestkov, 'QCDSP machines: design, performance and cost', in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '98, Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–6, ISBN: 0-89791-984-X. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509113 (cit. on pp. 12, 118).

[46] P. a. Boyle, D. Chen, N. H. Christ, M. A. Clark, S. D. Cohen, C. Cristian, Z. Dong, A. Gara, B. Joo, C. Jung, C. Kim, L. a. Levkova, X. Liao, G. Liu, R. D. Mawhinney, S. Ohta, K. Petrov, T. Wettig and A. Yamaguchi, 'Overview of the QCDSP and QCDOC computers', *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 351–365, Mar. 2005, ISSN: 0018-8646. DOI: 10.1147/rd.492.0351 (cit. on pp. 12, 13, 186).

[47] G. Bhanot, D. Chen, A. Gara, J. Sexton and P. Vranas, 'QCD on the BlueGene/L Supercomputer', *Nuclear Physics B - Proceedings Supplements*, vol. 140, pp. 823–825, Mar. 2005, ISSN: 09205632. DOI: 10.1016/j.nuclphysbps.2004.11.153 (cit. on p. 13).

[48] J. Doi, 'Peta-scale Lattice Quantum Chromodynamics on a Blue Gene/Q supercomputer', in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2012, ISBN: 978-1-4673-0806-9. DOI: 10.1109/SC.2012.96 (cit. on p. 13).

[49] P. Boyle, 'The BlueGene/Q supercomputer', in *The 30 International Symposium on Lattice Field ...*, 2012. [Online]. Available: http://pos.sissa.it/archive/conferences/164/020/Lattice%202012%5C_020.pdf (cit. on pp. 13, 114, 119).

[50] G. Shi, S. Gottlieb, A. Torok and V. Kindratenko, 'Design of MILC Lattice QCD Application for GPU Clusters', in *2011 IEEE International Parallel & Distributed Processing Symposium*, IEEE, May 2011, pp. 363–371, ISBN: 978-1-61284-372-8. DOI: 10.1109/IPDPS.2011.43 (cit. on p. 16).

[51] M. A. Clark and R. Babich, 'High-efficiency Lattice QCD computations on the Fermi architecture', in *2012 Innovative Parallel Computing (InPar)*, IEEE, May 2012, pp. 1–9, ISBN: 978-1-4673-2633-9. DOI: 10.1109/InPar.2012.6339591 (cit. on pp. 16, 113).

[52] K. Z. Ibrahim, F. Bodin and O. Pène, 'Fine-grained parallelization of lattice QCD kernel routine on GPUs', *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1350–1359, Oct. 2008, ISSN: 07437315. DOI: 10.1016/j.jpdc.2008.06.009 (cit. on p. 16).

[53] Y. Osaki and K.-i. Ishikawa, 'Domain Decomposition method on GPU cluster', in *The XXVIII International Symposium on Lattice Field Theory, Lattice2010*, 2010. [Online]. Available: http://pos.sissa.it/archive/conferences/105/036/Lattice%202010%5C_036.pdf (cit. on p. 17).

[54] T.-W. Chiu, T.-H. Hsieh and Y.-Y. Mao, 'Topological susceptibility in two flavors lattice QCD with the optimal domain-wall fermion', *Physics Letters B*, vol. 702, no. 2-3, pp. 131–134, Aug. 2011, ISSN: 03702693. DOI: 10.1016/j.physletb.2011.06.070. arXiv: 1105.4414 (cit. on p. 17).

[55] N. Cardoso and P. Bicudo, 'Generating SU(Nc) pure gauge lattice QCD configurations on GPUs with CUDA', p. 17, Dec. 2011. DOI: 10.1016/j.cpc.2012.10.002. arXiv: 1112.4533 (cit. on pp. 17, 28, 47, 111).

[56] A. Alexandru, M. Lujan, C. Pelissier, B. Gamari and F. Lee, 'Efficient Implementation of the Overlap Operator on Multi-GPUs', *2011 Symposium on Application Accelerators in High-Performance Computing*, pp. 123–130, Jul. 2011. DOI: 10.1109/SAAHPC.2011.13 (cit. on p. 17).

[57] A. Alexandru, C. Pelissier, B. Gamari and F. Lee, 'Multi-mass solvers for lattice QCD on GPUs', *Journal of Computational Physics*, vol. 231, no. 4, pp. 1866–1878, Feb. 2012, ISSN: 00219991. DOI: 10.1016/j.jcp.2011.11.003 (cit. on pp. 17, 28, 113).

[58] B. Jegerlehner, 'Krylov space solvers for shifted linear systems', no. December, pp. 1–16, Dec. 1996. arXiv: 9612014 [hep-lat] (cit. on p. 17).

[59] B. Joó, D. D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. W. Lee, P. Dubey and W. I. Watson, 'Lattice QCD on Intel Xeon Phi Coprocessors', in *Supercomputing*, J. M. Kunkel, T. Ludwig and H. W. Meuer, Eds., Springer Berlin Heidelberg, 2013, pp. 40–54, ISBN: 978-3-642-38750-0. DOI: 10.1007/978-3-642-38750-0_4 (cit. on pp. 17, 113, 114, 119, 120, 130).

[60] A. Kowalski and X. Shen, 'Implementing the Dslash Operator in OpenCL', Tech. Rep., 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.211.7733%5C&rep=rep1%5C&type=pdf (cit. on p. 17).

[61] P. Vranas, G. Bhanot, M. Blumrich, D. Chen, A. Gara, P. Heidelberger, V. Salapura and J. C. Sexton, 'The BlueGene/L supercomputer and quantum ChromoDynamics', in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06*, New York, New York, USA: ACM Press, 2006, p. 50, ISBN: 0769527000. DOI: 10.1145/1188455.1188507 (cit. on pp. 18, 118, 186).

[62] G. Hull, 'Life', *Amazing Computer Magazine*, vol. 2, pp. 81–84, 1987. [Online]. Available: http://www.archive.org/stream/amazing-computing-magazine-1987-12/Amazing%5C_Computing%5C_Vol%5C_02%5C_12%5C_1987%5C_Dec%5C#page/n81/mode/2up (cit. on p. 19).

[63] M. Pharr and R. Fernando, Eds., *Gpu Gems 2*. Amsterdam: Addison-Wesley Longman, 2005, ISBN: 978-0321335593. [Online]. Available: https://developer.nvidia.com/content/gpu-gems-2 (cit. on p. 19).

[64] *BrookGPU*. [Online]. Available: http://graphics.stanford.edu/projects/brookgpu/ (cit. on pp. 24, 26).

[65] *Aparapi*. [Online]. Available: http://code.google.com/p/aparapi/ (cit. on p. 28).

[66] B. Catanzaro, M. Garland and K. Keutzer, 'Copperhead', in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming - PPoPP '11*, New York, New York, USA: ACM Press, 2011, p. 47, ISBN: 9781450301190. DOI: 10.1145/1941553.1941562 (cit. on p. 28).

[67] D. Sharlet, 'Shevlin Park: Implementing C++ AMP with Clang/LLVM and OpenCL', in *LLVM Developer's Meeting*, 2012. [Online]. Available: http://llvm.org/devmtg/2012-11/Sharlet-ShevlinPark.pdf (cit. on p. 31).

[68] M. Bach, *clBandwidth*, 2012. [Online]. Available: https://github.com/theMarix/clBandwidth (cit. on pp. 35, 36, 134, 188).

[69] Advanced Micro Devices, 'AMD Accelerated Parallel Processing OpenCL™ Programming Guide (v2.8)', Tech. Rep. December, 2012 (cit. on pp. 38, 41, 42, 51, 59–61, 91, 95).

[70] *Cuda C Programming Guide*, 5.5, July. NVIDIA, 2013. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (cit. on pp. 41, 42).

[71] G. Pratx and L. Xing, 'GPU computing in medical physics: a review.', *Medical Physics*, vol. 38, no. 5, pp. 2685–2697, 2011. DOI: 10.1118/1.3578605 (cit. on p. 53).

[72] B. Schroeder, E. Pinheiro and W.-D. Weber, 'DRAM errors in the wild', in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems - SIGMETRICS '09*, New York, New York, USA: ACM Press, 2009, ISBN: 9781605585116. DOI: 10.1145/1555349.1555372. [Online]. Available: http://www.cs.toronto.edu/~bianca/papers/sigmetrics09.pdf (cit. on p. 54).

[73] S. Nakamoto, 'Bitcoin : A Peer-to-Peer Electronic Cash System', Tech. Rep., 2008. [Online]. Available: http://bitcoin.org/bitcoin.pdf (cit. on p. 55).

[74] *Phoenix Miner*, 2012. [Online]. Available: https://github.com/phoenix2/phoenix (cit. on p. 55).

[75] D. Rohr, 'ALICE TPC Online Tracking on GPU based on Kalman Filter', Diplomarbeit, Ruprecht-Karls-Universität Heidelberg, 2010 (cit. on p. 57).

[76] M. Bach, *pyclKernelAnalyzer*, 2012. [Online]. Available: https://github.com/theMarix/pyclKernelAnalyzer (cit. on pp. 59, 82, 134, 188).

[77] V. Volkov and J. Demmel, 'Benchmarking GPUs to tune dense linear algebra', in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2008, pp. 1–11, ISBN: 978-1-4244-2834-2. DOI: 10.1109/SC.2008.5214359 (cit. on pp. 59, 61).

[78] N. Govindaraju, S. Larsen, J. Gray and D. Manocha, 'A Memory Model for Scientific Algorithms on Graphics Processors', in *ACM/IEEE SC 2006 Conference (SC'06)*, IEEE, Nov. 2006, pp. 6–6, ISBN: 0-7695-2700-0. DOI: 10.1109/SC.2006.2 (cit. on p. 60).

[79] NVIDIA, 'CUDA C Programming Guide (v5.0)', Tech. Rep. October, 2012 (cit. on pp. 60, 61).

[80] M. Bach, *clPCI*, 2013. [Online]. Available: https://bitbucket.org/marix/clpci (cit. on pp. 61, 70, 71, 188).

[81] M. Harris, 'Optimizing Parallel Reduction in CUDA', Tech. Rep., 2010. [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/index.html (cit. on p. 64).

[82] M. Bach, *CLPCI2*, 2013. [Online]. Available: https://bitbucket.org/marix/clpci2 (cit. on p. 71).

[83] 'Developing a linux kernel module using RDMA for GPUDirect', NVIDIA, Tech. Rep. July, 2013. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma/index.html (cit. on p. 71).

[84] E.-M. Ilgenfritz, K. Jansen, M. P. Lombardo, M. Müller-Preussker, M. Petschlies, O. Philipsen and L. Zeidlewicz, 'Phase structure of thermal lattice QCD with N_{f}=2 twisted mass Wilson fermions', *Physical Review D*, vol. 80, no. 9, p. 094 502, Nov. 2009, ISSN: 1550-7998. DOI: 10.1103/PhysRevD.80.094502. arXiv: 0905.3112 (cit. on p. 75).

[85] O. Philipsen and L. Zeidlewicz, 'Cutoff effects of Wilson fermions on the QCD equation of state to O(gˆ2)', *Physical Review D*, vol. 81, no. 7, p. 9, Dec. 2008, ISSN: 1550-7998. DOI: 10.1103/PhysRevD.81.077501. arXiv: 0812.1177 (cit. on p. 75).

[86] F. Burger, E. .-.-M. Ilgenfritz, M. Kirchner, M. P. Lombardo, M. Muller-Preussker, O. Philipsen, C. Pinke, C. Urbach and L. Zeidlewicz, 'The thermal QCD transition with two flavours of twisted mass fermions', vol. 06, p. 11, Feb. 2011. arXiv: 1102.4530 (cit. on pp. 75, 117).

[87] B. Joo, B. Pendleton, A. D. Kennedy, A. C. Irving, J. C. Sexton, S. M. Pickles, S. P. Booth and U. Collaboration, 'Instability in the Molecular Dynamics Step of Hybrid Monte Carlo in Dynamical Fermion Lattice QCD Simulations', p. 22, May 2000. DOI: 10.1103/PhysRevD.62.114501. arXiv: 0005023 [hep-lat] (cit. on p. 76).

[88] M. Luescher, 'A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations', *Computer Physics Communications*, vol. 79, no. 1, pp. 100–110, Sep. 1993, ISSN: 00104655. DOI: 10.1016/0010-4655(94)90232-1. arXiv: 9309020 [hep-lat] (cit. on p. 82).

[89] I. U. Nikolaisen, *RANLUXCL*, 2011 (cit. on p. 82).

[90] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007, vol. 1, p. 1262, ISBN: 0521880688. [Online]. Available: http://www.nr.com/ (cit. on p. 82).

[91] C. Pinke and O. Philipsen, 'The nature of the Roberge-Weiss transition in N_f=2 QCD with Wilson fermions', in *The XXXI International Symposium on Lattice Field Theory, Lattice2013*, 2013. [Online]. Available: http://www.lattice2013.uni-mainz.de/presentations/2B/Pinke.pdf (cit. on pp. 97, 132, 135).

[92] G. M. Amdahl, 'Validity of the single processor approach to achieving large scale computing capabilities', *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, 1967. DOI: 10.1145/1465482.1465560 (cit. on pp. 99, 164).

[93] *AMD Developer Knowledge Base*. [Online]. Available: http://developer.amd.com/resources/documentation-articles/knowledge-base/ (cit. on p. 111).

[94] R. Baron, P. Boucaud, P. Dimopoulos, F. Farchioni, R. Frezzotti, V. Gimenez, G. Herdoiza, K. Jansen, V. Lubicz, C. Michael, G. Muenster, D. Palao, G. C. Rossi, L. Scorzato, A. Shindler, S. Simula, T. Sudmann, C. Urbach and U. Wenger, 'Light Meson Physics from Maximally Twisted Mass Lattice QCD', no. November, p. 40, Nov. 2009. arXiv: 0911.5061 (cit. on p. 115).

[95] J. Brodkin, *IBM Drops Price on Supercomputer*, 2007. [Online]. Available: http://www.pcworld.com/article/135334/article.html (visited on 06/07/2013) (cit. on p. 118).

[96] The University of Tennessee, *UT Wins $65M National Science Foundation Supercomputing Grant For Next-Generation Computing System*, 2008. [Online]. Available: http://www.tennessee.edu/media/kits/nsf/index.html (visited on 06/07/2012) (cit. on p. 118).

[97] J. Brodkin, *With 16 petaflops and 1.6M cores, DOE supercomputer is world's fastest*, 2012. [Online]. Available: http://arstechnica.com/information-technology/2012/06/with-16-petaflops-and-1-6m-cores-doe-supercomputer-is-worlds-fastest/ (visited on 06/07/2012) (cit. on p. 119).

[98] S. Kalcher, D. Rohr, M. Bach, A. A. Alaqeeli, H. M. Alzaid, V. Lindenstruth, S. B. Alkhereyf, A. Alharthi, A. Almubarak, I. Alqwaiz, R. Bin Suliman and D. Eschweiler, 'SANAM: An Energy- and Cost-Efficient Multi-GPU Supercomputer', 2013 (cit. on p. 119).

[99] N. Ernst, *Intels erste GPU-Beschleuniger ab 2.000 US-Dollar*, 2012. [Online]. Available: http://www.golem.de/news/xeon-phi-3100-und-5110p-intels-erste-gpu-beschleuniger-ab-2-000-us-dollar-1211-95669.html (visited on 06/07/2012) (cit. on p. 119).

[100] *ZES Zimmer LMG 95*. [Online]. Available: http://www.zes.com/english/products/single-phase-precision-power-analyzer-lmg95.html (visited on 25/09/2013) (cit. on p. 121).

[101] J. Gerhard, 'Refactoring the UrQMD model for many-core architectures', PhD thesis, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 2013. [Online]. Available: http://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/31346 (cit. on p. 134).

[102] J. Gerhard, V. Lindenstruth and M. Bleicher, 'Relativistic Hydrodynamics on Graphic Cards', p. 9, Jun. 2012. arXiv: 1206.0919 (cit. on p. 137).

[103] D. Rohr, M. Bach, M. Kretz and V. Lindenstruth, 'Multi-GPU DGEMM and High Performance Linpack on Highly Energy-Efficient Clusters', *IEEE Micro*, vol. 31, no. 5, pp. 18–27, Sep. 2011, ISSN: 0272-1732. DOI: 10.1109/MM.2011.66 (cit. on p. 141).

[104] *Consumable Resources in SLURM*. [Online]. Available: http://slurm.schedmd.com/cons%5C_res.html (cit. on p. 150).

[105] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov and A. Fasih, 'PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation', *Parallel Computing*, vol. 38, no. 3, pp. 157–174, Mar. 2012, ISSN: 01678191. DOI: 10.1016/j.parco.2011.09.001 (cit. on p. 172).

Glossary

C++ A programming language with support for object-oriented programming. 12–14, 28, 29, 78, 163, 166

C++11 The latest version of C++. 31, 71, 77

C++ AMP An extension of the C++ language for parallel computing. 31–33, 183

CL^{2}QCD is an OpenCL based implementation of twisted mass LQCD developed in the context of this thesis. iii, 2, 3, 11, 75–78, 81–84, 86–89, 97, 100–103, 105, 106, 109, 111, 114, 120, 121, 123, 124, 127, 128, 130, 132–135, 143, 174, 179, 180, 185, 188, 189

Johann Wolfgang Goethe-Universität Frankfurt am Main The University at which the work for this thesis was performed. 1, 75, 137

LOEWE-CSC A supercomputer equipped with AMD GPUs. For details see Appendix A. iii, 1, 2, 77, 109, 110, 115, 130, 132, 135, 137–139, 141, 144, 149, 164, 181

2MN Second order minimal. 6, 117

3dfx Interactive A GPU manufacturer. Build the Voodoo Graphics PCI GPU, which was the first successful consumer-GPU providing hardware-accelerated 3D graphics. 24, 175

above At a numerical index of higher numeric value. 87

ACM Association for Computing Machinery. 167

ALICE A Large Ion Collider Experiment. 57

ALU Algorithmic and Logic Unit. 24, 25

AMD is a manufacturer of GPUs and x86 CPUs. 1–3, 20, 24, 26, 29, 31, 38, 41, 42, 53, 56, 59, 60, 64, 65, 71, 73, 83, 93, 96, 109, 110, 118, 120, 133, 134, 144, 149–151, 163–170, 173

AMD FirePro AMD's series of GPUs for use in workstations and servers. 53, 109, 120

AMD FirePro S10000 The GPU that is used in SANAM. 2, 20, 26, 54–56, 61, 64–72, 76, 103, 106, 107, 109, 112–115, 117, 119, 120, 128–131, 133, 142, 152, 174, 178–180, 187, 189

AMD FirePro V7800 A GPU by AMD. Based on the Cypress chip, like the AMD Radeon HD 5870, it features 2 GB of memory instead of just 1 GB. 109, 115, 144

AMD Opteron 6172 A CPU by AMD. This CPU is used in LOEWE-CSC. 20, 133, 138, 139, 146

AMD Opteron 6220 A CPU by AMD. iii, 117, 121, 133, 147

AMD Opteron 6278 A CPU by AMD. 20, 145

AMD Radeon HD 5870 The GPU that is used in LOEWE-CSC. 2, 17, 20, 21, 23, 26, 38, 39, 47–51, 57, 58, 60, 89, 91–101, 109, 111–115, 117, 133, 134, 137, 138, 146, 164–166, 177–180, 187

AMD Radeon HD 6970 A GPU by AMD. Successor to the AMD Radeon HD 5870. 20, 23, 26, 38, 51, 53, 60, 97, 109, 111, 112, 146, 164, 178, 179

AMD Radeon HD 7970 A GPU by AMD. Successor to the AMD Radeon HD 6970. The first GPU to be based on the GCN architecture. iii, 20, 23, 26, 38, 41, 43, 44, 51, 56–59, 91, 95–97, 100, 101, 109, 112, 114, 117, 119–122, 133, 134, 145, 146, 174, 177–179, 187, 189

Amdahl's Law Amdahl stated, that the inherently serial part of an application limits the speed-up that can be achieved by parallelization of part of an application [92]. This can be generalized to general optimization problems, where the speed-up in the overall application is always limited by the proportion of the code which has not received optimization. 99

AoS A method of storing data in memory. See Subsection 3.1.4 for details. 45, 47, 82, 83, 88, 89, 91, 93, 133, 178, 179

APE A computer specialized on LQCD computations. 1, 11, 12, 18, 164, 186

APE-100 A computer specialized on LQCD computations. Successor of the APE. 11, 164

apeNEXT A computer specialized on LQCD computations. Successor of the APE and APE-100. 12

API Application Programming Interface. 2, 24, 29, 113, 166

APP Accelerated Processing Platform. 173

APU Accelerated Processing Unit. 73, 135

ARM A company that designs GPUs and reduced instruction set computing (RISC) CPUs. They license their designs for integration into actual chips by third parties. 27, 29, 169

ASIC Application-Specific Integrated Circuit. 12, 13

ATI A manufacturer of GPUs. Was acquired by AMD in 2006. 24

AVX Advanced Vector Extensions. 11

BAGEL A domain specific compiler for LQCD on Blue Gene systems. 14

below At a numerical index of smaller numeric value. 87

BiCGSTAB BiConjugate Gradient STABilized method. 8, 99, 130

Bitcoin A virtual currency. 55, 56

BLAS Basic Linear Algebra Subprograms. 27, 28, 30, 32–34, 183

Blue Gene A family of supercomputers by IBM. 1–3, 13, 14, 18, 119, 120, 165, 186, 189

Blue Gene/L The first generation of the Blue Gene family of supercomputers. 13, 14, 118

Blue Gene/Q The third generation of the Blue Gene family of supercomputers. 13, 14, 114, 118, 119

BrookGPU An early GPGPU programming model. 24

C A programming language. 12, 13, 45, 165

C99 The version of C on which OpenCL is based. 29, 60

CAL Compute Abstraction Layer. 26

Catalyst The proprietary driver provided by AMD for its GPUs. 93, 95–98, 100–102, 109, 179, 180

Cayman A GPU chip based on a VLIW architecture. It is used in the AMD Radeon HD 5870. 20, 42, 60

Cell Broadband Engine A microprocessor developed by IBM, Sony and Toshiba. Its most prominent use was in the Sony Playstation 3. 13, 14, 168, 173, 174

CERN The laboratory of the European Organization for Nuclear Research located near Geneva. iii, 185

CG Conjugate Gradients. 8, 16, 17, 99, 101, 102, 106, 114, 115, 119, 120, 128–133, 180

Cg A language for vertex and pixel shaders developed by NVIDIA. It can generate Direct3D or OpenGL shader programs. 15, 16, 28

Chroma A library for lattice field theory. 11, 16

Clang A C and C++ compiler based on LLVM. 31, 33

constant memory is a part of the GPU's memory that can only be written to from the host. It is main purpose is to provide efficient access to values that are accessed by all threads concurrently. 23

CPS Columbia Physics System. 11, 16

CPU Central Processing Unit. iii, 1–3, 11, 15, 17, 19–21, 23, 24, 27, 29, 31–33, 56, 59–65, 70, 71, 73, 77, 83, 91, 100, 101, 106, 110, 113–115, 117–126, 128, 129, 131, 133, 134, 137–139, 141–147, 149, 163, 164, 167–172, 174, 178, 180, 181

CU Compute Unit. 20, 21, 23, 25, 26, 57, 59–61, 95, 96, 169, 175

Cypress A GPU chip based on a VLIW architecture. It is used in the AMD Radeon HD 5870. 20, 26, 42, 60, 144, 164

DDR Double Data Rate. 13, 166

DDR2 The second generation of DDR memory. 14

DDR3 The third generation of DDR memory. 14, 143–147

Direct3D A graphics API by Microsoft. Dominant on the Windows OS. Originally it was also known as DirectX, which includes input, sound and 2D functionalities. 24, 25, 165

DirectGMA A technology to perform peer-to-peer data transfers in between AMD GPUs. 71–73, 105, 106, 128, 130, 179, 188

DP Double Precision. iii, 11, 14–16, 20, 25, 26, 76, 83, 89, 91, 110, 112–114, 119, 120, 130, 133, 134, 138, 141, 180

DSP Digital Signal Processor. 13

ECC In combination with memory used to designate memory that can detect and correct a certain kind of errors as it might be inflicted by background radiation or as a sign of memory becoming bad. 25, 53–56, 178, 187

Edge A cluster at LLNL equipped with NVIDIA Tesla M2050 GPUs. 135

EIB Element Interconnect Bus. 14

Evergreen A GPU architecture by AMD. 26

FAIR Facility for Antiproton and Ion Research. iii, 185

FDR A data rate of the IB interconnect. The signalling rate is about 14 Gbit/s. 142

Fermi A GPU architecture by NVIDIA. 16, 20, 25, 26, 53

FFT Fast Fourier-Transform. 19

FIAS Frankfurt Institute for Advanced Studies. 2, 109, 141

fixed-function pipeline A compute engine that performs a specific operation, in contrast to a processor that can run a variety of applications. 24

FMA Fused Multiply-Add. 11, 14

Fortran A programming language. 11, 14, 28

FPGA Field-Programmable Gate Array. 14, 15, 175

FPU Floating-Pointer Unit. 11, 13, 14, 23, 25, 26, 170

G80 A GPU architecture by NVIDIA. 25

GCN Graphics Core Next is a GPU architecture by AMD. 26, 164, 173, 174

GCR Generalized Conjugate Residual. 16, 167

GCR-DD A domain-decompositioned variant of the GCR algorithm [2]. 16, 130, 135

global memory is the normal main memory of the GPU that can be read and written to by all threads running on the GPU. This memory is also used to store data transferred to or from the CPU. 21, 23, 25, 29

GLSL OpenGL Shader Language. 31

Gordon Bell Prize A prize awarded each year by the Association for Computing Machinery (ACM) to recognize outstanding achievement in HPC. 12, 118

GPGPU General-Purpose computation on Graphics Processing Units. iii, 19, 25–27, 31–33, 53, 134, 165

GPU Graphics Processing Unit. iii, 1–3, 13, 15–21, 23–29, 31, 32, 35, 36, 38, 40–43, 49, 51, 53–57, 59–73, 75–78, 82–87, 89, 91, 95, 98–102, 104–107, 109–115, 117–126, 128–135, 137, 138, 141–147, 149–152, 163–175, 177–181, 183, 189

gpu-dev00 A GPU development system equipped with NVIDIA GPUs. For details see Table C.1. 143

gpu-dev01 A GPU development system equipped with AMD GPUs. For details see Table C.2. 144

gpu-dev03 A GPU development system equipped with AMD GPUs. For details see Table C.3. 145

gpu-dev04 A GPU development system equipped with AMD GPUs. For details see Table C.4. 81, 114, 117, 121, 124–128, 146, 180

GPUDirect A technology to perform peer-to-peer data transfers in between NVIDIA GPUs. 71

Green500 An alternative ranking of supercomputers according to energy-efficiency. The systems are ranked according to their performance in FLOPS/W when running the HPL benchmark. To qualify a system must be located in the TOP500. 2, 3, 14, 15, 137, 141, 169

GSI GSI Helmholtzzentrum für Schwerionenforschung GmbH. iii, 185

half-precision A floating point format using only 2 B. Usually only used as a storage format. 16

halo Extra cells stored on a device that are not part of the volume this device has responsibility for. They are used to allow a device to read into volume of neighbouring devices and have to be updated via copies from those. 86

HMC hybrid Monte Carlo. iii, 5–7, 9, 10, 17, 75–78, 96, 99, 102, 103, 109, 112, 114, 115, 117, 121–123, 126, 127, 133, 135, 177, 180, 181

HPC High-Performance Computing. 1, 3, 167

HPL The most popular implementation of the High Performance Computing Linpack Benchmark, which is the basis of the TOP500 list of supercomputers. `http://www.netlib.org/benchmark/hpl/`. 3, 15, 137, 141, 168

hyper-threading Intel's variant of SMT. 21, 27, 60

I/O Input/Output. 13, 15, 89, 99

IB InfiniBand. 17, 138, 139, 142, 167, 172

IBM International Business Machines Corporation. 13, 165, 171, 172

IBM PowerXCell 8i An enhanced version of Cell Broadband Engine processor used in the Sony Playstation 3. 14, 15

ILDG International Lattice Data Grid. 82

ILP Instruction Level Parallelism. 26, 59, 82

Intel is a manufacturer of x86 CPUs. 29, 31, 32, 60, 83, 110, 144, 168, 169

Intel Xeon E5-2650 A CPU by Intel. 142

Intel Xeon E5-2680 A CPU by Intel. 114, 119

Intel Xeon E5-2690 A CPU by Intel. 20

Intel Xeon E5520 A CPU by Intel. 144

Intel Xeon Phi An accelerator by Intel. Like a GPU based on a PCIe card, but using x86 cores. 1, 3, 17, 27, 32, 113, 114, 119, 130

Intel Xeon Phi 5110P An accelerator by Intel. 114, 119, 120

Intel Xeon Phi B1PRQ-7110P An accelerator by Intel. 113, 114

Intel Xeon X5680 A CPU by Intel. 113

Interlagos A CPU family by AMD. 117

KACST King Abdulaziz City for Science and Technology. 2, 141

Kepler A GPU architecture by NVIDIA. 20, 25, 171

kernel The atomic unit of application execution on a GPU. 27, 35, 36, 42, 47, 60, 77, 78, 80, 81

LHC Large Hadron Collider. iii, 185

Linux An open-source, Unix-like OS. 24

LLNL Lawrence Livermore National Laboratory. 135, 166

LLVM A collection of modular and reusable compiler and tool-chain technologies. Currently the basis of all major OpenCL compilers. 31, 33, 166

LMG95 A power meter that is accepted to be used for Green500 submissions. 121

local memory A small on-die memory local to each CU. It usually shares performance characteristics of the register file, but can be accessed by all threads of a work group. 23, 25, 26, 32, 175

lower An index in a specific direction that has a smaller numeric value. Given periodic boundary conditions this might also be a larger numeric value, for which incrementing by the given offset gets wrapped around by the modulo arithmetic. 87

LQCD Lattice QCD is the only a priory approach to describing the strong force. iii, 1–4, 10–18, 28, 35, 75, 76, 82, 86, 89, 91, 109, 118–120, 133, 135, 137, 163–165, 172, 174, 175, 181

Mali A GPU by ARM. 27

many-core A processor with more cores than a multi-core processor. Typical values are in the order of 20 to 64. Current multi-core processors have up to 12 cores. Many-cores are usually equipped with hundreds of FPUs, while multi-cores have significantly less than 100 FPUs. 17

MESI A method of managing cache lines in coherent multi-CPU systems. 14

MFC Memory Flow Controller. 15

Microsoft A soft- and hardware vendor. Mostly known for is OS and office suite. 31, 33, 166, 175

MILC MILC Lattice Computation. 11, 16

MIMD Multiple Instructions Multiple Data. 12

MMU Memory-Management Unit. 13

MPI Message Passing Interface. 13, 130, 134

Northern Islands A GPU architecture by AMD. 26

Numpy A Python library proving linear-algebra, statistics, plotting and other numerical functionality. 65

NVIDIA is a GPU manufacturer of GPUs. 2, 3, 16, 18–20, 24–26, 28, 29, 33, 38, 41–43, 51, 53, 60, 64, 71, 91, 109, 110, 113, 114, 117, 130, 133, 134, 143, 149–151, 165, 167–172, 174

NVIDIA CUDA is a GPU programming language and library by NVIDIA. 2, 16–19, 25, 28, 29, 33, 47, 71, 111, 113, 117, 130, 134, 172, 186

NVIDIA GeForce 3 A GPU by NVIDIA. 24

NVIDIA GeForce 7900 GTX A GPU by NVIDIA. 15

NVIDIA GeForce 8800 GTX A GPU by NVIDIA. 15, 17, 25

NVIDIA GeForce GTX 280 A GPU by NVIDIA. 16, 17, 20, 25, 113

NVIDIA GeForce GTX 285 A GPU by NVIDIA. 16, 130

NVIDIA GeForce GTX 295 A dual GPU graphics board by NVIDIA. 111

NVIDIA GeForce GTX 480 A GPU by NVIDIA. 20, 21, 23, 25, 38, 41, 109, 113, 170

NVIDIA GeForce GTX 580 A GPU by NVIDIA. Its chip is a shrink of that found in the NVIDIA GeForce GTX 480. 38, 109, 111, 170

NVIDIA GeForce GTX 680 A GPU by NVIDIA. Its the successor of the NVIDIA GeForce GTX 580. 20, 109, 112, 113

NVIDIA GeForce GTX Titan A GPU by NVIDIA. It is the top-of-the-line of the Kepler generation. 26

NVIDIA Quadro NVIDIA's series of GPUs for workstation graphics. 53

NVIDIA Tesla NVIDIA's series of GPUs for use a pure compute devices in workstations and servers. 25, 53

NVIDIA Tesla C1060 A GPU by NVIDIA. 17

NVIDIA Tesla K20 A GPU by NVIDIA. 20, 113, 114, 119, 120

NVIDIA Tesla M2050 A GPU by NVIDIA. 16, 166

NVIDIA Tesla M2070 A GPU by NVIDIA. 17

NVIDIA Tesla M2090 A GPU by NVIDIA. 113

NVIDIA Tesla S2050 A GPU by NVIDIA. 17, 117

OpenACC A pragma based framework for GPU computing. 32, 33, 183

OpenCL An open standard for GPU programming. iii, 2, 3, 17, 19–21, 23, 25–29, 31, 33, 36, 47, 59, 60, 64, 65, 70, 71, 73, 75, 77, 78, 80–84, 88, 98, 100, 111, 113, 117, 133, 134, 163, 165, 169, 172, 179, 185, 188, 189

OpenGL Open Graphics Library. 15, 16, 24–27, 31, 165, 171

OpenMP A pragma based framework for parallel computing on shared-memory architectures. 32–34, 183

OS Operating System. 14, 17, 31, 33, 149, 166, 169, 170, 175

PBO An OpenGL extension. 16

PC Personal Computer. 11, 27

PCIe Peripheral Component Interconnect Express. 17, 18, 61, 65, 70, 71, 107, 114, 126, 128, 132, 134, 135, 152, 169, 188

PE Processing Element. 20, 25, 26, 86

PGT Pure Gauge Theory. 6, 75

PIO Programmed Input/Output. 65, 70, 71, 73, 106

PowerPC Performance Optimization With Enhanced RISC — Performance Computing. 14

PowerPC 440 A 32-bit integer CPU by IBM. 13, 14

PowerPC A2 A CPU by IBM. 14

PPU Power Processing Unit. 14

private memory is a part of the GPU's memory that is partitioned among all threads running on the GPU. Each thread as exclusive use of a different part of this memory. 21, 23, 57

PRNG Pseudo-Random Number Generator. 75, 77, 82, 110, 172

PyOpenCL A module to utilize OpenCL from Python. For details see 'PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation' [105]. 29, 71

Python A scripting language. 28, 29, 36, 61, 65, 96, 170, 172

QCD Quantum Chromodynamics is the gauge theory describing the strong force. iii, 1, 3, 4, 6, 12, 75, 89, 91, 117, 120, 132, 133, 169, 185, 186

QCDOC A computer specialized on LQCD computations. 1, 13, 14, 186

QCDSP A computer specialized on LQCD computations. 12, 13, 118

QDR A data rate of the IB interconnect. The signalling rate is about 10 Gbit/s. 138, 139

QGP Quark Gluon Plasma. 75

QMP A message-passing library for LQCD applications. 13

QPACE A computer specialized on LQCD computations. 1, 14, 15, 135

QPI Quick Path Interconnect. 65, 71, 73, 106, 107, 128, 129, 131, 135, 180

QUDA is a LQCD framework for NVIDIA GPUs based on NVIDIA CUDA. 16, 113, 114

RAM Random Access Memory. 12, 13, 15, 18, 53

Ranlux A PRNG that can be used in a multi-threaded fashion. 82, 172

RANLUXCL An OpenCL based implementation of Ranlux. It is available at `https://bitbucket.org/ivarun/ranluxcl`. 82

RAS Restrictive Additive Schwarz. 17

REC12 A memory and bandwidth conserving technique of storing $SU(3)$ matrices. 4, 111, 113, 134, 189

RHMC Rational Hybrid Monte Carlo. 17

RISC Reduced Instruction Set Computing. 164

SANAM A supercomputer equipped with AMD GPUs. For details see Appendix B. iii, 2, 61–72, 75–77, 105, 109, 110, 114, 115, 117–119, 122, 124–133, 135, 141, 149, 150, 163, 178–180, 189

scratch register A register that is not located in the GPU's register file, but in the global memory of the GPU. Scratch registers incur high latencies when accessed, which often reduces application performance. 23, 57, 59, 82, 95, 97, 98, 103

SDK Software Development Kit. 173

SFU Special-Function Unit. 25, 26

SIMD Single Instruction Multiple Data. 11, 12, 14, 19–21, 27, 83

SIMT Single Instruction Multiple Threads. 20, 21

SLURM The Simple Linux Utility for Resource Management is an open-source workload manager that allows to schedule multiple jobs on a cluster, fairly distributing the resources between them. `http://slurm.schedmd.com/`. 137, 141, 149–152

SMT Simultaneous Multi-Threading. 14, 83, 168

SoA A method of storing data in memory. See Subsection 3.1.4 for details. 45, 47–56, 60, 83, 88–91, 93, 94, 133, 174, 178–181, 187

SoC System on a Chip. 13, 14

Sony One of the companies involved in the creation of the Cell Broadband Engine. 165, 173

Sony Playstation 3 A game console by Sony. 14, 165, 168

Sony Playstation 4 A game console by Sony. 24, 135

Southern Islands A GPU generation by AMD, based on the GCN architecture. 26

SP Single Precision. 11, 15–17, 20, 24–26, 76, 110, 111, 113, 114, 130, 134, 138, 141, 174, 180

SPE Synergistic Processing Element. 14, 15

spinor A vector of four complex three-vectors. 4, 8–10, 16, 17, 77, 78, 80, 82–84, 88, 89, 91, 99

SPMD Single Process Multiple Data. 12

sprofile A profiler for GPU applications included in AMD's Accelerated Processing Platform (APP) software development kit (SDK). 100

SPU Synergistic Processing Unit. 14, 15

stride The offset, usually in bytes, between two areas of memory. This typically occurs when storing multi-dimensional data or using a SoA layout. In the former it describes the offset between two rows, in the latter the offset between two arrays. The performance implications of different strides are discussed in Subsection 3.1.5. 47, 50–52, 178

SU(3) A group of 3×3 matrices with a determinant of 1. 4–6, 10, 82, 89, 91, 93, 99, 172

Tahiti A GPU chip based on the GCN architecture. It is used in the AMD Radeon HD 7970 and the AMD FirePro S10000. 20, 26, 53, 56, 102

TAO A programming language. 12

TCA Total Cost of Acquisition. iii, 118–120, 133

Tesla A GPU architecture by NVIDIA. Not to be confused with the product series of the same name. 20, 25, 26

Texas Instruments TMS320C31-50 A processor by Texas Instruments capable of SP floating-point arithmetic at a peak speed of 50 MFLOPS. 12

texture The representation of an image in GPU memory. 60

thread A sequence of instructions with some state. 42

titanic A CPU development system. For details see Table C.5. 117, 121, 147

tmlqcd An LQCD program suite for calculation with twisted-mass fermions. It was used as a reference code for CL^{2}QCD. 11, 115, 117, 120, 121, 130, 189

TOP500 A list of the 500 fastest supercomputers ranked according to the LINPACK benchmark. It is updated twice a year. `http://top500.org/`. 1, 2, 14, 15, 168

Toshiba One of the companies involved in the creation of the Cell Broadband Engine. 165

TPC Time Projection Chamber. 57

upper An index in a specific direction that has a higher numeric value. Given periodic boundary conditions this might also be a smaller numeric value, for which decrementing by the given offset gets wrapped around by the modulo arithmetic. 87

VLIW Very Long Instruction Word. 26, 165, 166

VLSI Very-Large-Scale Integration. 11

Voodoo Graphics PCI A GPU by 3dfx Interactive. The first successful consumer-GPU that provided hardware-accelerated 3D graphics. 24, 163

Wilson fermions A discretization variant of LQCD. iii, 9, 133

Windows An OS by Microsoft. 24, 31, 33, 166

work group A set of threads that will be scheduled to the same CU and can communicate via local memory. 20, 169

Xilinx Virtex-5 LX110T An FPGA by Xilinx. 15

List of Figures

List of Tables

List of Listings

Zusammenfassung

Große internationale Kooperationsprojekte am Large Hadron Collider (LHC) am CERN, sowie zukünftig an der Facility for Antiproton and Ion Research (FAIR) an der GSI Helmholtzzentrum für Schwerionenforschung GmbH (GSI), beschäftigen sich mit dem Verständnis der Quantenchromodynamik (QCD), der Wechselwirkung zwischen Gluonen und Quarks, den Bausteinen aller hadronischer Materie. Störungstheoretischen Ansätzen ist die QCD allerdings nur im Bereich hoher Energien zugänglich. *Ab initio* lässt sie sich für niedrigere Energien nur durch die Diskretisierung auf ein euklidisches Gitter in Raum und Zeit rechnen. Dieser Ansatz ist als Gitter-QCD bekannt.

Gitter-QCD-Rechnungen werden aufgrund ihres hohen Rechenbedarfs auf den größten wissenschaftlichen Clustern durchgeführt und haben wiederholt deren Architektur beeinflusst. Mit dem Aufkommen der Nutzung von Grafikprozessoren für nichtgrafische Berechnungen (GPGPU) wurden diese auch für die Berechnung der Gitter-QCD interessant. Anders als traditionell für die Gitter-QCD genutzte Rechner sind diese ein Massenmarktprodukt, was Vorteile in Hinblick auf Preis und Weiterentwicklung verspricht.

Im Rahmen dieser Dissertation wurde CL^{2}QCD entwickelt, eine auf OpenCL basierende Anwendung, welche Gitter-QCD-Rechnungen sowohl auf Grafikprozessoren als auch auf traditionellen Prozessoren ermöglicht. Anders als andere GPGPU-Anwendungen für Gitter-QCD ist CL^{2}QCD nicht auf Grafikprozessoren eines einzelnen Herstellers beschränkt. CL^{2}QCD bietet nicht nur eine hohe Rechenleistung, sondern übertrifft Systeme ohne Grafikprozessor auch in der Energieeffizienz. Außerdem ermöglicht CL^{2}QCD die Nutzung günstiger Hardware und skaliert auf mehrere Grafikprozessoren.

Um die Rechenleistung von CL^{2}QCD zu ermöglichen, wurden die Eigenschaften der Grafikprozessoren intensiv studiert und mehrere Optimierungstechniken für Anwendungen, welche durch die erreichbare Datenrate beim Speicherzugriff limitiert sind, evaluiert und entwickelt.

Einführung

QCD ist eine auf $SU(N_c)$ basierende Eichtheorie mit fermionischen Feldern und Eichfeldern. In der Natur hat N_c, die Anzahl der Farben, den Wert drei. Gitter-QCD ermöglicht auf dieser Theorie basierende Rechnungen dadurch, dass die Zustandssumme

$$Z = \int \mathcal{D}A_\mu \, \mathcal{D}\psi \, \mathcal{D}\overline{\psi} \, e^{-S_{\mathrm{QCD}}} \tag{1}$$

mit der Wirkung

$$S_{\mathrm{QCD}} = \int d^4x \left(\frac{1}{4} F_{\mu\nu} F^{\mu\nu} - \overline{\psi} M(A_\mu) \psi \right) \tag{2}$$

auf einem vierdimensionalen Raum-Zeit-Gitter diskretisiert wird. Das diskretisierte Gitter besteht aus dem Spinorfeld auf den Knotenpunkten des Gitters und den Linkvariablen auf den Kanten. Letztere werden anstelle des Eichfeldes verwendet und gehen durch ein Integral entlang der Kante aus diesem hervor.

Die diskretisierte Zustandssumme enthält immer noch so viele Integrationsvariablen wie das Gitter Linkvariablen enthält. Deshalb werden Monte-Carlo-Verfahren verwendet, um das Integral durch Abtastung des Phasenraumes auszuwerten. Das wichtigste Verfahren hierfür ist das *hybride Monte-Carlo-Verfahren (HMC)*, welches auf Basis eine Markov-Prozesses in einem fiktiven molekulardynamischen System eine Kette von Gitterkonfigurationen erstellt. Hierbei entspricht die Wahrscheinlichkeit für eine Konfiguration, in der Kette enthalten zu sein, ihrem Gewicht in der Zustandssumme.

Der fermionische Teil der Zustandssumme lässt sich ausintegrieren, was allerdings zur Berechnung der Determinante der Fermionmatrix führt. Um diese nicht berechnen zu müssen, werden zwei Pseudofermionfelder eingeführt. Dies führt zu einer effektiven Wirkung:

$$S_{\mathrm{eff}}(U, \phi) = S_{\mathrm{eich}}(U) + \phi_R^\dagger D^{-1}(U) \phi_I. \tag{3}$$

Die Berechnung der inversen Fermionmatrix wird aufgrund ihrer hohen Kosten durch die Lösung des entsprechenden Gleichungssystems

$$\psi = D^{-1} \phi \Rightarrow D\psi = \phi \tag{4}$$

ersetzt. Außerdem lässt sich die Fermionmatrix in eine Diagonalmatrix und $\not{D}$, eine Matrix welche ausschließlich Nebendiagonalen enthält, aufteilen. Die Anwendung von $\not{D}$ auf das Pseudofermionfeld dominiert die Rechenzeit in der Gitter-QCD.

Um den Anforderungen an Rechenleistung und Speichersystem gerecht zu werden, wurden immer wieder neue Technologien adaptiert und spezielle Systeme für die Gitter-QCD entwickelt, auch wenn dies häufig die Portierung tausender Zeilen Code bedeutete [61]. Über die Jahre gab es hierbei eine Verschiebung von vollen Eigenentwicklungen zur Nutzung von Massenmarktprodukten. Frühe Maschinen, wie die APE [5], wurden explizit nur für die Gitter-QCD gebaut, und Speicher und Rechenleistung explizit auf diese Anwendung abgestimmt. Die Blue Gene [8] hingegen ist ein kommerzielles Produkt. Allerdings basiert sie architektonisch auf QCDOC [46], welche explizit für die Gitter-QCD entwickelt wurde. All diesen Architekturen ist gemeinsam, dass die Kommunikation zwischen benachbarten Prozessoren besonders effizient ist.

Seit dem Aufkommen von GPGPU werden auch Grafikprozessoren für die Gitter-QCD verwendet. Allerdings basieren die meisten Anwendungen auf NVIDIA CUDA und sind damit auf Grafikprozessoren dieses Herstellers beschränkt, obwohl herstellerübergreifende Standards existieren. Selbst die nicht auf NVIDIA CUDA basierenden Lösungen sind größtenteils nur auf Prozessoren dieses Herstellers getestet.

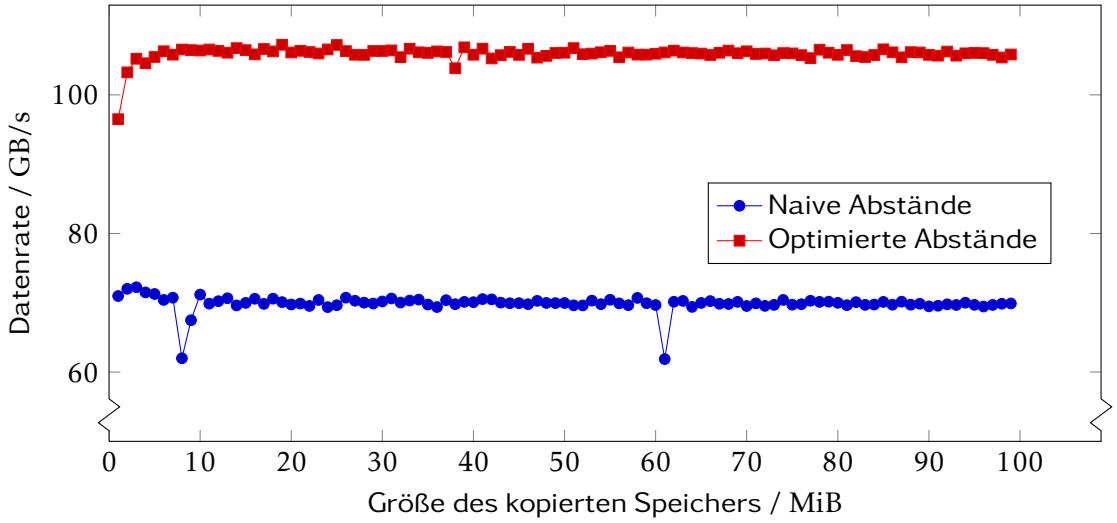

Abbildung 1.: Auf einer AMD Radeon HD 5870 beim Kopieren erreichte Datenrate für einen Datentyp von 32 B der per SoA auf zwei Segmente mit Typen von 16 B verteilt gespeichert wird.

Optimierungstechniken

Die Rechenleistung von Gitter-QCD-Anwendungen wird vor allem durch die beim Zugriff auf den Speicher verfügbare Datenrate bestimmt. Außerdem unterscheidet sich die Charakteristik von Speicherzugriffen eines Grafikprozessors deutlich von denen eines klassischen Prozessors. So optimiert der Cache eines Grafikprozessors vor allem gleichzeitige Zugriffe unterschiedlicher Threads, während er auf einem klassischen Prozessor aufeinander folgende Zugriffe des gleichen Threads optimiert. Deshalb habe ich Optimierungen untersucht, welche den Zugriff auf den Speicher betreffen.

Ein wichtiger Punkt bei der optimalen Nutzung der verfügbaren Datenrate ist der verwendete Datentyp. Die in der Gitter-QCD verwendeten Datentypen sind über 100 B groß. Ein Benchmark, bei dem nur Daten kopiert werden, zeigt aber, dass auf einer AMD Radeon HD 7970 bereits Typen ab 64 B lediglich 91 GB/s nutzen lassen. Einfachere Typen, wie zum Beispiel solche mit nur 8 B, erlauben es aber, über 200 GB/s zu nutzen.

Möchte man größere Datentypen auf hochperformante Datentypen abbilden, indem man das sogenannte SoA-Pattern nutzt, zeigt sich, dass die relative Positionierung der Daten im Speicher sich stark auf die erreichbaren Datenraten auswirkt. Durch einen im Rahmen dieser Arbeit entwickelten Algorithmus, der den Abstand zwischen den einzelnen Datensegmenten optimiert, lässt sich, wie in Abbildung 1 zu sehen ist, für Daten beliebiger Größe die bestmögliche Datenrate erzielen.

Die Untersuchung des Einflusses von ECC auf die erreichbaren Bandbreiten zeigt auf einer AMD FirePro S10000 einen Verlust von etwa der Hälfte der erreichbaren Da-

tenrate. Datenratensensitive Anwendungen sollten sich deshalb um andere Formen der Fehlererkennung bemühen.

Da auf einem Grafikprozessor die Anzahl der einem Thread zur Verfügung stehenden Register nicht fix ist, sondern von der Anzahl der gleichzeitig ausgeführten Threads abhängt, wurde auch untersucht, wie sich deren Nutzung auf die erreichbare Datenrate auswirkt. Es zeigt sich, dass bereits eine geringe Anzahl an Threads die maximale Datenrate erreichen kann, so dass in solchen Anwendungen mit der maximalen Anzahl an Registern gearbeitet werden kann.

Ein wichtiger Unterschied zwischen Grafikprozessoren und klassischen Gitter-QCD-Maschinen besteht in der Kommunikation zwischen benachbarten Prozessoren. Gitter-QCD-Maschinen verfügen für gewöhnlich über ein Torus-Netzwerk, in dem benachbarte Prozessoren direkt miteinander kommunizieren können. Häufig ist dies sogar von Cache zu Cache möglich. Grafikprozessoren sind hingegen auf die Mithilfe des Hauptprozessor angewiesen. Die Untersuchung verschiedener Kommunikationsmuster zeigt, dass es im Allgemeinen am effektivsten ist, die Details der Kommunikation OpenCL zu überlassen. Für kleine Datenmengen kann allerdings die Latenz verringert werden, indem der Hauptprozessors direkt auf den Speicher des Grafikprozessors zugreift. Hängen alle Grafikprozessoren im gleichen PCIe-Baum, lässt sich durch die Nutzung von DirectGMA eine noch schnellere Kommunikation erreichen, welche dann auch nicht mehr vom Hauptprozessor abhängt.

Für diese Analysen wurden mehrere Anwendungen erstellt, welche die Vermessung der Eigenschaften sowie die Analyse von GPU-Anwendungen erlauben [68, 76, 80].

CL^{2}QCD

Die im Rahmen dieser Arbeit entwickelte Anwendung nutzt die untersuchten Optimierungen, um Grafikkarten bei der Berechnung von Gitter-QCD optimal auszunutzen. Die Anwendung ist modular aufgebaut, so dass die Domänenlogik und die Optimierung auf die benutzten Prozessoren entkoppelt sind. So konnten sich meine Coautoren von „LatticeQCD using OpenCL“ [19] und „Lattice QCD based on OpenCL“ [20] auf die Physik konzentrieren. Ein Prototyp des Langevin-Algorithmus, einer Alternative zum HMC, wurde innerhalb eines Tages entwickelt und profitierte bereits von allen Optimierungen.

Zusätzlich zu den im Allgemeinen beschriebenen Optimierungen musste unter anderem die Formulierung des $\not{D}$ an den Compiler angepasst werden, so dass dieser nicht unnötig ressourcenhungrigen Code generierte. Um die in Frankfurt durchgeführten Studien zu ermöglichen, wurde außerdem besonderes Augenmerk auf die erreichte Rechenleistung bei kleinen Gittern gelegt. Zusätzlich wurde eine, für die Domänenlogik transparente, Möglichkeit geschaffen, die Ausführung auf mehrere Grafikprozessoren zu verteilen. Dies erlaubt Rechnungen, die nicht in den Speicher eines einzelnen Prozessors passen.

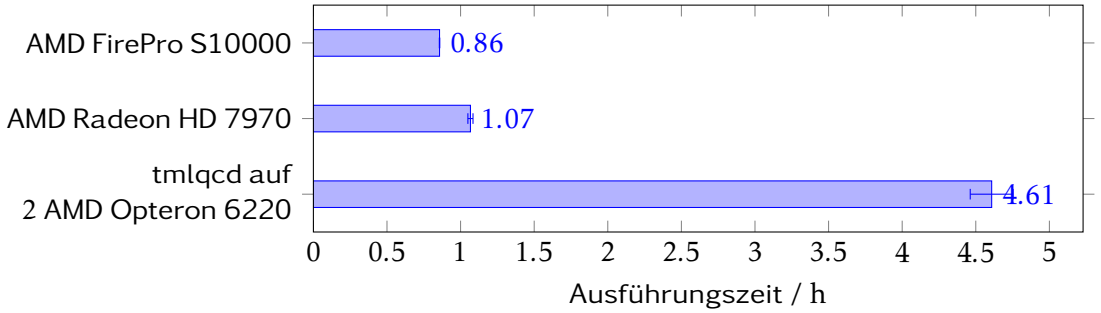

Abbildung 2.: Laufzeit für einen Schritt des HMC.

Ergebnisse

Die Implementierung des $\not{D}$ in CL^{2}QCD erreicht auf einer AMD Radeon HD 7970 bis zu 125 GFLOPS mit doppelter Genauigkeit. Dies ist die höchste veröffentlichte Rechenleistung für einen $\not{D}$ mit doppelter Genauigkeit. Auf aktuellen Grafikprozessoren übertrifft die Rechenleistung im gesamten HMC die eines Systems mit zwei Serverprozessoren, auf welchen die Referenzanwendung tmlqcd läuft, um das vierfache. Dies ist in Abbildung 2 dargestellt.

Die auf die Rechenleistung normalisierten Anschaffungskosten lassen sich bei der Nutzung von CL^{2}QCD bis auf circa \$0.012/MFLOPS senken. Andere GPU-basierte Gitter-QCD-Anwendungen liegen im gleichen Bereich. Auf dem Großrechner SANAM, welcher weniger auf Gitter-QCD optimiert ist, liegen die Kosten bei \$0.033/MFLOPS, und damit in der gleichen Größenordnung wie bei einer Blue Gene.

Die im Vergleich zu einem System ohne Grafikprozessoren um einen Faktor vier höhere Rechenleistung überträgt sich eins zu eins auf eine bessere Energieeffizienz. Durch die Nutzung mehrerer Grafikprozessoren in einem einzelnen System lässt sich dieser Vorteil sogar bis zu einem Faktor von 5.25 steigern. Hierbei ist allerdings auf eine hinreichende Kühlung der Grafikprozessoren zu achten.

Bei der Nutzung mehrerer Grafikprozessoren skaliert CL^{2}QCD für hinreichend große Gitter mit einer Effizienz von 85 % linear. Im $\not{D}$ erreichen die vier Prozessoren auf zwei AMD FirePro S10000 bis zu 400 GFLOPS. Im iterativen Gleichungssystemlöser werden 250 GFLOPS erreicht.

Ausblick

CL^{2}QCD ermöglicht kostengünstiges, energieeffizientes und schnelles Rechnen von Gitter-QCD. Aktuell wird bereits daran gearbeitet, weitere Varianten der Gitter-QCD zu unterstützen. Nächste Schritte wären die Einbindung weiterer Optimierungstechniken wie REC12, die weitere Verbesserung der Parallelisierung über mehrere Grafikprozessoren und die Optimierung auf weitere von OpenCL unterstützte Prozessoren. Insbesondere die Nutzung kombinierter Haupt- und Grafikprozessoren ist interessant.