Implementation and analysis of lattice QCD algorithms on graphics processing units

Diplomarbeit im Fachbereich Physik — **Bjoern Walk**

August 2009

Institut für Kernphysik

Johannes-Gutenberg-Universität Mainz

Johann-Joachim-Becher-Weg 45

55128 Mainz, Germany

Acknowledgements

This diploma thesis would not have been possible without the support of many people. First of all, I would like to thank my advisors, Prof. Hartmut Wittig and Prof. Elmar Schömer, for their guidance during my research. Both were always available and attentive and have patiently answered any open question.

Of course my whole study would not have been possible without the unconditional support of my parents who always stood behind me in my decisions and never doubted me. In particular, I would like to thank my father who encouraged me to scientific thinking from since childhood.

I thank my colleges at the *Institut für Kernphysik*, especially the guys in my office, Bastian Brandt, Erik Endreß, Bastian Knippschild and Michele della Morte, for their open ear and patiently answering even stupid questions. I also thank Egor Dranischnikow for his time and for his helping hand in the beginning of my research. I am especially thankful for the work of prove-reading this thesis by Nicole Rüdinger, Andreas Neiser and Fabian Nillius.

I am especially thankful for my fellow students, in particular Marius Hilt, Fabian Nillius and Robert Bormuth. On top of being great friends, they have been a constant source of encouragement and creative ideas.

Finally, I would like to thank all the guys who made the time at the university always a pleasure no matter if it was simple conversation over lunch or one of the uncountable (500+) games of *Doppelkopf*. It always helped me to maintain sanity during stressful times.

Contents

Chapter 1

Introduction

The theory of Quantum Chromodynamics (QCD) is widely regarded as the correct theory to describe one of the four interactions in nature, the *strong force*. It describes the fundamental interaction of elementary particles named quarks and gluons which are the basic building blocks of so-called hadrons such as the proton or the neutron.

Physicists are interested in the question, if the properties of the hadrons can be rigorously derived from the basic theory which can, in principle, be formulated as a one-line equation. However, due to the complex mathematical structure, analytical solutions to QCD have not yet been found.

The key reason for this absence is, that the most successful tool in physics, *perturbation theory*, is not as easy to apply to QCD as it is to other theories such as Quantum Electrodynamics (QED). The electromagnetic interaction can be considered as a small perturbation to the system and the theory of QED can be expanded in a small parameter, usually the coupling constant. For the coupling constant of QED it is $\alpha \approx 1/137$ and therefore, the series expansion converges quickly. This approach gave most impressive results confirmed by experiments such as the anomalous magnetic dipole moment g of the electron which is known up to 10 significant digits. This makes this value one of the most exact values known in physics and gives confirmation of the theory at very high precision.

Coupling constants usually show a dependence of the energy scale. That is, the coupling constant of QCD, α_s, is a function of the momentum transfer q^2 of the examined process. This energy dependence is given by the *beta-function* which for QCD was first found by Wilczek, Politzer and Gross. It reads

$$\alpha_s(q^2) \approx \frac{1}{\beta_0 \log(k^2/\Lambda^2)}$$

where β_0 is a constant and Λ is the QCD scale.

At both ends of the energy scale, QCD behave quite differently. In the limit of high energy, for which the coupling constant becomes small, quarks can move freely within the nuclei. That fact—called *asymptotic freedom*—was also discovered by Wilczek, Politzer and Gross[GW73, Pol73] and was rewarded with the Nobel Prize in 2004. On the other hand, in the limit of low-energy reactions, the coupling constant becomes large as q^2 decreases and perturbation theory is no longer applicable. This phenomenon is called *color confinement* and is the reason for the absence of free quarks in nature.

To study the low-energy regime of QCD and confinement in particular, in 1974 Kenneth Wilson introduced *Lattice QCD* [Wil74]. In this formulation, Wilson was able to treat QCD in a non-perturbative way and to investigate quark confinement analytically. However, this formulation also formed also the foundation to analyze QCD using numerical methods.

Lattice simulations demand extreme computing resources. To simulate a physical system with reasonable dimensions, the lattice volume in physical units need to be at least in the order of the simulated particles which is around 1–2 fm. On the other hand, the lattice needs to be fine enough to suppress discretization errors, this is usually the case for a physical lattice spacing of around 0.1 fm. For the extrapolation to the physical point with $a = 0$, it is necessary to simulate at even finer lattice spacings. Current lattice volumes have volume extents of $T = 128$ and $L = 64$ and more, resulting in tens or hundreds of millions of lattice sites.

Systems of that size can not be handled by single processors computers efficiently. On computer clusters, the combination of a multitude of single computing nodes with a fast

network interface, such lattices will be divided in smaller sub-lattices and each sub-lattice can be computed in parallel. With recent algorithms, the communication necessary between the sub-lattices can be kept at a minimum and the efficiency of this approach highly depends on the adjustment of the network interface and the performance of the single processors.

Another development gives the promising perspective to a whole new approach for lattice computations. In the last ten years, *graphic processing units* (GPU) turned from simple output devices which present information on the screen to the user to programmable, highly flexible, massively parallel processing units with a very reasonable prices tag. The hunt for more and more realistic 3D scenes in professional like CAD ("Computer Aided Design") software or video games and the ability to efficiently accelerate the basic calculations needed for the rendering process pushed development of hardware solutions further and further.

Nowadays, the peak performance of a single GPU has broken the magic barrier of one TFlops/s—that is one trillion or 10^{12} floating-point operations per second—and is available for the mass-market. The question is raised, if this vast computing power in almost every modern consumer personal computer can be exploited for lattice calculations.

This diploma thesis is organized as follows. In Chapter 2, a short introduction of lattice QCD. The Wilson-Dirac operator and the Neuberger operator for chiral fermions is described and the necessity of an efficient implementation for those operators is given with a short overview of numerical simulations. In Chapter 3, NVIDIA's framework for general purpose computations on the GPU, CUDA ("Compute Unified Device Architecture"), is presented with some technical description of the used hardware. Chapter 4 focuses on the implementation of the Wilson-Dirac operator and the Neuberger operator and the results obtained are presented in Chapter 5.

Chapter 2

Theoretical background

The generic concepts of the lattice formulation can be summarized by the following statement[Wit08]:

Lattice QCD is the non-perturbative approach to the gauge theory of the strong interaction through regularized, Euclidean functional integrals. The regularization is based on a discretization of the QCD action which preserves gauge invariance at all stages.

The starting point for lattice QCD is the Euclidean functional integral itself, thus avoiding any particular reference to perturbation theory in the first place. Therefore, lattice QCD can be regarded as an *ab initio* method to QCD which can compute observables from first principles. Furthermore, the Euclidean formulation reveals the close relation between Quantum Field Theory and Statistical Mechanics. The Euclidean functional integral is equivalent to the partition function of the corresponding statistical system and by that, it is possible to take advantage of the whole toolkit of condensed matter physics, in particular Monte Carlo simulations.

In the following a short overview is given about how QCD can be formulated on the lattice. Different approaches to the simulation of fermions on the lattice are outlined and Wilson fermions and Neuberger fermions for chiral fermions on the lattice are introduced. In the end some essential techniques for numerical simulations are described like the inversion of the fermion matrix and a deeper investigation of the Neuberger operator.

2.1 Lattice QCD

The Euclidean action of QCD S_{QCD} in the continuum can be divided into the action for the gauge fields S_{G} and the fermionic part S_{F}. In a common formulation the action reads

$$S_{\mathrm{G}} = \int \mathrm{d}^4x \left\{ -\frac{1}{2g_0} \mathrm{Tr}\left(F_{\mu\nu}F_{\mu\nu}\right) \right\}$$

where g_0 denotes the gauge coupling and

$$S_{\mathrm{F}} = \int \mathrm{d}^4x \sum_{f=u,d,s,\ldots} \overline{\psi}_f \left(\gamma_\mu D_\mu + m_f\right) \psi_f .$$

The covariant derivative is defined through $D_\mu = \partial_\mu + A_\mu$. The field strength tensor then reads

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu + \left[A_\mu, A_\nu\right], \qquad A_\mu^\dagger = -A_\mu .$$

However, for the rest of this work only the fermionic part of the action will be discussed and the gauge fields will be regarded as background fields. Details of the dynamics of the gauge fields are not crucial for the rest of this work.

For the discretization of the theory the hyper-cubic lattice Λ_E is introduced as the set of discrete space-time points, i.e.

$$\Lambda_E = \left\{ x \in \mathbb{R}^4 \mid x^0/a = 1, \ldots, T;\ x^i/a = 1, \ldots, L,\ i = 1,2,3 \right\}.$$

Any space-time point is an integer multiple of the lattice spacing a and the total number of lattice sites is $T \times L^3$. The fermionic degrees of freedom will sit on the lattice sites itself, later it will be convenient to identify the corresponding links between the sites with the gauge fields.

To see the regularization properties of the lattice the dual lattice Λ_E^* which is related to the Euclidean lattice Λ_E via a Fourier transformation, is taken into account. It is defined by

$$\Lambda_E^* = \left\{ p \in \mathbb{R}^4 \mid p^0 = \frac{2\pi}{aT} n^0;\ p_i = \frac{2\pi}{aL} n^i,\ i = 1,2,3 \right\}$$

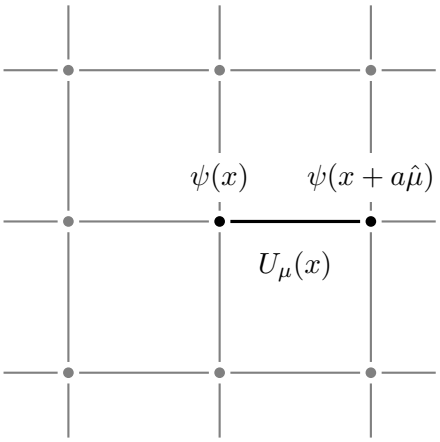

Figure 2.1: A two-dimensional extract of a lattice is shown to demonstrate various important elements. Filled dots are the lattice sites and are identified with the fermionic degrees of freedom, the links between lattice sites are identified with the gauge variables.

where $n^0 = -T/2, -T/2+1, \ldots, T/2-1$ and $n^i = -L/2, L/2+1, \ldots, L/2-1$. This has two implications: the momenta p_0 and p_i are quantized in units of $2\pi/T$ and $2\pi/L$, respectively, and a momentum cutoff has been introduced because the momenta are limited by

$$-\frac{\pi}{a} < p_\mu \leq \frac{\pi}{a}.$$

To find a discretized version of the covariant derivative D_μ required in the presence of the gauge fields, first of all we consider the corresponding lattice derivatives d_μ for the partial derivative ∂_μ in the continuum. Usually the first order difference quotient will be used and can be written on the lattice as

$$d_\mu \phi(x) := \frac{1}{a}\big(\phi(x + a\hat{\mu}) - \phi(x)\big) \qquad \text{“forward” derivative,}$$
$$d^*_\mu \phi(x) := \frac{1}{a}\big(\phi(x) - \phi(x - a\hat{\mu})\big) \qquad \text{“backward” derivative.}$$

Here and for the rest of this work $\hat{\mu}$ denotes a unit vector in direction of μ. In the limit $a \to 0$ it is easy to show that both formulations give the correct partial derivative ∂_μ.

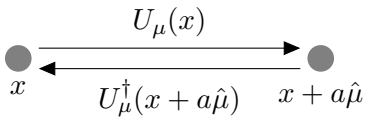

Figure 2.2: The link variable $U_\mu(x)$ sits on the lattice site with index x and points into direction of lattice site $x + a\hat{\mu}$. The same link variable can be obtained from the lattice site with index $x + a\hat{\mu}$ with the inverse direction.

Before it is possible to write down a discretized version of the fermionic action S_F, the notion of a lattice gauge field in the non-Abelian case has to be discussed. The familiar transformation law for the gauge fields in the continuum,

$$A_\mu(x) \to g(x)A_\mu(x)g(x)^{-1} + g(x)\partial_\mu g(x)^{-1}, \qquad g(x) \in \mathrm{SU}(3),$$

no longer holds exactly when ∂_μ is replaced by its discrete version d_μ defined above.

In the presence of a background gauge field a quark moving from point y to x picks up a non-Abelian phase factor

$$U(x, y) = \mathrm{P.O.} \exp\left\{-\int_y^x \mathrm{d}z_\mu A_\mu(z)\right\}$$

where "P.O." denotes path ordering, as a consequence of the non-Abelian nature of the gauge field. As the gauge potential A_μ is an element of the Lie algebra of SU(3), the *parallel transporter* $U(x, y)$ is an element of the gauge group itself. On the lattice the *link variables* are defined as the parallel transporter between neighboring lattice sites x and $x + a\hat{\mu}$

$$U(x, x + a\hat{\mu}) \equiv U_\mu(x) \quad \text{and} \quad U(x + a\hat{\mu}, x) = U(x, x + a\hat{\mu})^{-1}.$$

Now it is possible to obtain a consistent and manifestly gauge invariant discretization of QCD, if the gauge degrees of freedom are identified with the link variables $U_\mu(x)$ which transform like

$$U_\mu(x) \to g(x)U_\mu(x)g(x + a\hat{\mu})^{-1}, \quad g(x), g(x + a\hat{\mu}) \in \mathrm{SU}(3).$$

2.2 Lattice actions

Now the focus will be on the problem to discretize of the fermionic action S_{F}. The quark and antiquark fields $\psi(x)$ and $\overline{\psi}(x)$ are associated with the lattice sites and transform under the gauge group as

$$\psi(x) \to g(x)\psi(x), \qquad \overline{\psi}(x) \to \overline{\psi}(x)g(x)^{-1}.$$

With the transformation properties of the link variables it is straightforward to write down a discretized version of the covariant derivative

$$\begin{aligned}\nabla_\mu \psi(x) &\equiv \frac{1}{a}\left(U_\mu(x)\psi(x+a\hat{\mu}) - \psi(x)\right), \\ \nabla^*_\mu \psi(x) &\equiv \frac{1}{a}\left(\psi(x) - U_\mu(x-a\hat{\mu})^{-1}\psi(x-a\hat{\mu})\right)\end{aligned}$$

where ∇_μ and ∇^*_μ denote the "forward" and "backward" derivatives, respectively, as in the case of the partial derivative. For the construction of a discretized action one should note that there is certain degree of freedom because the lattice action is not unique. For the discretized expression it is only required that it reproduces the continuum result as the lattice spacing a is taken to zero. It is always possible to add a term to the discretized action which formally vanishes as $a \to 0$. As shown later, it is possible to exploit this fact to tune the lattice formulation, e.g. for a better rate of convergence.

For the massless Dirac operator D there are four basic conditions conditions the discretized version should satisfy.

(a) The lattice Dirac operator D is local. In particular, the absence of long-ranged interaction is a basic property of any quantum field theory which describes elementary particles.

(b) The Fourier transformation of D is of the form $\widetilde{D}(p) = i\gamma_\mu p_\mu + \mathcal{O}(ap^2)$. This ensures the correct continuum behaviour of the quark-gluon interaction.

(c) The Fourier transformation $\widetilde{D}(p)$ is invertible for $p \neq 0$. This property makes sure that the lattice Dirac operator gives the correct fermion spectrum. Fermion masses are associated with the poles of $\widetilde{D}(p)$ and in the continuum, poles only occur at vanishing four-momentum.

(d) The lattice Dirac operator satisfies $\gamma_5 D + D\gamma_5 = 0$. This relation expresses invariance under chiral transformation in the massless case.

The most simple discretization of the massless lattice Dirac operator can be written as

$$D_{\text{disc}} = \frac{1}{2}\gamma_\mu \left(\nabla_\mu + \nabla_\mu^*\right).$$

Often this definition is referred as a "naïve" discretization. This term can understood by the Fourier transform of D_{disc} which is given by

$$\widetilde{D}_{\text{disc}}(x) = i\gamma_\mu \frac{1}{a}\sin(ap_\mu) = i\gamma_\mu p_\mu + \mathcal{O}(a^2).$$

While the Taylor expansion shows that the condition (b) above is satisfied, the occurrence of $\sin(ap_\mu)$ implies that $\widetilde{D}_{\text{disc}}$ vanishes not only at $p_\mu = 0$, but also at $p_\mu = \pi/a$ for $\mu = 0, \ldots, 3$, thus violating condition (c). The massless propagator $\widetilde{D}_{\text{disc}}(p)$, therefore, has $2^4 = 16$ poles giving a 16-fold degeneracy of the fermion spectrum. This problem is called *fermion doubling* and it is closely linked to the issue of chiral symmetry on the lattice. In [NN81] it was shown that the conditions (a)–(d) can not be fulfilled simultaneously. This fact is known as the Nielsen-Ninomiya No-Go theorem. Since locality and condition (b) are too important to give up this implies that either (c) or (d) must be violated.

There are a few methods to address the discrepancy between the correct fermion spectrum and the preservation of chiral symmetry and a short introduction of the most popular methods will be given in the next paragraphs.

2.3 Wilson fermions

As the indicated by the name, Kenneth Wilson[Wil74] introduced a first solution to lift the degeneracy completely. Unfortunately, the price to pay for this accomplishment was to break chiral symmetry explicitly. However, for most applications in lattice QCD, explicit breaking of chiral symmetry is not a serious obstacle.

As seen above, it is permitted to add a term to D_{disc} that vanishes in the continuum limit $a \to 0$. This degree of freedom makes it possible to find a term that pushes the masses of the unwanted doubler states to the cutoff scale at any non-zero value of the lattice spacing. The Wilson-Dirac operator D_{W} can be written as

$$D_{\text{W}} = \frac{1}{2}\gamma_\mu \left(\nabla_\mu + \nabla_\mu^*\right) + ar\nabla_\mu\nabla_\mu^*$$

where r is a free parameter called Wilson parameter which is usually set to one. For the spectrum of the Wilson-Dirac operator it is necessary to calculate the Fourier transform of D_{W} which is given by

$$\widetilde{D}_{\text{W}}(p) = i\gamma_\mu \frac{1}{a}\sin(ap_\mu) + \frac{2r}{a}\sin^2\Big(\frac{ap_\mu}{2}\Big).$$

The poles at $p_\mu = \pi/a$ are shifted by a term proportional to r/a which is of order of the cutoff for $r = \mathcal{O}(1)$. It can be shown in numerical simulations that the degeneracy is indeed lifted completely. However, Wilson fermions have a number of unwanted features: as a result of the counterterm proportional to ar, the Wilson fermion action differs from the classical action in the continuum by terms of order a. By contrast, the leading discretization effects of the naïve action are only $\mathcal{O}(a^2)$, therefore, the Wilson fermion formulation will have a reduced rate of convergence towards the continuum limit. Furthermore, the Wilson term results in an explicit breaking of chiral symmetry, since the massless theory is no longer invariant under global axial rotations

$$\psi(x) \to \mathrm{e}^{i\alpha\gamma_5}\psi(x), \qquad \overline{\psi}(x) \to \overline{\psi}(x)\,\mathrm{e}^{i\alpha\gamma_5}.$$

It is straightforward to see that this is not a symmetry of the Wilson action and that D_{W} does not anti-commute with γ_5.

For the rate of convergence to the continuum limit, a special program has been employed, the "$\mathcal{O}(a)$ improvement" by Symanzik[Sym83] which is a systematic approach to remove lattice artifacts order by order in the lattice spacing. In a nutshell the improvement amounts to extending the renormalization procedure of a field theory to the level of irrelevant operators, i.e. operators that formally vanish as $a \to 0$. In this sense suitable counterterms are added which for any non-zero value of a produce a cancellation of the cutoff effects at a given order. How this is realized explicitly for the Wilson fermion action can be seen in [SW85]. However, the rate of convergence will not be discussed in this work and an $\mathcal{O}(a)$-improved Wilson-Dirac operator will not be used.

The massive Wilson-Dirac operator Q for a quark with a bare mass of m_0 is then given by $Q = D_{\mathrm{W}} + m_0$. The operator can be rewritten so that it is better suited for numerical analysis if the so-called *hopping parameter* κ is introduced. Then the Wilson fermion action can rewritten in terms of this parameter rather than m_0.

The action can be written as

$$\begin{aligned} S_{\mathrm{F}}^{\mathrm{W}} &\equiv \sum_{x\in\Lambda_E} \overline{\psi}(x) Q \psi(x) \\ &= \sum_{x\in\Lambda_E} \Big\{ \overline{\psi}(x)\psi(x) - \kappa \sum_{\mu=0}^{3} \big(\overline{\psi}(x)\,(1-\gamma_\mu)\,U_\mu(x)\psi(x+a\hat{\mu}) \\ &\qquad + \overline{\psi}(x-a\hat{\mu})\,(1+\gamma_\mu)\,U_\mu(x-a\hat{\mu})^{-1}\psi(x)\big) \Big\}. \end{aligned} \tag{2.1}$$

As mentioned already above, the Wilson parameter r is set to 1. The hopping parameter is related to the bare mass m_0 via $\kappa^{-1} = 2am_0 + 8$.

From Eq. (2.1) the Wilson-Dirac operator $Q \equiv D_{\mathrm{W}} + m_0$ can be read off and with the definition $U_{-\mu}(x) \equiv U_\mu^{-1}(x - a\hat{\mu})$ and $\gamma_{-\mu} \equiv -\gamma_\mu$ it is possible to write it in a most compact way

$$Q\psi(x) = \psi(x) - \kappa \sum_{\mu=\pm 0}^{\pm 3} U_\mu(x)(1-\gamma_\mu)\psi(x+a\hat{\mu}),$$

Matrix-form of the Wilson-Dirac operator Often the application of the Wilson-Dirac operator can be interpreted as a sparse matrix-vector multiplication such as $\psi_x = Q_{xy}\psi_y$. In that case, the spinor fields at each lattice site are just appended to one large column vector, i.e. $\psi_x \equiv \psi(x)$. The Wilson-Dirac operator is expressed by the *hopping matrix* M which is defined by

$$Q_{xy} = \delta_{xy} - \kappa M_{xy}$$

where the matrix elements M_{xy} of the hopping matrix are given by

$$M_{xy} = \sum_{\mu=\pm 0}^{\pm 3} U_\mu(x)\,(1+\gamma_\mu)\,\delta_{x+\hat{\mu},y}.$$

It must be pointed out that $U_\mu(x)$ and γ_μ are itself matrices which have color indices and Dirac indices, respectively. These indices have been and will be suppressed for the rest of this work. The matrices Q and M are square matrices with dimension $4 \cdot 3 \cdot T \times L^3$ in each direction, the spinor field ψ_x is a vector with length $4 \cdot 3 \cdot T \times L^3$.

2.4 Staggered fermions

Another method to solve the fermion doubling problem at least partially are the so-called *staggered fermions* [KS75, BSK76].

The spurious fermions states in the fermion propagator for the naïve discretization are distributed over different non-zero corners of the first Brillouin zone of the lattice, the unit box of the dual lattice Λ_E^*. In the free theory, fermion doubling is just an unwanted feature because one wants to describe the physical fermion species only. For interacting theories, however, the interesting question is, if the fermion doubling can be avoided in the continuum where those doublers, in principle, could be pair-produced.

In contrast to Wilson's approach, one way to deal with the problem is to interpret extra fermions as new physical "flavours". The basic idea is to assign only a single fermion field component to every lattice site which reduced the doublers from 16 to 4. This

is somewhat remarkable in the sense that this approach is a non-local description for fermions as the four fermion components for one single spinor sit on different lattice sites. However, it is not a direct violation of condition (a). One can imagine that the lattice is divided into blocks on four points and each block in the continuum limit goes to a single point. With this technique it is possible to reduce the number of degrees of freedom by a factor of four.

A simple distribution of the spinor components is not sufficient to define the fermion action, since the Dirac matrices mix different spinor components. The staggered fermion action is obtained after performing a spin diagonalization in spinor space which decouples the individual components. The actual calculation for this can be found in [MM94]. The staggered fermion action reads

$$S_{\mathrm{F}}^{\mathrm{stagg}} = \sum_{x\in\Lambda_E}\sum_{\alpha=0}^{3}\Big\{ m_0\overline{\chi}_a(x)\chi_a(x) + K\sum_{\mu=0}^{3}\eta_\mu(x)\left[\overline{\chi}_\alpha(x)\chi_\alpha(x+a\hat{\mu}) - \overline{\chi}_\alpha(x+a\hat{\mu})\chi_\alpha(x)\right]\Big\},$$

In that definition χ_α and $\overline{\chi}_\alpha$ denote one-component Grassmann variables. The spin diagonalization has replaced the Dirac matrices γ_μ by a real phase factor $\eta_\mu(x)$ which is given by

$$\eta_0(x) = 1, \qquad \eta_i(x) = (-1)^{n_0+\cdots+n_{i-1}}, \quad n_i = x_i/a.$$

The action is decomposed into four identical pieces and in order to occupy all 16 corners of a four-dimensional hypercube with one-component Grassmann variables one needs four Dirac spinors. By that, one still obtains a four-fold degeneracy of staggered fermions which are commonly referred to as “tastes”, in order to distinguish them from physical flavours. The formulation above can be re-expressed in terms of the spin-taste basis from which it can be seen that the taste symmetry is broken, except one axial generator. Therefore, the staggered fermion approach preserves a $U(1)\otimes U(1)$ subgroup of chiral symmetry.

2.5 Chiral fermions

One way to evade the Nielsen-Ninomiya theorem is to relax condition (d) as suggested in [GW82] in favour of

$$\gamma_5 D + D\gamma_5 = aD\gamma_5 D.$$

The so-called Ginsparg-Wilson relation implies an exact symmetry of the associated action[Lüs98] with infinitesimal variations proportional to

$$\delta\psi = \gamma_5 (1 - aD)\, \psi, \qquad \delta\overline{\psi} = \overline{\psi}\gamma_5.$$

Moreover, this symmetry reproduces the correct chiral anomaly in the flavour singlet case and therefore, all properties of the correct chiral behaviour in the lattice theory.

Domain wall fermions are one particular construction that satisfies the Gisparg-Wilson relation and preserve chiral symmetry in the continuum limit[Kap92, FS95]. The basic idea is, to introduce an extra fifth dimension and couple the fermions to a mass defect in that extra dimension. For clarifiction, let x, y denote the coordinates in the four-dimensional space-time and s, t the coordinates in the fifth dimension, which has a finite length of N_5. The gauge field can be chosen to be trivial in this fifth dimension and the Dirac operator has the general structure

$$D_{\mathrm{dwf}}(x, s; y, t) = D^{\parallel}(x, y)\delta_{st} + \delta(x - y)D^{\perp}_{st}$$

where $D^{\parallel}(x, y)$ denotes the Wilson-Dirac operator but with a negative mass term. The operator $D^{\perp}_{st}$ couples the fermions in the fifth dimension and contains the physical bare quark mass. It can be shown that for vanishing bare mass $m_0 = 0$ and in the limit $N_5 \to \infty$ there are no fermion doublers and, more importantly, chiral modes of opposite chirality are trapped in the four-dimensional domain walls $s = 1$ and $s = N_5$.

For numerical simulations of domain wall fermions N_5 has a finite value and the decoupling of the chiral modes is not exact. However, the suppression of chiral symmetry breaking effects can be expected to be exponential which has been confirmed in several simulations.

That way the domain wall formulation of QCD offers a method to realize chiral symmetry in the continuum at the expense of simulation a five-dimensional theory.

Another non-trivial solution to the Ginsparg-Wilson relation was found with the Neuberger operator[Neu98a, Neu98b] which is defined as

$$D_{\mathrm{N}} = \frac{1}{\overline{a}}\left(1 - \frac{A}{\sqrt{A^\dagger A}}\right), \qquad A = 1 + s - aD_{\mathrm{W}}, \quad \overline{a} = \frac{a}{1+s}$$

where D_{W} is the massless Wilson-Dirac operator and $|s| < 1$ is a tunable parameter. With $Q = -\gamma_5 A$ the Neuberger operator can be rewritten in the compact form

$$D_{\mathrm{N}} = \frac{1}{\overline{a}}\left(1 + \gamma_5 \,\mathrm{sign}(Q)\right).$$

Due to the occurrence of the square root in the definition and the sign function two problems arise. First, the application of D_{N} in a computer program is very expensive in term of computational costs. This is because the sign function of the matrix Q has to be implemented for instance using polynomial approximation[GHLW03].

The second issue is the question, if D_{N} satisfies condition (a), that means, if it is a local operator. Before it is possible to decide this, it is necessary to be aware of a good definition for locality. As mentioned above, locality is the absence of long-ranged interactions between fields in a quantum field theory. Then *strict locality* for which only fields in a local neighborhood of a given lattice site are coupled, can be defined. There exists a cutoff length ξ for which two fields $\phi(x)$ and $\phi(y)$ do not interact if $\xi < |x - y|$.

Strict locality is often a too rigorous requirement for a quantum field theory. A more relaxed condition is the following: if $D(x, y)$ denotes a generic lattice Dirac operator which couples fields at sites x and y, a sufficient condition for locality of D is the exponential suppression of non-local interactions, i.e.

$$\|D(x, y)\| \leq \mathrm{e}^{-\gamma|x-y|/a}$$

where $|x - y|$ is the distance between the sites and $\|\cdot\|$ is a suitable defined matrix norm.

It is shown in [HJL99] that the Neuberger operator D_{N} is local in the above sense, if the physical lattice spacing is not larger than a certain value.

2.6 Numerical simulations

The goal of numerical simulations in lattice quantum field theory is to estimate the expectation values of some functions $A[\varphi]$ of the field variables $[\varphi] \equiv \{\varphi_\alpha(x)\}$. Here $\varphi_\alpha(x)$ denotes a real field component with index α at lattice site x.

We can express this expectation value with the functional integrals as

$$\langle A \rangle = Z^{-1} \int [\mathrm{d}\varphi] A[\varphi] \, \mathrm{e}^{-S[\varphi]}, \qquad Z = \int [\mathrm{d}\varphi] \, \mathrm{e}^{-S[\varphi]}$$

where $S[\varphi]$ is the lattice action. Usually, the fermionic part in $S[\varphi]$ is integrated out with the usual techniques and the fermionic action is written as the so-called *quark determinant* via

$$\langle A \rangle = Z^{-1} \int [\mathrm{d}U] \widetilde{A}[U] \det(D_{\mathrm{lat}}) \, \mathrm{e}^{-S_{\mathrm{G}}[U]}$$

where the integration over the gauge fields $U \equiv \{U_\mu(x)\}$ remains. $\widetilde{A}$ denotes the representation of A in the effective theory where the fermionic quark fields have been integrated out. D_{lat} denotes a generic, massive lattice Dirac operator, for instance, the Wilson-Dirac operator $D_{\mathrm{lat}} = D_{\mathrm{W}} + m_0$.

The number of integration variables in

$$[\mathrm{d}\varphi] \equiv \prod_{x,\alpha} \mathrm{d}\varphi_{x\alpha}$$

can easily be of the order of 10^6 and more and it is obvious that the only possibility to evaluate the functional integral is to use Monte Carlo integration.

Numerical simulations are a two step procedure. In the first step one has to generate a set of gauge configuration. Thereafter, the second step is to evaluate expectation values with on this set.

A configuration in the set of gauge configurations represents the collection of all link variables on the lattice

$$\{U_\mu(x) \mid x \in \Lambda_E, \mu = 0, \ldots, 3\}.$$

A collection of an infinite number of configurations is called an ensemble.

The weight factor

$$W = \det(D_{\text{lat}}) \exp(-S_{\text{G}}[U])$$

makes sure that the integrand is strongly peaked around those configurations for which W is large. Therefore, it is possible to replace the ensemble with a finite sample of gauge configurations which is dominated by those configurations for which W is large.

A sample is generally produced by an *updating process*. Each step in the updating process generates a new configuration $\{U_\mu(x)\}_{i+1}$ from an old configuration $\{U_\mu(x)\}_i$. The probability for this transition is a function of the statistical weights of the two configurations, W_i and W_{i+1} respectively.

Then in the end to make the algorithm exact again, an *accept-reject step* is performed which assures that each configuration can be reached with the updating process. Otherwise, the updating algorithm would automatically converge in the most probable configuration and it would not be possible to cover the whole configuration space.

The quark determinant $\det(D_{\text{lat}})$ can be written as

$$\langle A \rangle = Z^{-1} \int [\mathrm{d}U] A[U] \, \mathrm{e}^{-S_{\text{eff}}[U]}$$

where the *effective gauge action* is introduced

$$S_{\text{eff}}[U] \equiv S_{\text{G}}[U] - \log\det(D_{\text{lat}}[U]) = S_{\text{G}}[U] - \mathrm{Tr}\log(D_{\text{lat}}[U]).$$

The quark determinant can also be written as a functional integral over an auxilliary complex scalar field ϕ. We have, namely,

$$\det(D^\dagger D) \propto \int [\mathrm{d}\phi^\dagger \mathrm{d}\phi] \exp\left(\phi^\dagger (D^\dagger D)^{-1} \phi\right).$$

This formulation can be used for example in QCD with two degenerate flavours for the calculation of the quark determinant. To evaluate the functional integral stochastically,

one generates the *pseudofermion field* ϕ and applies the inverse of the lattice Dirac operator.

In dynamical numerical simulations the evaluation of the inverse quark determinant can easily account for 80% of the computational costs. Therefore, it is necessary to have an efficient implementation for the application of the lattice Dirac operator suitable to work in iterative inversion algorithms.

We can see that the evaluation of the inverse fermionic matrix $(D^\dagger D)^{-1}$ is a crucial point for numeric simulations. The main difficulty for fermionic theories is, that the quark determinant in the effective action is non-local. This makes the numerical evaluation of the transition probabilities in the Monte Carlo updating process rather slow.

In the beginning of numerical simulations in the 1980s, the available computer power was not sufficient enough to include the quark determinant. Instead, the so-called *quenched approximation* was used. For quenched simulations the quark determinant was set to a constant, i.e. $\det(D_{\text{lat}}) = 1$ which implies that the effects of virtual quark loops are entirely suppressed. The quenched approximation is based on the assumption that most of the non-perturbative contributions are carried by the gauge field.

For several quantities, such as the masses of the lightest hadrons, the error for using a quenched simulation amounts to just 10% – 15% [B$^+$98, A$^+$00]. However, it is clear that dynamical quark effects must be taken into account in order to arrive at reliable, non-perturbative predictions with a total accuracy at the percent level.

2.7 Fermion matrix inversion

The inversion of the fermion matrix occurs at two crucial points in a lattice simulation. Firstly, for dynamical fermions we have seen that the inverse of the fermion matrix appears for computation of the quark determinant. Secondly, for the computation of masses of the simulated fermions it is necessary to know the *fermion propagator* which is basically just the inverse lattice Dirac operator.

There are several algorithms[GSZ90, Fro03, Gre97] for the inversion of a generic operator A, that is, to find the solution vector x for the equation

$$Ax = b. \tag{2.2}$$

I will give a quick overview about different techniques to motivate why the crucial part to write efficient simulation code is to optimize the implementation of the lattice Dirac operator for optimal performance.

For our special problem, the operator A would be the hermitian lattice Dirac operator $A = D$ which in the case of the Wilson-Dirac operator can be written in terms of the hopping matrix as $D = I - \kappa M$. The source b is the pseudofermion field ϕ and x corresponds to the solution vector for the initial equation $x = A^{-1}b$.

The most simple iterative method to solve this equation is the *Jacobi iteration* which is defined

$$x_0 \equiv b, \qquad x_{i+1} = b + \kappa M x_i, \quad i \geq 0.$$

The solution is given by $x = \lim_{i\to\infty} x_i$, if the limit exists. The convergence is guaranteed, if the absolute value of the largest eigenvalue of κM is less than 1.

Another iterative scheme is the *minimal residue* iteration. In general the *residue* after i iteration steps is defined by

$$r_i \equiv b - Ax_i. \tag{2.3}$$

The iteration starts from an initial guess x_0 for the solution, and then for $i = 0, 1, 2, \ldots$ one calculates

$$x_{i+1} = x_i + \frac{(Ar_i, r_i)}{|Ar_i|^2} r_i$$

which implies

$$r_{i+1} = r_i - \frac{(Ar_i, r_i)}{|Ar_i|^2} Ar_i.$$

Here and for the rest of this work $(\cdot,\cdot)$ denotes the usual inner product of two vectors and $|\cdot|$ the norm. The iteration is stopped if $r_i = 0$ when x is the exact solution, or if the absolute value of the residue $|r_i|$ is at least smaller than some given small value δ. We can see that for each step in the iteration a multiplication of A is necessary which takes most of the computational expense.

The most popular iterative method for the inversion of the fermion matrix is the *conjugate gradient* iteration[PTVF92]. One may show that the solution to Eq. (2.2) minimizes the function

$$q(z) = (b, z) - (z, Az)$$

and that the minimum is non-degenerate[She94].

The conjugate gradient algorithm starts from some arbitrary initial guess x_0 for the solution x and then constructs a sequence of vectors x_i, $i = 0, 1, \ldots$, such as $q_i = q(x_i)$ is monotonically decreasing. Given x_i and a *search direction* p_i the next vector x_{i+1} is determined through

$$x_{i+1} = x_i + \alpha_i p_i$$

where α_i is chosen such that $q_{i+1} \equiv q(x_{i+1})$ becomes as small as possible. The search directions can be found from the gradient $g(z)$ of the quadratic form $q(z)$

$$g(z) = b - Az$$

which is equivalent to the residue r_i in Eq. (2.3). The gradient $g_i = g(x_i)$ points to the steepest descent at x_i. One could choose $p_i = g_i$ for the search direction, however, a better choice is given by

$$p_0 = g_0, \qquad p_{i+1} = g_{i+1} + \beta_i p_i, \quad i \geq 0$$

where β_i is determined through

$$(p_i, Ap_{i+1}) = 0.$$

Explicit expressions for the coefficients α_i and β_i can be found in [KS96] but are of no further interest here.

The algorithm stops when the current vector for the search direction p_i vanishes. From the definition above, this is possible, if and only if the gradient g_i vanishes, from which it is clear that $x_i = x$, i.e. the exact solution of the linear system has been found. However, in practice it is required that the residue r_i^2 is smaller than some small value ε^2. With this stopping criterion it can be assured that the solution is found up to a precision of ε despite any rounding errors introduced in the algorithm.

2.8 Neuberger operator

Not only for the inversion of the lattice Dirac operator an efficient implementation is necessary, but also for the investigation of chiral fermions, for example using the Neuberger operator.

The Neuberger operator can be written as a function of the massless Wilson-Dirac operator. The definition involves the sign-function which is essentially the step function of the argument and has to approximated since its definition as an infinite polynomial can not be realised as a computer program.

Chebychev approximation of sign(Q) Each time the Neuberger operator is applied to a fermion field η on the lattice, one needs to calculate the action

$$\mathrm{sign}(Q)\eta.$$

For numerical stability[PTVF92], *Chebychev polynomials* are chosen for the polynomial approximation of the sign-function of the operator Q. The Chebychev polynomials T_n are defined[BS91] by

$$T_n(x) = \cos(n \arccos x)$$

and the first few are explicitly given by

$$T_0(x) = 1$$
$$T_1(x) = x$$
$$T_2(x) = 2x^2 - 1$$
$$T_3(x) = 4x^3 - 3x.$$

Chebychev polynomials are orthogonal in the interval $[-1, 1]$

$$\int_{-1}^{1} \mathrm{d}x \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}} = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases}$$

and T_n has exactly one maximum with $T_n(x_{\max}) = 1$ and one minimum with $T_n(x_{\min}) = -1$. Because of that, the Chebychev polynomials are very useful in polynomial approximations of functions.

The task is to find an approximation of $\mathrm{sign}(Q)\eta$ to a specified precision. We are looking for a polynomial $P(y)$ of degree n that minimizes the deviation

$$\delta = \max_{\varepsilon \leq y \leq 1} |h(y)|, \quad h(y) \equiv 1 - \sqrt{y}P(y)$$

for a given $\varepsilon > 0$. In the interval $\sqrt{\varepsilon} \leq |x| \leq 1$ the function $xP(x)$ approximates $\mathrm{sign}(x)$ uniformly with a maximal deviation of δ. A polynomial approximation that minimize the maximal relative error is referred as *minmax polynomial*. Existence and uniqueness for a minmax polynomial can be shown, if the function that is to be approximated, does not vanish.

Using Chebychev polynomials a general Ansatz is made such as

$$P_{n,\varepsilon}(y) = \sum_{k=0}^{n} c_k T_k(z), \qquad z = \frac{2y - 1 - \varepsilon}{1 - \varepsilon}$$

and the coefficients c_k have to be adjusted so that the deviation δ is minimized. Details about the actual implementation to find the minmax polynomial can be found in

[GHLW03]. The sign-function then will be approximated by

$$\text{sign}(Q) \approx XP_{n,\varepsilon}(X^2), \qquad X \equiv Q/||Q||.$$

If ε is chosen so that $Q^2 \geq \varepsilon||Q||^2$, the error in this formula is an operator with norm less than or equal to δ. In other words, the approximation error is always bounded by $\delta|\eta|$, uniformly in the field η to which the operator is applied.

The most elegant method to evaluate the polynomial approximation

$$f(x) \approx \sum_{k=0}^{N} c_k F_k(x)$$

is to use *Clenshaw's recurrence formula*, if the base functions $F_k(x)$ obey the recurrence relation

$$F_{k+1}(x) = \alpha(k,x)F_k(x) + \beta(k,x)F_{k-1}(x).$$

For the special case of the Chebychev polynomials the functions α and β are given by

$$\alpha(x,k) = 2x \qquad \beta(x,k) = -1.$$

Define the quantities y_k for each $k = N, N-1, \ldots$ by the recurrence

$$y_k = \alpha(k,x)y_{k+1} + \beta(k,x)y_{k+2} + c_k,$$

with the initial values $y_{N+2} = y_{N+1} = 0$. If the definition of y_k is solved for c_k and the sum in the approximation is written explicitly, the only surviving terms are given by

$$f(x) = \beta(1,x)F_0(x)y_2 + F_1(x)y_1 + F_0(x)c_0.$$

Therefore, in order to evaluate the approximation it is only necessary to know the first two polynomials. Then it is sufficient to make one pass down the y_k's with the definition until y_2 and y_1 are calculated, and then apply the recurrence formula and get the desired answer.

In case of Chebychev polynomials this translates to

$$d_j = 2xd_{j+1} - d_{j+2} + c_j, \qquad j = n-1, n-2, \ldots$$

with the initial values $d_{n+1} = d_n = 0$. Then the approximated function can be evaluated by

$$f(x) \equiv d_0 = xd_1 - d_2 + \frac{1}{2}c_0.$$

Low-mode projection In most practical cases the operator Q^2 can have some exceptionally low eigenvalues which which can negatively impact the rate of convergence properties of the inversion algorithms. In that case, the straightforward approach mentioned above is not possible and it is necessary and far more efficient to separate the few lowest modes and treat them exactly. This should be done in such a way that the error of the total approximation still remains under control.

The spectrum of Q in the vicinity of the origin can be reliable determined by the *Ritz functional* of Q^2. This technique has the advantage that it also yields to an approximation of the associated eigenvectors.

In general terms, a linear operator A acts on a complex vector space V of dimension N. It can be assumed that V is equipped with a positive definite scalar product $(\cdot, \cdot)$ and that

$$(v, Aw) = (Av, w) \quad \forall\, v, w \in V,$$

i.e. A is a Hermitian operator on V. Then there exists an orthonormal basis v_i, $i = 0, 1, \ldots, N-1$, of eigenvectors such that

$$Av_i = \lambda_i v_i.$$

Without loss of generality the eigenvalues can be assumed to be ordered such as

$$\lambda_0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_{N-1}.$$

For the low-mode projection only the l lowest eigenvalues of A are important. It is safe to expect that these eigenvalues λ_k, $k = 0, 1, \ldots, l$ are separated from zero and from the rest of the spectrum of A by a distance greater than ϱ.

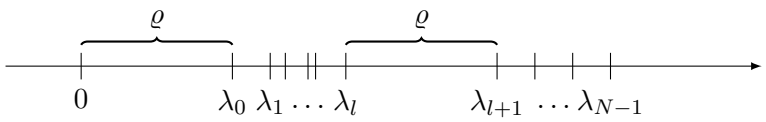

Figure 2.3: An example for the spectrum of the linear operator A. The operator A is supposed to be positive definite and the low-lying eigenvalues are separated from zero and from the rest of the spectrum of A by a distance of ϱ.

Now the Ritz functional is defined by

$$\mu(z) = \frac{(z, Az)}{(z, z)}$$

which has to be minimized for all non-zero vectors $z \in V$. The minimum is attained if z lies in the subspace corresponding to the lowest eigenvalue λ_0 and the minimum of the Ritz functional is precisely equal to λ_0.

To find the minimum of the Ritz functional a conjugate gradient method is used. Again, a sequence of vectors x_i, $i \geq 0$ is constructed

$$x_{i+1} = x_i + \alpha_i p_i,$$

where p_i are the search directions for the conjugate gradient as explained above. The parameter α_i is tuned such that $\mu_{i+1} \equiv \mu(x_{i+1})$ is minimized.

The gradient of the Ritz functional $\mu(z)$ is given by

$$g(z) = \frac{[A - \mu(z)]\, z}{(z, z)}.$$

As before we set $g_i \equiv g(x_i)$. The gradient satisfies

$$(x_i, g_i) = 0$$

because the Ritz functional is invariant under scale transformation, i.e. $\mu(az) \equiv \mu(z)$. This implies that the search directions p_i should be chosen such that

$$(x_i, p_i) = 0.$$

The recurrence for the conjugate gradient is defined as above

$$\begin{aligned} p_0 &= g_0 \\ p_{i+1} &= g_{i+1} + \beta_i \left[\frac{p_i - x_{i+1}\left(x_{i+1}, p_i\right)}{\left(x_{i+1}, x_{i+1}\right)} \right], \qquad i \leq 0 \end{aligned}$$

where β_i is given by

$$\beta_i = \frac{\left(g_{i+1}, g_{i+1}\right)}{\left(g_i, g_i\right)}.$$

To understand the convergence of the algorithm the *regularity* of the algorithm is defined such that the algorithm is regular at i if x_i is non-vanishing and if one of the following conditions are met

(a) $g_i = p_i = 0$ and the algorithm terminates.

(b) g_i and p_i do not vanish and the absolute minimum μ_{i+1} is attained for some finite value of α_i.

It can be proven, that if the algorithm is regular at i, and if it does not terminate at this point, it is also regular at $i + 1$. With the initial condition $p_0 = g_0$ the algorithm is regular at $i = 0$.

In the sense of regularity the algorithm proceeds smoothly until it terminates which is the case if $g_i = 0$. From the definition of the gradient one can see that x_i is an eigenvector of A which means the algorithm does stop if and only if an exact eigenvalue is found.

However, in practice the algorithm is stopped, if some criterion is satisfied. The usual choice is to stop the algorithm when

$$|\hat{g}_i| < \omega,$$

where ω is some fixed value. Here, $\hat{g}_i$ is defined as the gradient of the normalized value $\hat{x}_i \equiv x_i/|x_i|$.

The stopping value ω also plays the role for the error estimation. If z is a unit vector in V and if the stopping criterion $|g(z)| < \omega$ is met, then one can prove that there exists an eigenvalue λ to A such that $|\rho - \mu(z)| < \omega$. This allows to control the error on the computed eigenvalues in a most direct and reliable way. In particular, the accumulation of rounding errors during the recursion is completely irrelevant: once the algorithm stops, the error is known since the stopping criterion is met in the last step. It is also possible to specify a relative error by choosing ω to be proportional to the current value of the Ritz functional.

Once the Ritz functional is minimized to a certain precision the vector $\hat{x}_i$ can be expected to have converged towards the eigenvector v_0. Wether this is really true will not be important because only the eigenvalues are of interest and these will be obtained with controlled errors. The eigenvector to the next eigenvalue λ_1 lies in the orthogonal subspace to $\hat{x}_i$ of V, where $\hat{x}_i$ is the last vector constructed before the minimization of the Ritz functional was stopped.

Let $w_i = \hat{x}_i$ then V_1 is the subspace of V orthogonal to w_1. The corresponding projector $\mathbb{P}_1$ is given by

$$\mathbb{P}_1 z = z - w_1 (w_1, z)$$

and $A_1 = \mathbb{P}_1 A \mathbb{P}_1$ is a linear operator in V_1. The lowest eigenvalue of A_1 may now be determined by applying the algorithm above with the simple substitutions $V \to V_1$ and $A \to A_1$ in all formulae. In particular, the initial vector $\hat{x}_0$ must be contained in V_1 and the gradients must be computed with the matrix A_1 instead of A.

When the algorithm stops one has found a vector w_2 which is an approximation to the eigenvectors v_2 of A. This procedure can be iterated by minimizing the Ritz functional in the orthogonal subspace to w_1, w_2, and so on. In this way a sequence $w_0, w_1, w_2, \ldots, w_n$ of orthonormal vectors is obtained such that

$$\mathbb{P}_{k-1} A w_k = \mu(w_k) w_k + r_k, \qquad k = 1, \ldots, l,$$

where the residue $|r_k| < \omega$ and it is convenient to define projectors to the orthogonal subspace to eigenvalues with positive and negative eigenvalues λ_i such that

$$\mathbb{P}_+ z = z - \sum_{\lambda_k > 0} w_k \, (w_k, z)$$
$$\mathbb{P}_- z = z - \sum_{\lambda_k < 0} w_k \, (w_k, z)$$

Similar it is possible to define the projectors $(\mathbb{P}_\pm)_{\text{exact}}$ which projects the orthogonal to the subspace spanned by the exact eigenvectors v_i if we substitute $w_i \to v_i$.

Now the question arises how accurately the computed projectors $\mathbb{P}_\pm$ approximate the exact projectors $(\mathbb{P}_\pm)_{\text{exact}}$. The answer depends on the size of the residues

$$\varrho_k = || \, (Q - \lambda_k) \, w_k ||$$

and also on the distance between the eigenvalues of Q. Rather than to estimate the deviation of the individual eigenvectors, what is interesting is the deviation of the projectors and therefore the distance d_k of λ_k from the exact spectrum of Q in the subspace orthogonal to the range of $(\mathbb{P}_+)_{\text{exact}}$ if $\lambda_k > 0$ or $(\mathbb{P}_-)_{\text{exact}}$ if $\lambda_k < 0$. The quality of the approximation is then controlled by the parameter

$$\kappa_\pm^2 = \sum_{\pm\lambda_k > 0} \varrho_k^2 / d_k^2.$$

It can be proven, that the inequality

$$||\mathbb{P}_\pm - (\mathbb{P}_\pm)_{\text{exact}}|| \leq \frac{\kappa_\pm (1 + 2\kappa_\pm)}{1 - 2\kappa_\pm (1 + 2\kappa_\pm)}$$

holds[GHLW03]. It is noteworthy that for practical uses $\kappa_\pm \ll 1$.

Once the low-lying eigenvalues and eigenvectors have been computed it is possible to substitute the sign-function by

$$\text{sign}(Q) \simeq \mathbb{P}_+ - \mathbb{P}_- + (1 - \mathbb{P}_+ - \mathbb{P}_-) \, X P_{n,\varepsilon}(X^2), \qquad X \equiv Q/||Q||,$$

where ε is set to a value equal to or slightly less than the smallest eigenvalue of $Q^2/||Q||^2$. The associated approximation $\widetilde{D}_N$ of the Neuberger operator then satisfies

$$||\widetilde{D}_{\mathrm{N}} - D_{\mathrm{N}}|| \leq \frac{1}{a}(1 + s - \frac{1}{2}am)\left\{2\left(\kappa_+ + \kappa_-\right) + \delta\right\}$$

up to term proportional to $\kappa_\pm\delta$ and $\kappa_\pm^2$.

The program to compute the projectors and replace the Neuberger operator D_{N} by its approximation $\widetilde{D}_{\mathrm{N}}$ is straightforward. The number of low modes to be included in the projectors should be determined dynamically in such a way that the spectral distance from the other mode is not very small by accident. The parameters $\kappa_\pm$ can be estimated without difficulties and the minimization of the Ritz functional is stopped when the desired level of precision is reached.

Chapter 3

Compute Unified Device Architecture

Since the introduction of *hardware transform and lighting* (T&L) with the NVIDIA GeForce 256 (NVIDIA Corporation, 1999), a lot has changed in the world of graphics computing.

In the following I will show a brief overview of the development of graphics processing units (GPU) in the past ten years and how it came that it is now possible to solve problems with the help of a GPUs which are not necessarily tied to computer graphics.

After that, I will give a short technical explanation about the shader units—the core hardware elements for GPUs—of the NVIDIA G80 chip set and summarize the CUDA programming model. In the end, I will talk about what is necessary for CUDA programs to achieve optimal performance.

For the reader who is not that familiar in the field of computer graphics, I try to give a short description for the many technical terms in this chapter. A glossary can be found in the appendix.

3.1 Historical development

The two most important parts in the art of computer graphics are transform and lighting. Transform is the task of converting three-dimensional coordinates of a virtual scene to the two-dimensional view of the screen. Lighting is the task of taking light objects in a virtual scene and calculating the resulting color of surrounding objects as the light falls upon them.

By the time being, both tasks where accomplished on the CPU in software, either by the program itself or by the display driver. But with scenes growing in complexity memory-intensive rendering could not be performed in real-time anymore since CPUs are generally optimized for low latency computations. Both tasks, however, involve massive applications of matrix-vector operations. For transformations to the rendering plane, it is necessary to apply the projection matrix to each vertex in the scene and for lighting calculation the angle between the direction of the light and the normals of the faces plays an important role. Matrix-vector operations can be implemented in hardware easily and since every calculation was independent of others both tasks could be massively parallelized.

NVIDIA was among the first manufactures of graphics processing units to introduce hardware T&L with the GeForce 256 chip set which would take calculations for transform and lighting from the CPU and serve as a *coprocessor* to the host system for those computations. First benchmarks showed an enhancement in performance up to 50%.

There are two standard graphics programming interfaces that are used in most commercial applications: DirectX (Microsoft 1995) and OpenGL (Silicon Graphics Inc., 1992). While OpenGL focuses on a stable API which does not change very often, Microsoft releases new versions of DirectX bound tightly to GPU release schedules. Over the time DirectX more and more defined a certain feature set which hardware vendors had to implement in their products. As a result, it has become very convenient to classify GPUs by the DirectX version they support.

Hardware T&L became an essential feature with the release of DirectX 7.0. Both leading vendors at that time —NVIDIA and ATI— released products with full DirectX 7.0 compliance with the NVIDIA GeForce 2 (NVIDIA Corporation, 2000) and the ATI Radeon 7000 series (ATI Technologies, 2000). By the end of 2001, hardware T&L was integrated in every chip set.

Despite the growing compute power of graphics processors the hardware was not yet suited for general purpose computations. The GPU was not programmable beyond advanced texture blending capabilities and calculation precision was limited to eight bits per channel.

The next step towards the current hardware technology was taken with the definition of *programmable shaders* in DirectX 8.0 and the implementation with the NVIDIA GeForce 3 (NVIDIA Corporation, 2001). Shader architecture replaced the former fixed-function rendering pipeline with a highly flexible programming pipeline. A shader program is a small set of software instructions intentionally designed to achieve coloring on a per-pixel basis. The application of pixel shaders range from applying a lighting value to the scene to complex operations like bump mapping, shadows, specular highlights and other dynamical effects.

For each specific class of application dedicated types of shaders existed in the beginning. Besides pixel shaders, we have already learned about, another type is a vertex shader responsible for operations on each vertex in the three-dimensional scene. The purpose of this shader is to transform three-dimensional world coordinates to two-dimensional on-screen coordinates. Vertex shaders can manipulate properties such as position, depth component or color of a given vertex, but it is not able to create new vertices. Later this was possible with the introduction of geometry shaders which could add and remove vertices from the rendered mesh to add volumetric detail to an existing mesh that would be to expensive to process on the CPU.

Shader programs act independently on a large dataset with a given transformation which is a crucial prerequisite for efficient parallel processing. To exploit parallelism further, shader hardware was constructed in multiple pipelines.

The shading kernel was mostly written in assembler until higher level languages like *Cg*, *HLSL* or OpenGL's *GLSL* were developed. General purpose problems had still to be "translated" to a graphics language and then fed to the shaders.

With DirectX 9.0 and the NVIDIA G80 chip set (NVIDIA Coorporation, 2007) the door was opened for the current generation of graphics processing units. It introduced the *unified shading model* which overcame the distinction between different shader types. Furthermore, shader programs were extended to enable data-dependent branching and full floating point precision throughout the graphics pipeline.

The graphics processor changed from a hardware to display vertices and lighting on the screen to a flexible, self-balanced shading architecture with a decoupled and threaded data processing.

3.2 Shader architecture

The core of the G80 chip set is a homogeneous array of floating point processors. Each processor is a scalar *arithmetic logical unit* (ALU) with FP32 precision and rounding properties that conform to the IEEE754 standard[P7585].

This array is grouped in clusters of 16 scalar processors together with an own scheduler, a register file, special function units, a unit for data address and setup and the texture unit. Those scalar processors are further organised into two pairs of 8, called a *streaming multiprocessor*. The scheduler is able to run the same instruction on each multiprocessor across a number of cycles, depending on the work.

The whole cluster can be viewed as 8-way MIMD (*multiple-instruction multiple-data*) setup of 16-way SIMD (*single-instruction multiple-data*) scalar processor clusters. In the end, each thread gets mapped to one scalar processor core and each scalar thread executes independently with its own instruction address and register state. NVIDIA gives this architecture its name, SIMT (*single-instruction multiple-thread*). The key difference to SIMD is that SIMD vector organizations expose the SIMD width to the

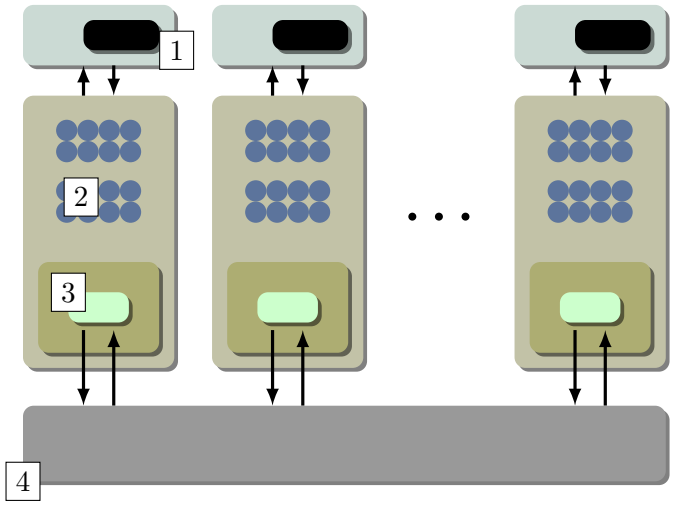

Figure 3.1: Simplified illustration of the shader core architecture of the NVIDIA G80 chip set. Up to 8 pairs of streaming multiprocessors are arranged on the chip, each consisting of (1) a instruction scheduler and the register file, (2) 16 scalar processors and (3) the texture fetching unit and the texture cache. Each multiprocessor has access to the (4) global memory.

software, whereas SIMT instructions specify the execution and branching behavior of a single thread. That way the programmer is able to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For correctness it is safe to ignore the vector width, but it is necessary to keep it in mind for optimal performance.

Scalability plays an important role in the design of the hardware. The program must be insensitive to the number of multiprocessors, and a programmer should be able to write one program for any number of multiprocessors which runs on differently sized GPUs without recompiling.

Threads within a block get scheduled in groups of parallel threads called *warps*. The warp size, similar to the vector width in a SIMD architecture, for the G80 chipset is 32. Individual threads within a warp get executed physically at the same time and start together at the same program address but are free to branch and execute independently. Every instruction issue time, the scheduler selects a warp that is ready to execute and

issues the next instruction to the active threads of a warp. Sometimes the notion of a *half-warp* is used, a half-warp is simply the first or the last 16 threads in a warp.

Full efficiency is realized when all threads in a warp can execute a common instruction path. If threads in a warp diverge because of a data-dependent conditional branch, the warp serially executes each path of the branch disabling threads that are not on that path until all paths are complete and the threads converge back to the same execution path. Branch divergence can only occur within a warp as different warps execute independently.

The multiprocessor can independently create, manage and execute concurrent threads with zero scheduling overhead. Context switching is basically free at every cycle. Hardware allocates resources—thread slot, registers and shared memory—and a block does not run until enough resources are available. Threads are dynamically allocated to do the work which is currently needed, the goal is, to keep the scalar processors as busy as possible. The strategy behind this is to hide memory latencies with computation through context switching which means that parallelism is necessary for performance.

Each multiprocessor has access to its own pool of each of the four types of on-chip memory besides several layers of L1 and L2 caches but only those four are explicitly exposed and controllable by the user. In short, those four memory pools consist of

- a *register file* of 32-bit wide floating point registers,
- a *shared memory* which provides low latency temporary storage and is important for inter-thread communication,
- a read-only *constant cache* that speeds up reads from the constant memory space which is a read-only region in the device memory
- and a read-only *texture cache* that speeds up reads from the texture memory space which is a read-only region in the device memory.

Shared memory, constant cache and texture cache are shared among the scalar cores within the same multiprocessor. Additionally, each multiprocessor has access to an

off-chip local memory to reduce the usage of register and read-write access to global device memory. Access to both memory spaces are not cached and latencies are around two orders of magnitude higher than latencies for shared memory.

Each multiprocessor performs a texture data fetch via a texture unit which provides certain sampling abilities like filtering. Threads which fetch data can execute asynchronously to threads on the cluster which perform work, allowing the hardware to hide fetch and filter latency as much as possible.

The rest of the chips pipeline is to leverage the shader cores and to present the computed results on the screen. Data from the host CPU gets send to the thread setup units which keep the shader cores as busy as possible with meaningful, non-wasteful work, sorting, organising, generating and presenting data to be worked on. They also implement some pre-shading optimisations to save computation power on work that will never be displayed. A global thread controller takes over, determining the work to be done in any given cycle before dispatching to the shader cluster and their own individual schedulers.

After the data is being processed by the shader cores, it will be passed to raster operation units at the very back end of the pipeline. Raster units are responsible for reading and writing depth and stencil information and doing alpha blending and testing. The final data being processed will be handed over to the display logic and finally displayed on the screen.

These parts of the pipeline are not used for general purpose computing on the GPU and because of that, this will not be discussed further for the rest of this work.

3.3 The programming model

CUDA is designed to transparently scale the available parallelism in the number of multiprocessors just as graphics applications transparently scale their parallelism to the wide varying number of cores. A program must be insensitive to the number of multiprocessor cores and should run on any sized GPU without recompiling. In the

following, I will explain, how this goal of scalable parallelism is implemented and how the programming model maps to the hardware described above.

Execution model The CUDA programming model is based on layers of parallelism of different granularity. A problem to be solved is divided into a series of sequential shader programs which in the CUDA terminology will be called *kernels*. Each kernel is executed by a number of computing parallel blocks and each block decomposes into a number of computing parallel threads. This hierarchy along with shared memory and barrier synchronizations are simply exposed to the programmer and provide fine-grained data parallelism and thread parallelism nested within coarse-grained data parallelism and task parallelism. These abstractions guide the programmer to partition the problem into sub-problems that can be solved independently in parallel and then into finer pieces that can be solved cooperatively in parallel.

There are two possibilities to write CUDA programs: the CUDA driver API, a low-level C API which provides functions to load kernels as modules of CUDA binary code, get information about and launch them. The binary code is obtained by compiling kernels written in C. The other method is using the CUDA runtime a minimal extension to the C language. These extensions allow programmers to define kernels as special C functions and provide new syntax to specify the thread geometry of the shader programs.

CUDA comes along with a tool chain to compile and translate shader programs and link them together with the generic C host code.

The runtime API is built on top of the CUDA driver API. Initialization, context and module management are all implicit and the resulting code is more compact. The use of the CUDA driver API requires more code and is harder to program and debug, but it offers a better level of control and is language independent since it handles binary code only.

Both, the runtime API and the driver API provide functions to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices and so forth.

I decided to choose the CUDA runtime which gives me the possibility to concentrate on the actual physical problem without being distracted by the larger amount of device management which is unavoidable with the CUDA driver API. In the following I will restrict my explanation to the CUDA runtime.

CUDA's programming model assumes that CUDA threads execute on a physically separate *device* that operates as a coprocessor to the *host* running the generic C program. In most cases kernels execute on a GPU and the rest of the C program executes on the CPU. Furthermore, both host and device maintain their own DRAM, referred to as *host memory* and *device memory*, respectively. The host program manages device memory through calls to functions, defined with the CUDA runtime.

Each hardware implementation comes with a specific set of features and hardware can differ in its capabilities. Therefore, CUDA defines the *compute capability* of a device by a major revision number and a minor revision number. Devices with the same major revision number are of the same core architecture. The minor revision number corresponds to incremental improvements to the core architecture. That way programmers can simply define a minimal set of features a device has to support to run the written code. Compute capabilities and relevant properties can be found in the appendix and in the programming guide[NVI09].

Thread hierarchy Each CUDA thread that executes a kernel is given a unique *thread ID* for the purpose of identification of threads inside a kernel function. Within a kernel it is accessible through the built-in variable `threadIdx`. In order to provide a natural way to invoke computations across elements in domains such as vectors, matrices or fields, `threadIdx` is a three-component vector and threads can be identified using a one-dimensional, two-dimensional or three-dimensional thread index. The index of a thread and its thread ID can be given in a straightforward way: in a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + yD_x)$. Likewise, in a three-dimensional block with size (D_x, D_y, D_z), the thread ID of a thread with index (x, y, z) is $(x + yD_x + zD_xD_y)$.

On current GPUs up to 512 threads form a one-dimensional, two-dimensional or three-dimensional thread block. If there are more threads than that multiple blocks are grouped together in an one-dimensional or two-dimensional *grid*. Similar to the thread ID a block within the grid can be identified by a one-dimensional or two-dimensional *block ID*. The block ID is accessible through the built-in variable `blockIdx`.

Since thread blocks are required to execute independently, it must be possible to execute them in any order, in parallel or in series. This independence requirement allows the hardware to schedule thread blocks in any order across any number of cores. The number of thread blocks in a grid is typically dictated by the size of the data being processed, it should greatly exceed the number of processors in the device to fully utilize the hardware.

Kernel invocation We have already encountered the key element for the CUDA runtime, *kernels*. A kernel defines the actual code, processed by the shader cores. A kernel is a special C function that gets executed N times in parallel by N different CUDA threads, as opposed to general C functions which get only once executed. A kernel is defined by using the declaration specifier `__global__` and the number of CUDA threads for each call is specified using a new syntax

```
__global__ void func(float *parm, ...)
{
    // kernel code
}

// somewhere in the code, invoke the kernel function
dim3 Dg, Db;
func <<< Dg, Db >>> (parm1, ...);
```

The special parameters `Dg` and `Db` specify the thread hierarchy as described above. `dim3` is a three-dimensional vector and can hold the dimensions of the grid and the blocks, respectively.

The kernel gets executed asynchronously and making it possible to overlap program code on the host with code on the device. If it is necessary to synchronize host and device, this is possible with an explicit call to `cudaThreadSynchonize()`. Certain operations that depend on the GPU to have finished the kernel computation like memory copies perform a synchronization implicitly.

Memory hierarchy In agreement with the memory layout of a multiprocessor, each CUDA thread can access data from multiple memory spaces during their execution. For intermediate results each thread has a private set of registers from the register file. Registers are not shared between threads and are not dynamically indexable. The compiler is optimized to reuse aggressively registers in order to keep the overall register usage at a minimum. Unlike SSE, in CUDA there is no way to assign registers by hand. Depending on the hardware revision, there are 8192 registers (16384 on 1.3 hardware). Each can store a 32-bit variable, giving 32KB (64KB) memory per multiprocessor. Accessing a register introduces no additional penalty per instruction, but a delay can occur due to register read-after-write dependencies and register bank conflicts. Read-after-write delays can be safely ignored, if there are at least 192 active threads per multiprocessor. The scheduler schedules instructions as optimally as possible to hide bank conflicts. Best results are achieved, if the block size is a multiple of 64.

Each CUDA thread has access to shared memory, visible to all threads within the same block. It has also the same lifetime as a block: once a block finishes, shared memory gets discarded. Current generation hardware provides 16KB shared memory per multiprocessor. Access to shared memory can be as fast as registers for optimal access patterns.

All CUDA threads have access to global device memory. Global device memory is not cached and for optimal bandwidth special access patterns, *coalescing rules* need to be fulfilled. Coalescing is a very important part since the difference between uncoalesced and coalesced access to global device memory can easily be around an order of magnitude. Accessing global device memory introduces an additional latency of 400 to 600 cycles

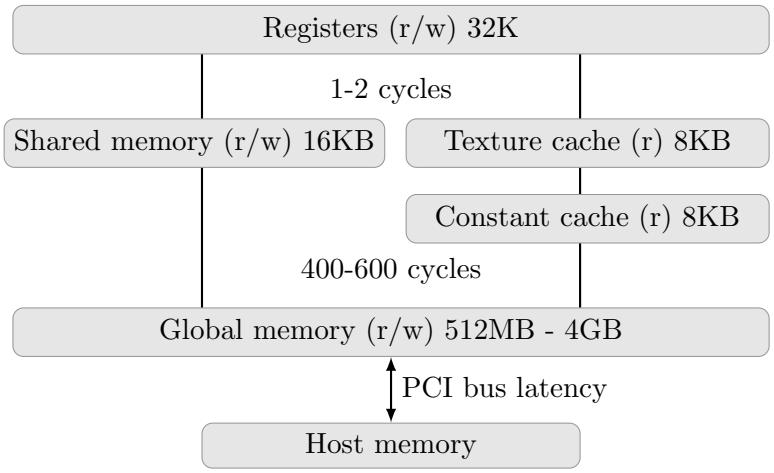

Figure 3.2: Hierarchical overview over the memory spaces available on the GPU. Memory spaces are ordered by access latency from the kernel from low to high latency times. For detailed description see the text.

which can be hidden by the thread scheduler, if there are sufficient enough arithmetic instructions that can be issued while waiting for the memory access to complete.

Because of their great importance to overall performance, optimizing access patterns to shared and global memory will be covered in more detail below.

Additionally there exist two read-only memory spaces accessible by all CUDA threads: constant memory space and texture memory space. Both memory spaces are cached, so a access to them costs one memory read from device memory only in case of a cache miss. If all threads read from the same address in constant cache, data will be broadcasted and the access is as fast as reading from a register. The cost scales linearly with the number of different addresses read by all threads. To benefit from the texture memory, it is necessary to bind a region in device memory as a texture.

A texture can be one-dimensional linear memory or a so called CUDA array. CUDA arrays are opaque memory layouts optimized for texture fetching. They can be one-dimensional, two-dimensional or three-dimensional. A texture element, short *texel*, can be one of the built-in data types CUDA introduces which have 1, 2 or 4 components that may be signed or unsigned 8-, 16-, 32-bit integer, 16- or 32-bit float. Texture cache is

optimized for 2D spatial locality and threads reading texture addresses that are close together will achieve best performance. The total amount of constant memory is 64KB with 8KB cache per multiprocessor. The cache size of texture memory is up to 8KB per multiprocessor.

3.4 Getting optimal performance

In the following, I will give a brief overview how to optimize CUDA code based on several strategies. An optimal programming pattern that emphasizes the different layers of memory should be kept in mind. Each thread should perform the following steps:

(a) Load data from device memory to shared memory.

(b) Synchronize all threads in a block to guarantee that each thread can safely access locations in shared memory written by different threads.

(c) Perform computations on the data in shared memory.

(d) Synchronize again all threads in a block, in order to make sure that shared memory has been updated with the results.

(e) Write results back to device memory.

It is not mandatory that each problem can be mapped to this optimal pattern. However, small modifications are required for those problems and the rest of the discussion persists.

Access patterns to global memory To achieve optimal performance for accesses to global memory, it is necessary to fulfill special access patterns for memory requests.

For global memory *coalescing rules* apply which consists of two statements. Firstly, the device is capable of reading 32-, 64- or 128-bit words from global memory into registers in a single instruction. An assignment such as

```
__device__ type device[32];
type data = device[threadIdx.x];
```

will be compiled into a single load instruction, if and if only `sizeof(type)` is equal to 1, 2 or 4 and variables of type `type` are aligned to `sizeof(type)`. This requirement is automatically fulfilled for built-in data types like `float2` or `float4`.

For structures, size and alignment requirements can be enforced by the compiler using alignment specifiers as `__align__(8)` or `__align__(16)`. For structures larger than 16 bytes, the compiler generates several load instructions. To ensure that it generates the minimum number of instructions, such structures should be defined with `__align__(16)`.

Secondly, global memory bandwidth is used most efficiently, if simultaneous memory accesses of threads in the same half-warp can be coalesced into one or two memory transactions of 32, 64 or 128 bytes. This is possible if all threads of a half-warp satisfy the following conditions:

- Threads must access either 32-bit words which will result in a single 64-byte memory transaction, or 64-bit words which will result in a single 128-bit memory transaction, or 128-bit words which will result in two 128-bit memory transactions.
- All 16 words have to lie in the same segment of size, equal to the memory transaction size (or twice in case of 128-bit words).
- Threads must access words in sequence, that is, the k^{th} threads in a warp must access the k^{th} element in a block of memory being read.

Not all threads of a warp need to participate in the memory access. If there is divergence within a warp, the memory access is predicated.

If a half-warp does not fulfill all requirements above, a separate memory transaction is issued for each thread and throughput is significantly reduced by a factor of up to 16. Coalesced 64-bit access delivers slightly lower bandwidth than coalesced 32-bit access

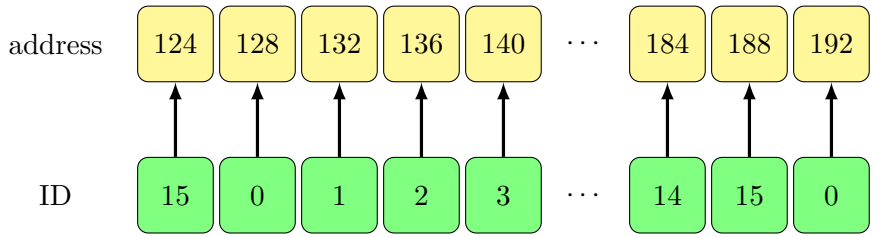

Figure 3.3: Example of coalesced access to global memory. All threads in a half-warp read one memory word in sequential order and the memory access is aligned to 32 byte and only a single memory instruction is issued. Coalescing would also be possible for 64 and 128 byte accesses.

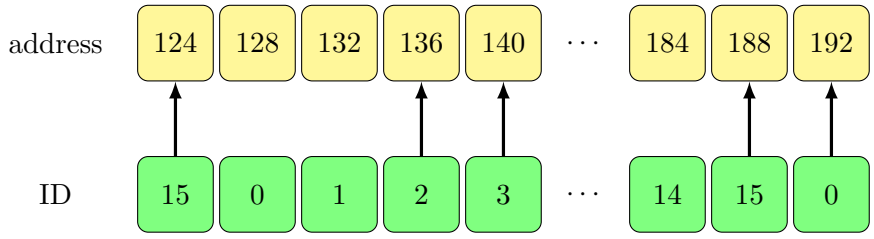

Figure 3.4: Example of coalesced access to global memory. Although not all threads in the half-warp participate in the memory access, a single memory instruction is issued because the memory access can be predicated.

and coalesced 128-bit access delivers noticeably lower bandwidth than 32-bit coalesced access.

If `id` is the thread ID as described above, a common access pattern follows the form

```
BaseAddress + id
```

where BaseAddress is of type `type*`. To get memory coalescing, `type` must meet the size and alignment requirements. If `type` is a structure larger than 16 bytes, it is necessary to split it into several structures which meet the requirements. Data should be laid out in memory as a list of several arrays of these structures instead of a single array of type `type*`. One often refers to *structure of arrays* (SoA) instead of *array of structures* (AoS).

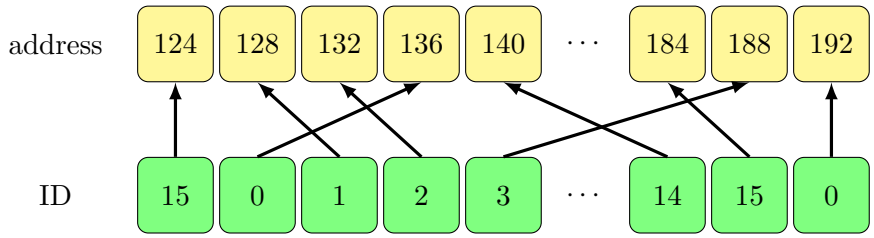

Figure 3.5: Example of uncoalesced memory access to global memory. Threads in the half-warp access global memory in random order which results in the maximum of 16 memory instructions.

Most of the time, threads need data from different positions in global memory and coalesced access is not possible directly. After all, it is often possible to use shared memory to allow a coalesced access if it is used as a caching layer. Each thread can read data coalesced into shared memory and then this data can be redistributed among the threads. After the necessary calculations have been completed, again each thread stores back the result to global memory in a coalesced way.

If memory requests do not follow the access patterns global memory requires to get optimal performance and coalescing can not be achieved even through shared memory, cached texture memory can help to improve total bandwidth and latency, if there is some locality in the texture fetches. However, it is necessary to emphasize that only read accesses can benefit from texture memory as only reading from texture memory is cached.

The coalescing rules have been slightly simplified for recent hardware. Threads do not need to read data from global memory sequentially as long as each half-warp accesses address words in the same 64-bit segment. It is possible to have a random access pattern within the memory segment and even to have multiple threads read the same address. This relaxation makes access to global memory a lot better to handle for programmers, however, I have not yet optimized the code for newer coalescing rules.

Examples of coalesced memory access to global memory are illustrated in Fig. (3.3) and (3.4). A memory access which does not satisfy requirements for coalescing access is given

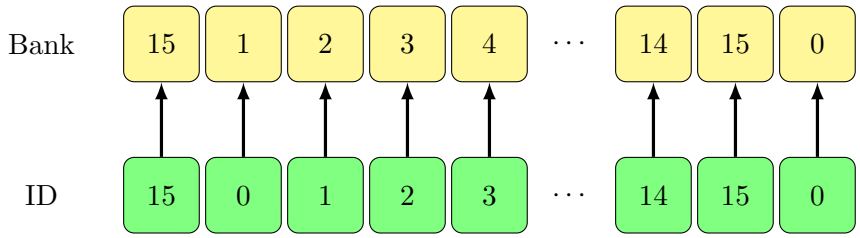

Figure 3.6: Example of access to shared memory without bank conflicts. Each thread in the half-warp accesses a different memory bank.

in Fig. (3.5).

Access patterns to shared memory As with global memory special access patterns need to be fulfilled to achieve optimal performance, even though shared memory is much faster than global memory.

Shared memory is divided into 16 *memory banks* which can be accessed simultaneously. Memory read or write requests made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times higher then the bandwidth of a single bank.

If two addresses fall in the same memory bank, there is a *bank conflict* and the access has to be serialized. The hardware splits a request with bank conflicts into as many separate requests as needed, decreasing effective memory bandwidth by a factor equal to the number of separate requests. If the number of separate memory requests is n, the original request is said to cause a n-way bank conflict.

A request to shared memory by a warp is split into one request for the upper half-warp and one request for the lower half-warp. By definition, there can not be a bank conflict between threads belonging to different half-warps.

In order to use shared memory efficiently and to avoid bank conflicts, it is important to know, how memory addresses map to banks. Each memory bank holds a 32-bit word and successive 32-bit words are assigned to successive banks. In G80, we have $m = 16$ memory banks. Usually a warp accesses an array `shared[]` in shared memory like

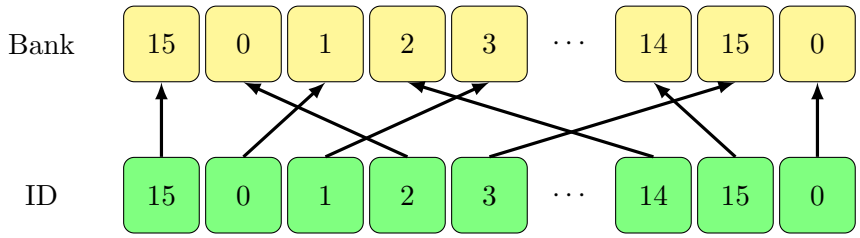

Figure 3.7: Example of access to shared memory without bank conflicts. Although the access is randomly distributed, all threads in the half-warp access a different memory bank.

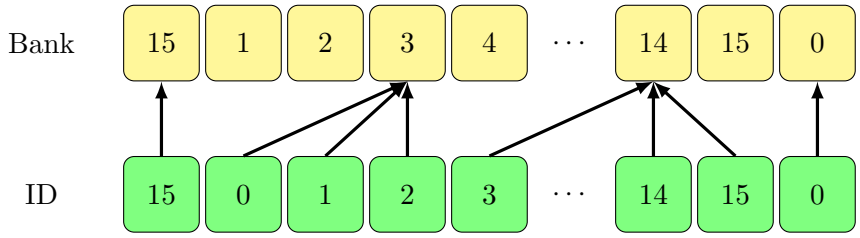

Figure 3.8: Example of access to shared memory with bank conflicts. Memory banks 3 and 14 get accessed multiple times by threads in the half-warp and therefore need to be serialized. However, both memory requests can be combined in a single broadcast, resulting in a two-way bank conflict.

```
shared[BaseIndex + s * id]
```

where id is the thread ID like above and s is some stride. Two threads id and $id + n$ access the same bank, whenever $s \cdot n$ is a multiple of m/d where d is the greatest common divisor of m and s. Therefore, there can be no bank conflict, if and if only half the warp size is less than or equal to m/d. For hardware with compute capability the size of a half-warp is equal to the number of shared memory banks. As a consequence, there will be no bank conflict, if the stride s is an odd number.

Bank conflicts will be further reduced, if several threads of a warp request an address within the same 32-bit word. Shared memory features a broadcast mechanism where a 32-bit word can be broadcast to several threads simultaneously. If all threads read an address within the same 32-bit word, the access is conflict-free.

In Fig. (3.6) and (3.7) memory accesses to shared memory without bank conflicts are shown. A memory access with a two-way bank conflict is given in Fig. (3.8).

Instruction throughput Each instruction takes a finite number of clock cycles to be completed. Therefore, it is necessary to optimize instruction count to achieve an optimal instruction throughput. Throughputs will be given in number of operations per clock cycle per multiprocessor. An *operation* is the execution of an instruction by the whole warp. For a warp size of 32, an operation consists of 32 instructions. If T is the number of operations per clock cycle for a given instruction, its instruction throughput is one instruction every $32/T$ clock cycles.

In the following, only single-precision (FP32) will be assumed. A short discussion about double-precision (FP64) will be given below.

For basic arithmetic instructions like add, multiply and multiply-add (mad) the instruction throughput is 8 operations per clock cycle, division is 0.88 operations per clock cycle, but there exists a less precise version `__fdividef()` with almost twice the throughput.

For reciprocal and reciprocal square the throughput is 2 operations per clock cycle. Square root is implemented as a reciprocal square root followed by a reciprocal. Its throughput is 1 operation per clock cycle.

Special functions like sin, cos, exp will be treated differently. Two paths for the evaluation exists dependent on the magnitude of the argument. For small arguments a fast path will be taken with a throughput of one operation per clock cycle. This path will also be taken for the functions `__sinf()`, `__cosf()` and `__expf()` which are, however, less precise for large arguments. For large arguments a slow path is executed which consists of lengthy computations to achieve correct results over the entire argument range. Also, the slow path requires more registers and local memory to store intermediate results than the fast path. At the end, throughput for the slow path is one order of magnitude lower than for the fast path.

The throughput for integer add is 8 operations per clock cycle, for 32-bit integer multiplication it is 2 operations per clock cycle. Integer division and modulo operations are costly and should be avoided and replaced by bitwise operations, if possible. The compiler will translate (`i/n`) to (`i >> log2(n)`) and (`i%n`) to (`i&(n-1)`) if `n` is a power of 2.

For min, max, cmp and similar operations the throughput is 8 operations per clock cycle, as for any bitwise operation and type conversions.

Flow-Control As with any SIMD architecture flow-control statements can significantly impact the effective instruction throughput in a negative way by causing warps to diverge, that is, to follow different execution paths. Different execution paths have to be serialized, increasing the total instructions executed for each warp.

If the control flow depends on the thread ID to achieve optimal performance, the controlling condition should be written such that it minimizes divergent warps. This is possible because the distribution of warps across the block is deterministic. If the controlling condition only depends on `id/WSIZE` where `WSIZE` is the warp size, no warp divergences occur since the controlling condition is perfectly aligned with the warps.

The compiler may optimize control statements like

```
if (condition)
    ...
```

and

```
switch (condition)
    case value:
    ...
```

as it may unroll loops by using *branch predications* instead. In these cases no warp can ever diverge. With branch predications none of the instructions whose execution depends

on the controlling condition get skipped. Instead, each of them is associated with a per-thread condition code—called *predicate*—that is set to true or false based on the initial controlling condition. While all instructions get scheduled, only those with a true predicate get executed. Instructions with a false predicate do not write results and do not read addresses or read operands.

The compiler uses some heuristics to decide, when it replaces a control statement with a predication. This is based on the number of instructions controlled by the control statement and how likely the control statement produces divergent warps.

General optimization strategies To achieve optimal performance for a given algorithm, one should follow three simple strategies:

- Maximization of parallel execution.
- Optimization of memory usage to achieve maximum memory bandwidth.
- Optimization of instruction usage to achieve maximum instruction bandwidth.

For the first strategy —to maximize parallel execution— it is necessary to structure the algorithm in a way that exposes as much data parallelism as possible. If at some point in the algorithm parallelism is broken, because threads need to synchronize and share data between each other, there are two possibilities. Either these threads belong to the same block in which case they should use the synchronization barrier and share data through shared memory within the same kernel call, or they belong to different blocks, in which case they must share data through global memory, using two separate kernel calls.

Once the parallelism of the algorithm has been exposed, it needs to be mapped to the hardware as efficiently as possible. This is done by carefully choosing the optimal execution configuration for each kernel invocation. Since the amount of registers and shared memory available is shared per multiprocessor, those become the crucial factor. The total number of registers needed is basically $R \times T$ where R is the number of registers

required for each kernel and T is the number of threads per block. The total amount of shared memory for each block is simply the sum of statically allocated shared memory, dynamically allocated shared memory and, because parameters like device pointers get passed to the kernel function through shared memory, the total size of the parameters to the kernel function.

More threads per block are usually better for efficient scheduling, but the more threads per block, the fewer registers are available per thread. However, there should be a minimum of 64 threads per block and should be chosen, only if there are multiple active blocks per multiprocessor. The block size should be a multiple of the warp size, for reasons discussed above a multiple of 64 is better.

The total amount of blocks per grid should be chosen to maximize the utilization of the available computing resources. There should be at least as many blocks as multiprocessors there are on the device. Running only one block per multiprocessor will force the multiprocessor to idle during thread synchronizations and also during memory reads, if there are not enough threads per block to hide the load latency. Usually two or more active blocks per multiprocessors allow an overlap between waiting blocks and running blocks. For this, the amount of registers and shared memory must be low enough to allow more than one active block per multiprocessor.

As a rule of thumb the more blocks stream in a pipelined fashion through the device, the better any overhead can be hidden. It is recommended to have at least 100 blocks available per grid, to be sure, that the application will scale across several device generations an order of 1000 blocks should be sufficient.

It is often not clear *a priori* which suitable configuration should be chosen and a good way is to parametrize the application in the number of threads per block and the amount of shared memory. Benchmarks with different configurations can then yield the optimal configuration for maximum performance.

Parallelism on a higher level can be exposed explicitly by the concurrent execution of tasks. Several functions execute asynchronously and return control to the host thread

before the device has completed the task. Among those functions are kernel functions itself and functions related to data transfer. It is possible to overlap those functions with calculations on the host, to maximize efficiency.

Optimizing memory usage starts with minimizing data transfers with low bandwidth. Bandwidth of data transfer between host DRAM and device is around an order of magnitude lower than for data transfer between global memory and device. Sometimes, the best optimization is to avoid any data transfer in the first place by recomputing data instead wherever needed. Data should be transfered in large chunks to fully utilize the available PCI bandwidth. For smaller data transfers, PCI latency plays a bigger role, which is in the order of 400 – 600 ns. To achieve optimal performance for data transfers, *page-locked memory* should be used, which enables *direct memory access* to the allocated memory regions.

Double-precision calculations Double-precision calculations (FP64) are supported for devices with compute capability 1.3. The GeForce GTX 280 for example, has one dedicated unit for double-precision computations. The rest of the shader core is still running in single-precision. In consequence, this means 1/8 of the total peak performance is available for 64-bit floating-point calculations.

However, the FP64 ALU is still notable in its abilities. It is capable of a double-precision multiply-and-add (MAD) operation in a single clock and supports 32-bit integer computation with no clock penalty which is not available for other double-precision processors already available such as any x86 or Cell.

The MAD operation of the FP64 ALU is intended to accelerate software support for specials and divides, where possible. At the time this work was written, 64-bit calculations on the GPU are not yet comparable to calculations on the CPU, but in the future NVIDIA intends to give support for double-precision at peak performance within their cards.

Chapter 4

Implementation

For the actual implementation of the Wilson-Dirac operator and the Neuberger operator, I was guided by the CPU implementation (Lüscher, 2001) and recent work on implementing a lattice Dirac operator on the GPU [E$^+$07] and [BBB$^+$08].

This implementation was optimized for a NVIDIA GeForce 8800GT which is hardware with compute capability 1.1. The GeForce 8800GT has 14 multiprocessors running at 1.5 GHz clock speed and a physical memory of 512 MB at 900 MHz. Its theoretical peak performance is 504 GFlops/s and the peak memory bandwidth is 57.6 GB/s.

For comparison, I had the opportunity to test the code on a NVIDIA GeForce GTX 280 which is a card with compute capability 1.3. However, I did not optimize for this hardware. The GTX 280 has 30 multiprocessors at 1.3 GHz clock speed and 1 GB physical memory at 1.1 GHz. Its theoretical peak performance is 933 GFlops/s and 141.7 GB/s peak memory bandwidth.

First of all, I will give some thoughts about the the general problem of the Wilson-Dirac operator such as the memory requirements for various elements and try to identify possible bottlenecks. Then the actual implementation of the Wilson-Dirac operator and the Neuberger operator will be described.

4.1 Preliminary considerations

The minimal independent unit for the calculation of the Hermitian Wilson-Dirac operator Q is the computation of one result spinor field $\phi'(x) = Q\phi(x)$ at the lattice site x. The work of [IBP08] showed that it is possible to parallelize at each computed direction of the Wilson-Dirac operator, but the amount of extra effort necessary to eliminate flow-control is not justified by the advantage of this approach.

To recapitulate, with an appropriate rescaling of the spinor fields, the massive, γ_5-Hermitian Wilson-Dirac operator is given by

$$Q\phi(x) = \gamma_5\Big((4 + m_0)\phi(x) - \sum_{\mu=\pm 0}^{\pm 3} U_\mu(x)(1 + \gamma_\mu)\phi(x + \mu)\Big).$$

We can now count the necessary amount of work and memory at each lattice site and, therefore, for each thread. We need to know local storage constraints to find possible thread configurations for the kernel function and construct data layouts which allow for optimal data access.

The spinor field $\phi(x)$ is an object with three color and four spin components and requires 24 floats per lattice site to store. The gauge field $U_\mu(x)$ is a 3×3 matrix with complex entries, giving 18 floats to store per link. Later we will see that the number of independent entries can be reduces via constraints from the fact that $U_\mu(x)$ is an element of the gauge group SU(3).

At each step in the kernel, we need to store the resulting spinor $\phi'(x)$ which can be used to accumulate intermediate results of each direction μ. To compute this intermediate result we need to load and store in local memory one spinor field $\phi(x \pm \mu)$ at the neighbor site and a gauge field $U_\mu(x)$ and $U_\mu^\dagger(x - \mu)$ accordingly. This would require at least 66 float variables or 264 byte to be held in local memory available to each thread at all time.

This amount of memory needs to be stored in the accessible local memory like shared memory and the register file to give a reasonable performance. For shared memory we

have 16384 byte available per thread block. As for the register file, 8192 registers are available and each is capable of holding a 32-bit word which results in 65536 byte.

If we target the suggested 192 active threads per multiprocessor, we are able to store about 64 float variables or 256 byte per thread which would not be sufficient for our computation.

However, the explicit form of the chiral representation for the Dirac matrices γ_μ we have chosen, shows that we have taken into account redundant information. In the chiral representation the Dirac matrices are given as

$$\gamma_\mu = \begin{pmatrix} 0 & e_\mu \\ (e_\mu)^\dagger & 0 \end{pmatrix},$$

where $e_0 = \mathbb{I}_{2\times 2}$ is the identity matrix in two dimensions and e_i, $i = 1, 2, 3$, are 2×2 matrices which can be chosen proportional to the Pauli matrices. If we now take a look at the term $(1 - s\gamma_\mu)\phi$, $s = \pm 1$, we can explicitly give the result in terms of the spinor components of ϕ for each $\mu = \{0, 1, 2, 3\}$.

It turns out that this combination projects the full spinor ϕ into a *half-spinor*, with only half of the components independent. This half-spinor requires just 12 float and since the form of the Dirac matrices is known at each time we can project the spinor field $\phi(x \pm \mu)$ at the neighbor site directly into a half-spinor.

This reduces the amount of local memory necessary from the original 66 float variables to 54 float variables which would fit with previous considerations.

Data layout For an optimal data layout we need to take a look at the data being loaded and stored in each thread. For each direction we have seen that we need to load the spinor field $\phi(x \pm \mu)$ and the gauge field $U_\mu(x)$ or $U^\dagger_\mu(x - \mu)$. The fact that we load data from the neighboring site of the lattice and that this data is distributed in memory more or less randomly depending on the indexing of the lookup table, makes the use of the texture cache mandatory. After having computed each direction, we need

to store the resulting vector $\phi'(x)$. This means, we have to fulfill coalescing rules for the spinor fields by all means, even if we load the spinor fields through the texture unit. Otherwise, we would suffer from large penalties for the memory bandwidth as we have seen in Chap. (3.4).

The spinor field with 24 float variables can be stored into 6 variables of type `float4` which is a 4-component vector. If we align those 6 variables next to each other linear in memory it would require to load and store the spinor fields through shared memory to be able to form an access pattern, suited for coalesced memory transactions. However, we can see that this is not possible with the target number of active threads per multiprocessor of 192. We need to store $24 * 4 = 96$ byte for each spinor which gives $96 * 192 = 18432$ byte necessary which is slightly above the amount of shared memory available.

For our target size, it is not possible to have the spinor fields grouped closely together in memory and still access them in a way that allows for coalesced memory transactions. This means, we need to break up the data structures and rearrange them in a way which is efficient for the hardware to access. Instead of an array of structures we need to design a structure of arrays.

For the gauge fields it is essentially the same except that we always just load gauge fields without the need to store any. If we could rely on the texture cache, we would not be dependent on coalesced memory access at all. The 18 float variables fit into 5 variables of type `float4`. The spare two float variables could be used for further information which is not the case in my implementation though.

The right metric We need to be clear what the optimization goal is. If the kernel will be compute-bound, when there is a high instruction count per data, then we should try to maximize GFlop/s. If our kernel is memory-bound and data access is the limiting factor, then we should try to maximize for effective bandwidth.

For recent generation GPUs, we find a ratio of computing power to memory bandwidth of around 10:1, e.g. the for the GeForce GTX 280 we have stated above a peak performance

of 933 GFlop/s and the peak memory bandwidth of about 141.7 GB/s. For an optimal utilization of the GPU we therefore should have ten floating-point operations per byte of memory we need to load.

The resulting spinor field $\phi'(x)$ of the Wilson-Dirac operator requires the original spinor field $\phi(x)$ as well as the neighboring spinor fields $\phi(x+\mu)$ at all positive directions and $\phi(x-\mu)$ at all negative directions. As a subtotal, this gives $(1+4+4)*24 = 216$ float values or 864 byte for the spinor fields. Additionally, the gauge fields at each connecting link is required. This gives another $8*18 = 144$ float values or 576 byte memory. As a total we need 1440 byte per lattice site.

To compute the resulting spinor field $\phi'(x)$, we project the spinor into half-spinor as stated above. This will just cost 12 floating-point operations per direction giving 96 operations total. Both components of those half-spinors will be multiplied by the complex 3×3 matrix $U_\mu(x)$ or $U_\mu^\dagger(x-\mu)$ respectively. Each multiplication costs 66 operations per direction, giving $66*8*2 = 1056$ operations and so 1152 operations total. In order to accumulate each intermediate result, we have 24 operations for each term in the sum which is an additional count of $8*24 = 192$ operations. For the diagonal term proportional to the original spinor field $\phi(x)$ and the multiplication with γ_5 we need 24 operations both which gives a total of 1392 floating-point operations per lattice site.

Consequently, this gives around one floating-point operation per byte loaded which is typical for bandwidth-optimal problems. Therefore, we should strive for peak bandwidth and take this metric into account, if we evaluate the performance for our implementation. However, I will always give the achieved GFlop/s as a reference to allow a comparison with other projects.

The GeForce 8800GT in the test system has a 256-bit memory interface at a memory clock of 900 MHz. The optimal peak bandwidth we can expect, is therefore 57.6 GB/s. In practical benchmarks, however, only around 40 GB/s could be reached as a peak performance.

SU(3) reconstruction We have seen that the kernel for the application of the Wilson-Dirac operator is bound by the memory bandwidth. Therefore, it would be preferable, if we could save the amount of data to be loaded.

We know that the matrices U_μ are elements of the gauge group G which is in our case the special unitary group SU(3). An element $U \in \mathrm{SU}(3)$ can be represented by complex 3×3 matrices with the properties

$$UU^\dagger = 1, \quad \det U = +1.$$

In general, we need 18 real values to parametrize the matrix, and if we write the matrix explicitly as

$$U = \begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix}, \quad a_i, b_i, c_i \in \mathbb{C},$$

we can interpret each column as a complex vector with 3 components, e.g. $\mathbf{a} = (a_1\ a_2\ a_3)^T$. However, the defining properties for SU(3) give 4 constraints for orthogonality and 6 constraints for normality which all eliminate one degree of freedom.

Therefore, it is possible to parametrize an SU(3) matrix with two vectors $\mathbf{a}$ and $\mathbf{b}$ and drop the last column from the general form above. This vector can be reconstructed simply by the cross product $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ since if $|\mathbf{a}|^2 = 1$ and $|\mathbf{b}|^2 = 1$ then $|\mathbf{c}|^2 = 1$ as required.

For the vectors $\mathbf{a}$ and $\mathbf{b}$ we need to store 12 float values which fit in 3 variables of type `float4`. This would reduce the amount of memory required to load at each direction by a factor 1.6. The missing elements could be reconstructed on-the-fly for the explicit matrix-vector multiplication and the additional work which is necessary for this reconstruction, should be hidden behind memory loads which still dominate the time spent in the kernel.

As noted above, we can take this one step further and just use the minimal amount of numbers to parametrize the gauge fields. There are several possibilities to do so, I have

investigated a method based on [Bro88]. As parameters we can choose 8 angles, 3 angles give the value and 5 angles are for the phases of the matrix elements. An explicit form for the matrix elements are also given in the appendix.

These 8 parameters can be stored in two variables of type `float4`. This would further reduce the amount of memory required for the gauge field by another factor of 1.5.

This parametrization, however, has some drawbacks. The amount of work for the reconstruction is much larger, due to the massive use of trigonometric functions which are quite expensive on the GPU. The second problem is that the mapping from the angles in which we parametrize and the actual matrices they describe, is not easily invertible. At some day it would be desirable to use real data for the gauge fields and then it is necessary to convert this data from the layout it is saved which is usually the explicit form to the data layout we use here.

4.2 Data layout

Now we can talk about the actual data layout I used. With the discussion above we can employ a simple but efficient system for the layout of the spinor and gauge fields the Wilson-Dirac operator will act on. We need to keep in mind that we have to mirror each data set on the device as well as on the host due to the heterogeneous programming model.

One way to rearrange the components of the spinor fields is to group components of different Dirac indices together. The first two `float4` variables would hold real and imaginary parts for the first color components of the Dirac spinor. The index `x, y, z, w` of each variable defines the Dirac index itself. That way we can independently compute the components of the projected half-spinor later and the compiler would be free to rearrange the load instructions for best overlap with calculations.

We want to keep several copies of the spinor fields as working fields for necessity and for practical reasons. The maximal number of those working fields is defined by the

`MAX_NO_FIELDS` constant. The actual set of spinor fields then is defined on the host as a three-dimensional array of pointers to `float4` variables,

```
float4 *ps[MAX_NO_FIELDS][6][VOLUME];
```

in which `VOLUME = L*L*L*T` is the number of lattice sites. Then the spinor field $\phi(x)$ in the working set k would be given by

```
float4 *psk = ps[k][i][x];
```

with `i` $\in \{0, 1, 2, 3, 4, 5\}$. Furthermore, we can identify real and imaginary parts by the index `i`, we access the real part for `i` = even and the imaginary part for `i` = odd. The Dirac index is given as above as the component of `psk`.

For the device memory layout it is recommended only to use linear memory so we need to index calculations ourself. On the device the spinor fields are merely one-dimensional arrays of pointers to variables of type `float4`,

```
float4 *ds[MAX_NO_FIELDS];
```

with `MAX_NO_FIELDS` as above. It is reasonable to keep multiple working sets of spinor fields on the device, as we want to do multiple operations on those fields. This would require a slow data transfer from host to device between each operation otherwise.

On the device the spinor field $\phi(x)$ of the working set k is given by

```
float4 *dsk = ds[k] + i * VOLUME + x;
```

with `i` $\in \{0, 1, 2, 3, 4, 5\}$. Again, the Dirac index is given by the component of `dsk`.

For many operations like data transfer between host and device or texture binding, it is necessary to know the base address of the working fields and the length of each field in memory. The base address is simply the address of the first entry for the working fields—`ps[k][0][0]`—which will be stored for quick reference. The length of each working field is `6 * VOLUME * sizeof(float4)`.

The gauge fields are arranged quite similarly. However, there is no need to regroup them for coalesced memory access as we rely on the texture cache entirely in this case. With each lattice site we can affiliate four links and therefore, four gauge fields, one for every direction $\mu = 0, 1, 2, 3$. Since we regard the gauge field as a background field we only need to store one copy and it is not necessary to keep any working fields for them. Then on the host, the gauge fields are grouped together in a three-dimensional array of pointers to variables of type `float4`,

```
float4 *pu[VOLUME][4][d];
```

`d` = 2 or 3 depending on which parametrization we choose. If we choose column reconstruction, it is preferable if we can independently access different components of each vector. We can assign one `float4` for each complex component of both vectors. The components are then identified by the last index `d`. For the parametrization through angles, we are free to distribute each of them to both `float4` variables.

On the device the gauge fields are kept in a linear array of pointers to `float4` as it is the case for the spinor fields. We also have to calculate our indices ourselves in the following way. The gauge field $U_\mu(x)$ at lattice site x in direction μ is given by

```
float4 *u = ds[x * 4 * 3 + mu * 3 + d];
```

and `d` as above. Then `(*u).x` and `(*u).y` are real and imaginary part of the first vector and `(*u).z` and `(*u).w` of the second vector respectively.

Lookup tables We have seen that we need to access the neighboring lattice sites $x \pm \mu$ of the site x in direction μ. If the spinor and gauge fields are arranged linearly in memory, then we need to know the index of those neighboring sites, so we can access the spinor and gauge fields residing there.

Probably the most efficient way to solve this is using a predefined lookup table where we save the index of the neighboring sites in each direction for each lattice site. The index is just a single integer value for which the texture cache is very efficient. This means,

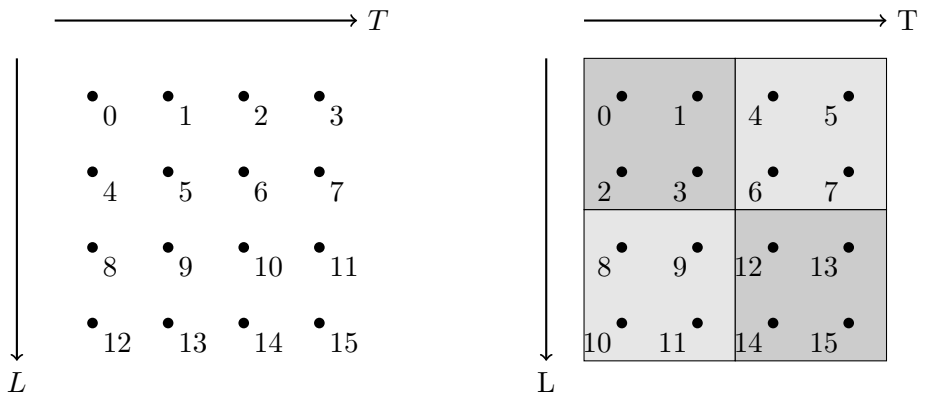

Figure 4.1: Left: The lattice sites are indexed in sequential order. It is easy to see that there are huge jumps for the memory addresses of adjacent lattice sites in the spatial direction. Right: Lattice sites get grouped into blocks. Inside the blocks, sites are indexed in sequential order. Huge jumps only occur between sites of different blocks.

we don't need to take care of aligning the lookup table itself in memory to guarantee optimal access patterns.

We just define two-dimensional arrays of integers for both positive and negative directions

```
int iup[VOLUME][4];
int idn[VOLUME][4];
```

and two linear arrays of pointers to integer variables on the device where we need to take care of the index calculation.

There is a certain degree of freedom how to calculate the lookup table. We have not yet chosen a scheme to map the four-dimensional lattice coordinates x_0, x_1, x_2 and x_3 to an one-dimensional parameter space.

There is a multitude of possibilities to do so, of which the *naïve indexing* is the most simple one. For naïve indexing, we just count each lattice site dimension by dimension and assign the counter to the site. The index i of a site with coordinates x_0, x_1, x_2 and

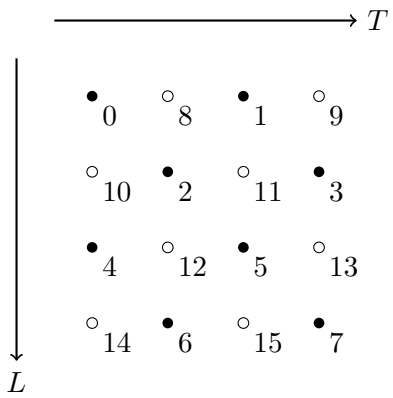

Figure 4.2: An even/odd-indexed lattice. Black dots are even lattice sites, white dots are odd lattice sites. From that scheme it is clear that even lattice sites are surrounded by odd sites only, and vice versa. That way, jumps for the memory addresses can be avoided almost completely.

x_3 on a lattice with a spatial extent of L and T for the time direction is given as

$$i = x_0 L^3 + x_1 L^2 + x_2 L + x_3.$$

The orientation of the lattice is arbitrary and should not have any impact.

We can use this degree of freedom to choose the lookup table to support the texture cache for optimal performance. As we remember the texture cache is optimized for two-dimensional spatial localization which means we should try to avoid huge jumps of the memory address.

We can group lattice sites into *blocks* and assign each lattice site in each block a continuous index. That way, a jump in memory access which could have impact on the performance of the texture cache could only occur, if sites sitting on the border of such a block need to access sites outside the block. The size of the block is a free parameter and can be tuned for optimal performance.

Another often used index scheme is *even/odd-indexing*. This technique comes from the preconditioning of linear systems and can be used in a conjugate gradient solver for the Wilson-Dirac operator to reduce the number of iterations. For even/odd-indexing we

define two sets of lattice sites. A lattice site will be called even if $x_0 + x_1 + x_2 + x_3 =$ even and odd otherwise. We then store the first all the even sites in memory and afterwards all odd sites. Suppose that x is a even lattice site. That way, it is clear that all neighboring sites are odd sites and memory access is limited to that memory region.

4.3 The Wilson-Dirac kernel

With the discussion above, we can now outline the actual kernel which computes the action of the Wilson-Dirac operator Q on a working spinor field ϕ_k and stores the result into the working spinor field ϕ_l.

We assume that the initial spinor field ϕ, the gauge fields U_μ and the lookup tables already reside on the device and that the textures for the gauge field and the lookup tables are already bound and accessible through texture fetches. The texture for the input working field ϕ_k needs to be bound right before the call to the kernel function.

The task for the kernel functions is straightforward. First of all, we identify our threads by their thread ID and compute for which lattice site x we do the calculations. For the accumulation of the resulting spinor field $\phi'(x)$, we need to access allocated memory very often. It is a good idea to have this spinor field stored in registers to prevent long latencies and guarantee optimal performance.

We then process the expression of the Wilson-Dirac operator term by term. We will start with the diagonal part which provides an initialization for the resulting spinor field. We perform the texture fetches for all components of the spinor $\phi(x)$, multiply them by the mass term and store the result to registers.

For each direction μ we first fetch the index of the nearest neighbor of x in direction μ. Then we fetch the spinor field $\phi(x \pm \mu)$ and directly project it to the half-spinor which will be stored into registers. For the gauge field we fetch the parameters for our chosen parametrization and reconstruct the full 3×3 matrix which we will afterwards multiply with the two Dirac components of the projected half-spinor. Both steps can be

combined to reduce the instruction count. The last step is to accumulate the result for each direction to the output spinor field.

Finally we multiply with $-0.5\gamma_5$ and store the result to global memory.

Tricks and tweaks Besides the general strategies for performance optimizations I described in the previous chapter, it is necessary to take a few things into account.

It is noteworthy that we have no explicit control over the registers file from the source code, like it is the case for example in SSE. We can not declare variables to reside in registers only and not get flushed out to local memory as we can not disallow the use of registers. The reason behind this is, that the compiler usually is in a better position to judge what goes into registers. This works well for most kernel functions, especially, if the kernel has only few instructions. As shader functions usually are small programs, the compiler is optimized for this type of code.

The compiler is optimized in order to reuse registers aggressively. However the compiler also reorders instructions to hide the latencies of memory accesses. This reordering can sometimes introduce internal dependencies on intermediate results which prevent the compiler from freeing registers. I have made the experience that especially for a high instruction count this effect increases. NVIDIA, of course, does not give detailed information about the algorithms the compiler uses.

There are a few options to influence the compiler's optimization decisions. One thing is to reuse local variables, if intermediate results are not used anymore. That way the compiler translates the code to instructions which use registers in-place and does not explicitly allocate new ones. More important is, to use scoping of variables and declare temporary variables, needed for calculations inside blocks separated by curly braces.

```
__global__ void
kernel(parameter)
{
    {
        int value;
        value = something();
    }
    // value has lost its scope
}
```

If the variables lose their definition scope the compiler drops the corresponding register and it is more likely that it will be reused instead of the allocation of a new register.

4.4 The Neuberger operator

Now that we have a working and efficient implementation of the Wilson-Dirac operator working entirely on the GPU we can go on and integrate this implementation in a program which applies the Neuberger operator D_N to the spinor field $\psi(x)$.

The implementation of the Neuberger operator contains two distinct parts. First, the evaluation of the minmax polynomial for the approximation of the sign-function via Clenshaw recurrence which is basically only vector-vector addition and the multiplication of a vector with a scalar. The other part is the calculation of the low-lying eigenvalues and eigenmodes of the Wilson-Dirac operator through the minimization of the Ritz functional as described in the first chapter. The calculations involve a lot of linear algebra operations such as scalar products and norms of the spinor fields which need to be implemented on the GPU as well.

For the following discussion we assume the spinor fields to be arranged in large column vectors as mentioned in Chap. (2.3).

For two different spinor fields ψ_k and ψ_l, we define the site-wise operations

$$c\,\psi_k \equiv c(\psi_k)_x \qquad \forall x \in \Lambda_E, c \in \mathbb{C}$$
$$\psi_k + \psi_l \equiv (\psi_k)_x + (\psi_l)_x \qquad \forall x \in \Lambda_E.$$

The application of the Wilson-Dirac operator Q is defined as a matrix-vector operation

$$Q\psi_k \equiv (Q\psi_k)_x = \sum_{y \in \Lambda_E} Q_{xy}(\psi_k)_y \qquad \forall x \in \Lambda_E.$$

Evaluation of the minimax polynomial As we have seen in the first chapter we need to calculate the approximation

$$\text{sign}(Q) \approx X P_{n,\varepsilon}(X^2) \qquad X = Q/||Q||$$

with

$$P_{n,\varepsilon}(y) = \sum_{k=0}^{n} c_k T_k(z) \qquad z = \frac{2y - 1 - \varepsilon}{1 - \varepsilon}$$

and $T_k(z)$ the Chebychev polynomials.

We assume that we have found an approximation of the sign-function to a certain precision. Since this is not a time-critical component of the algorithm we can use the C host code available and further assume that the coefficients c_k of the minimax polynomial will be stored in an global array

```
float *cf
```

on the host.

We need to define some parameters for the Clenshaw recurrence such that

```
float r1 = 4.0f / (norm * norm * (1.0f - eps));
float r2 = -2.0f * (1.0f + eps) / (1.0f - eps);
float r3 = 1.0f / norm;
```

where `norm` is the operator norm $||Q||$ of the Wilson-Dirac operator and `eps` is the precision ε of the polynomial approximation of the sign-function.

Since we need to access those parameters in each thread it is worthwhile to store them in the constant memory of the device in order to benefit from the constant cache. Similarly, this could be done for the coefficients c_k as well since the access patterns is the same. However, it is not possible to dynamically allocate constant memory on the device and one of the goals was to have a dynamical, error-given approximation of the sign-function in which the degree of the polynomial is not predetermined. This is not a problem on the other hand as we can exploit texture memory which gives the same caching behaviour for this access pattern.

The actual evaluation code is a multistep algorithm. The first step includes the initialization of the Clenshaw recurrence, then at each subsequent step we make one iteration for the the auxillary coefficients y_k. The last step consists of the mass term for the Neuberger operator.

As usual the function for the Neuberger operator takes two index parameters `k` and `l` as the source and the result spinor fields and applies the Neuberger operator such that $\psi_l = D_N \psi_k$. For the whole calculation we need three working fields which will be assumed to have been allocated and reserved for the Neuberger operator. Those working fields will be indexed by `iw1`, `iw2` and `iw3` and will have no further meaning in the end except the storage of intermediate results. For the rest of this discussion we assume

```
iw1 = 1; iw2 = 2; iw3 = 3;
```

for simplicity.

In the first step of the algorithm we need to initialize the recurrence formula and implement the first two iterations by hand. For that we calculate $\psi_3 = Q^2 \psi_k$. Then the first intermediate result will be

$$\psi_2 = a\psi_3 + b\psi_k = aQ^2\psi_k + b\psi_k$$

where the coefficients a and b are given by

$$a = \frac{4c_n}{(1-\varepsilon)||Q||^3} \quad \text{and} \quad b = \frac{1}{||Q||}\big(-2c_n\frac{1+\varepsilon}{1-\varepsilon} + c_{n-1}\big).$$

With the definition for z as above, it is straightforward to see that

$$\psi_2 = (2zc_n + c_{n-1})\,\frac{\psi_k}{||Q||}.$$

If we were about to stop the recurrence right here, e.g. in the case $n = 1$ then we would be missing a factor of $1/2$. This comes from the definition of the recurrence and can be introduced in a redefinition of the coefficients $a \to a/2$ and $b \to b/2$. Then we find that for the case $n = 1$

$$\psi_2 = \big(T_1(z)c_1 + T_0(z)c_0\big)\frac{\psi_k}{||Q||}.$$

The next part of the first step completes the initialization for the recurrence and is similar to the first part. Again, we define some coefficients

$$a \equiv r_1 = \frac{4}{||Q||^2(1-\varepsilon)}, \quad b \equiv r_2 = -2\frac{1+\varepsilon}{1-\varepsilon}$$

and

$$c = r_3(c_{n-2} - c_n) = \frac{1}{||Q||}(c_{n-2} - c_n)$$

as well as the input working fields

$$\psi_2 = (2zc_n + c_{n-1})\,\frac{\psi_k}{||Q||} \quad \text{and} \quad \psi_3 = (2zc_n + c_{n-1})\,\frac{Q^2\psi_k}{||Q||}.$$

Then the next intermediate result will be

$$\psi_1 = a\psi_3 + b\psi_2 + c\psi_k$$

which is in the same manner as above equivalent to

$$\psi_1 = \big((4z^2 - 1)c_n + 2zc_{n-1} + c_{n-2}\big)\,\frac{\psi_k}{||Q||}$$

which, for the case $n = 2$, can be written such as

$$\psi_1 = \big(T_2(z)c_2 + T_1(z)c_1 + T_0(z)c_0\big)\frac{\psi_k}{||Q||}.$$

For that last result, we have to introduce the missing factor $1/2$ again in the definition of the coefficients.

Now that the initial conditions for the Clenshaw recurrence have been set, we can jump to the actual iteration. We step down the polynomial degree and execute one iteration every step $j = n-3, n-4, \ldots, 0$. The coefficients we need in every step are

$$a \equiv r_1, \quad b \equiv r_2 \quad \text{and} \quad c = \frac{c_{n-j}}{||Q||},$$

as well as the working fields

$$\begin{aligned}
\psi_1 &= \big((4z^2-1)c_n + 2zc_{n-1} + c_{n-2}\big)\frac{\psi_2'}{||Q||} \\
\psi_2 &= (2zc_n + c_{n-1})\frac{\psi_1'}{||Q||} \\
\psi_3 &= \big((4z^2-1)c_n + 2zc_{n-1} + c_{n-2}\big)\frac{Q^2\psi_1'}{||Q||}.
\end{aligned}$$

Then the intermediate result for the step j is given by

$$\psi_2 = a\psi_3 + b\psi_1 - \psi_2 + c\psi_k.$$

Here, ψ_i' denotes the spinor field from one step earlier. After we have completed one iteration we swap the working fields $\psi_1 \leftrightarrow \psi_2$ and start over again. That way, we *ping-pong* between the two working fields ψ_1 and ψ_2 for the iteration of the Clenshaw recurrence.

For the last iteration of the recurrence at $j = 0$ we need to take into account the factor $1/2$ for our coefficients such as $a = r_1/2$ and $b = r_2/2$. With that we can again put everything together and see that

$$\psi_2 = \big((4z^3-3z)c_n + (2z^2-1)c_{n-1} + zc_{n-2} + c_{n-3}\big)\frac{\psi_k}{||Q||}$$

and again, if if this would be the last iteration for $n = 3$, we can write

$$\psi_2 = \big(T_3(z)c_3 + T_2(z)c_2 + T_1(z)c_1 + T_0(z)c_0\big)\frac{\psi_k}{||Q||}.$$

After the main iteration is complete we apply one last time the Wilson-Dirac operator to the accumulated intermediate result residing in ψ_1 because the approximation of the sign-function involves one explicit application of the Wilson-Dirac operator.

If we have computed any exceptional low-lying eigenmodes of the Wilson-Dirac operator, which will be discussed later on, we can now further replace the sign-function by the precalculated projectors to those eigenstates. The result spinor field for the replaced Neuberger operator is supposed to reside in ψ_3 at the end.

The final step for the Neuberger operator is the calculation of the mass term and multiplications with γ_5. For that, we need two coefficients

$$a = -(1 - s) - m/2$$
$$b = -(1 - s) + m/2$$

and then calculate

$$\psi_l = a\psi_3 + b\gamma_5\psi_k$$

where ψ_l is the resulting spinor field for the application of the Neuberger operator.

4.5 Low-mode projection

The simple addition and multiplication by scalar coefficients are tasks which are *embarrassingly parallel* because the operation can be performed component-wise and there is no exchange of information necessary between threads. For the computation of exceptional low-lying eigenmodes and the projectors to those states we need some more linear algebra functions such as scalar products, norms and the orthogonal projection of spinors.

The goal is to have the whole projection algorithm running entirely on the GPU. This means, we need to find efficient parallel algorithms for those linear algebra tasks.

For the scalar products and norms we can use what is called a *reduction* which is a special case of a so-called *prefix sums* or *scan operation*. Suppose we have an array of N elements

$$[a_0, a_1, \ldots, a_{N-1}] .$$

Then the prefix sum of this array for the binary operator $\oplus$ is defined by the result

$$[a_0, (a_0 \oplus a_1) , \ldots , (a_0 \oplus a_1 \oplus \cdots \oplus a_{N-1})] .$$

For the scalar product of ψ_k and ψ_l we need to calculate

$$(\psi_k, \psi_l) = \sum_{x \in \Lambda_E} (\psi_k)_x (\psi_l^*)_x .$$

We can now do a two-step calculation suitable for parallelized computation to calculate the scalar product of the two spinor fields ψ_l and ψ_k. First, we need to calculate the intermediate spinor field ψ_m such as

$$(\psi_m)_x = (\psi_l)_x (\psi_k^*)_x \quad \forall x \in \Lambda_E .$$

If we now apply a prefix sum with addition for the binary operation, we can obtain the wanted scalar product from the last element in the resulting array. If we have the scalar product of two spinor fields, the norm squared is by the definition of the norm of a vector in a vector space just the scalar product of the spinor with itself, i.e.

$$||\psi|| \equiv (\psi, \psi) .$$

An extensive analysis about an efficient implementation of prefix sums in CUDA can be found in [Har08].

There is one drawback we can not avoid for this kind of calculation. Since the host program controls the algorithm for the minimization of the Ritz functional, we need to

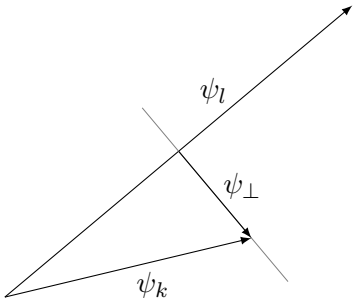

Figure 4.3: The spinor fields can be considered as vectors in a vector space V. Then $\psi_\perp$ is the projection of ψ_k onto the orthogonal subspace to ψ_l.

get the information about the residuals and, consequently, about the scalar products and norms of spinor fields to the host program. This result, which is mostly a complex number, needs to be transfered via the PCI bus. Because the amount of data is small, the main penalty in performance is the PCI latency when we copy the result of the scalar product or norm operation back to the host. In the future, we could benefit from a recent feature of the CUDA toolkit, called *mapped memory*. It is possible to map page-locked host memory into the device's address space so it can be accessed from kernels directly. This allows us to avoid an explicit memory transfer and leave the communication entirely to the driver, which should result in a better utilization of the PCI transfer speed and latency hiding. However, I was not able to test mapped memory at the time of writing but I expect a gain in performance with this feature in further code optimizations.

With the parallel implementation of the scalar product, the orthogonal projection of ψ_k onto the spinor ψ_l

$$\psi_\perp = \psi_k - \psi_l \left(\psi_k, \psi_l\right)$$

can again be implemented embarrassingly parallel and performed component-wise.

With the relevant parts working entirely on the GPU, the algorithm strictly follows the original implementation by Lüscher. Because the algorithm was designed with 64-bit precision in mind and on the GPU only 32-bit precision is available at full speed, we need to tune the convergence parameters of the algorithm.

Chapter 5

Results

As noted earlier, the results for the implementation of the Wilson-Dirac operator and the Neuberger operater were obtained on a NVIDIA GeForce 8800GT and a NVIDIA GeForce GTX 280. The former GPU was hosted on a system based on an Intel Core2Duo running at 2.13 GHz with 4 GB memory. On that system also the benchmarks for the CPU were run. The core hardware features of both GPUs are listed in Tab. (5.1).

5.1 Correctness

Of course, after all tuning for performance, the programs and algorithms should always give the intended results. To ensure this behaviour, for each important task I wrote a unit check that compares the obtained result from the written code with some predefined result.

There are multiple options what exactly to check for correctness. For the Wilson-Dirac and the Neuberger operator I checked if the operators are Hermitian or γ_5-Hermitian, respectively, and if both operators are gauge invariant. For the Neuberger operator it is necessary to check that the implementation fulfills the Ginsparg-Wilson relation. I also had the opportunity to check my results against the results obtained from the CPU

Device	8800 GT	GTX 280
Memory	512 MB	1024 MB
Memory interface width	256-bit	512-bit
Nr. of cores	112	240
Processor clock	1500 MHz	1296 MHz
Memory clock	900 MHz	1107 MHz
Peak performance	504 GFLOP/s	933 GFLOP/s
Memory bandwidth	57.6 GB/s	141.7 GB/s
Compute capability	1.1	1.3

Table 5.1: Key features of the used hardware. Both GPUs are from NVIDIA's consumer series GeForce and are hosted on systems based on Intel Core2Duo with 4 GB memory.

version of the code. For simpler tasks like linear algebra functions predefined inputs where used and the result could be compared directly.

All checks where performed up to machine precision for 32-bit. Because of different implementation for the IEEE754 standard for floating-point arithmetics we do not obtain the exact result from the CPU and the GPU. However, in all cases, the results for each unit check is correct up to $\mathcal{O}(10^{-7})$ which is the limit of 32-bit floating point calculations.

5.2 Benchmarks

To benchmark the application of the Wilson-Dirac operator, I used five different, randomly initialized input spinor and gauge fields and took the average for 64 applications to each input spinor field. Because of the different input values, I can safely ignore caching effects and the average over several applications helps to avoid outliers. The benchmark is performed for different typical lattice sizes.

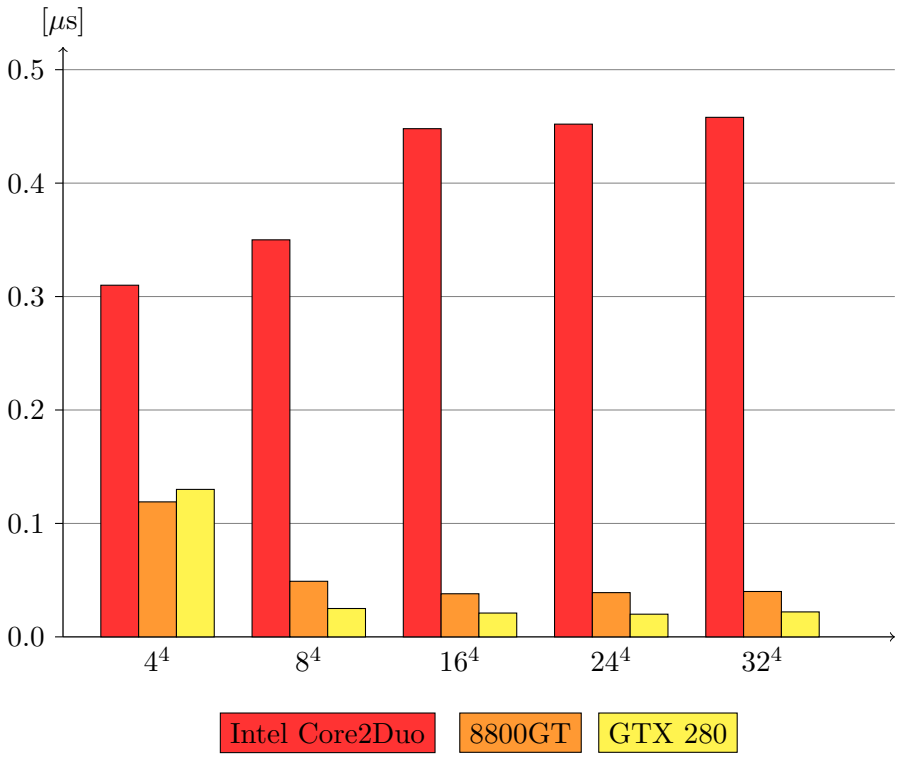

Figure 5.1: The execution runtime per lattice site of the Wilson-Dirac operator for different lattice volumes. Compared to the CPU implementation, the GPU implementation around 12 times faster for the NVIDIA 8800GT and around 22 time faster for the NVIDIA GTX 280. One noteworthy details is, that there is almost no volume dependence for the GPU implementation.

The execution time measured is normalized to the volume $T \times L^3$ of the lattice to give the execution time per lattice site t_{D} for the Wilson-Dirac operator. The time is in the order of micro seconds and the smaller the value, the faster the execution was performed.

The result for the execution time per lattice site for the Wilson-Dirac operator is visualized in Fig. 5.1 for the original CPU implementation and for two different GPUs, to NVIDIA GeForce 8800GT and the NVIDIA GeForce GTX 280.

From the execution time t_{D} we can deduce the achieved floating point operations per second and, more important, by the sustained memory bandwidth. We have seen, that

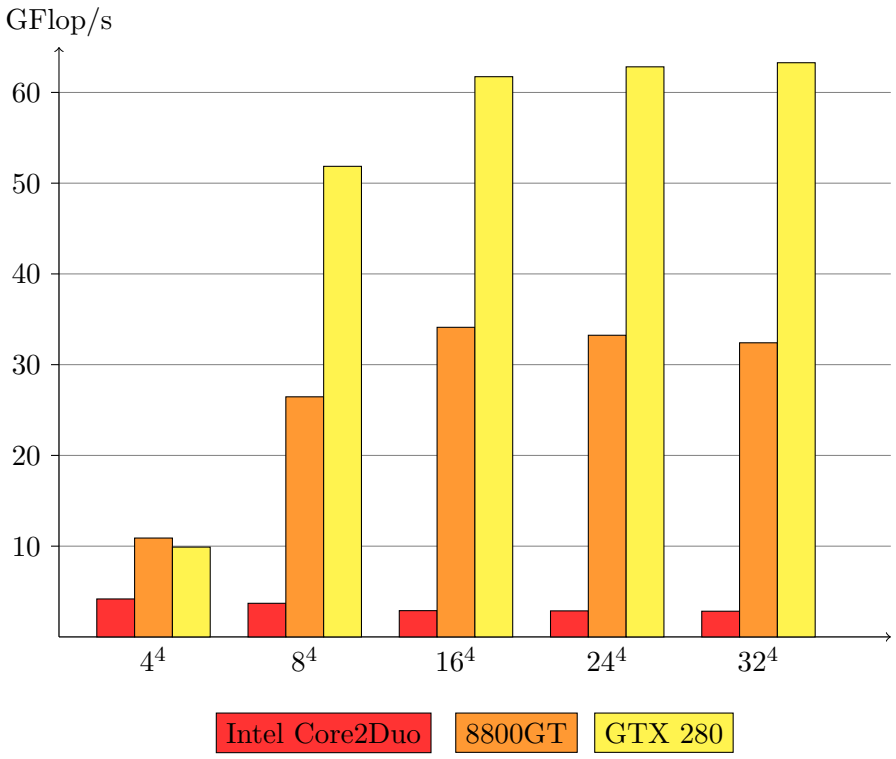

Figure 5.2: Effective floating-point operations per second for the Wilson-Dirac operator. The reconstruction of the gauge fields is not taken into account as the work necessary is not mandatory.

we perform 1392 Flop at each lattice site for the Wilson-Dirac operator and we need to access 1440 byte memory for the calculations.

A comparison in the execution time for the two investigated reconstruction schemes for the gauge fields can be seen in Fig. 5.3. I have chosen a lattice size of 16^4 for the benchmarks and again around 250 applications of the operator to eliminate outliers. For the lookup tables, the index scheme as described in Chapter 4.2, I could not measure any significant differences in performance. The texture cache does a very good job when it comes to provide maximum memory bandwidth and at least for the problem investigated here, it was not possible to support the internal algorithms any further.

For the Neuberger operator, I averaged the execution time over 250 applications to avoid outliers. However, I did not use different input spinor fields because with the necessity

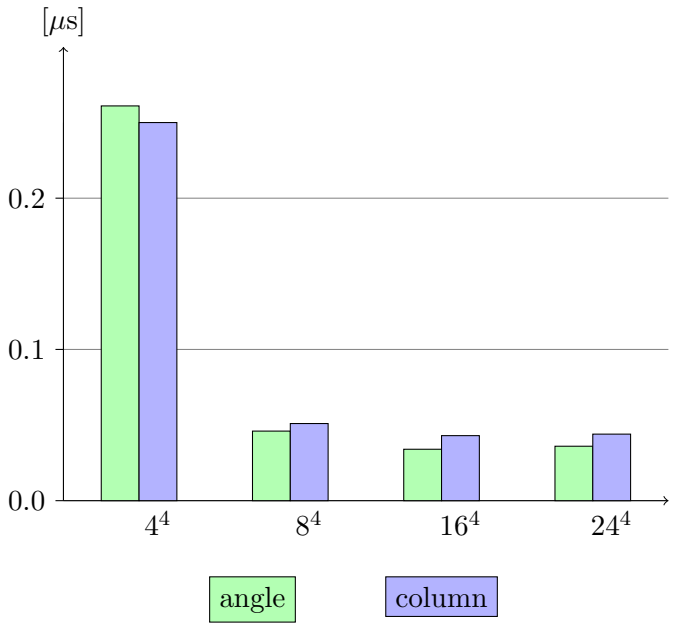

Figure 5.3: Comparison of the two different reconstruction schemes used for the gauge fields. Naïvely, we would expect the angle reconstruction to be faster by a factor of 1.5. However, the maximum gain in performance is around 20% which could be related to the underestimated work necessary to reconstruct the matrix elements with trigonometric functions.

of auxiliary working fields for the calculation of the Neuberger operator the amount of memory on the GPU would not be sufficient to allocate multiple input fields. This is the same argument why I could not take any benchmark results for the 32^4 lattice on the GeForce 8800GT.

The execution time is again normalized to the volume of the lattice as well as the degree of the polynomial approximation.

To benchmark the low-mode projection, I measured the total execution time the algorithm takes to converge. I used 10 different configurations I generated at runtime and calculated the low-lying eigenmodes on those configurations. The execution time was averaged over those ten measurements and normalized to the lattice volume and the number of eigenmodes found. This benchmark was performed on different lattice sizes up to 16^4,

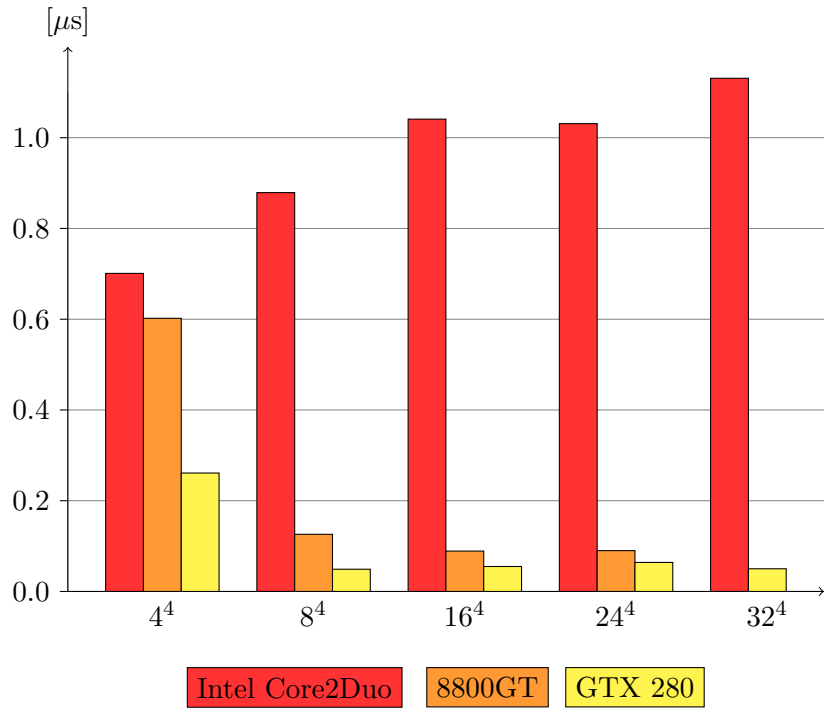

Figure 5.4: Execution runtime of the Neuberger operator without low-modes. On the y-axis the runtime of the evaluation for the Chebychev polynomial approximation of $\text{sign}(Q)\eta$ in the Neuberger operator is shown. The runtime is normalized to the lattice volume and the degree of the approximating polynomial.

the amount of memory on the GPU did not allow to benchmark larger volumes.

In Fig. (5.6) the impact of different optimization methods on the performance of the Wilson-Dirac operator is shown in a symbolic way. The given times have been taken at different stages in development and the changes highlighted do not have to be the only relevant changes in the code.

I have started with the most simple, naïve version of the Wilson-Dirac operator running on the GPU which just parallelizes the outer loop over the lattice sites. The next bar, "alignment", is obtained, if the predefined data structures are given alignment information as described in Chapter 3.4. This step helps the compiler to optimize memory access to global memory. Similar to this is the step labeled "data layout" in which the data layout was changed to allow fully coalesced access to memory. As anticipated, this step gave a

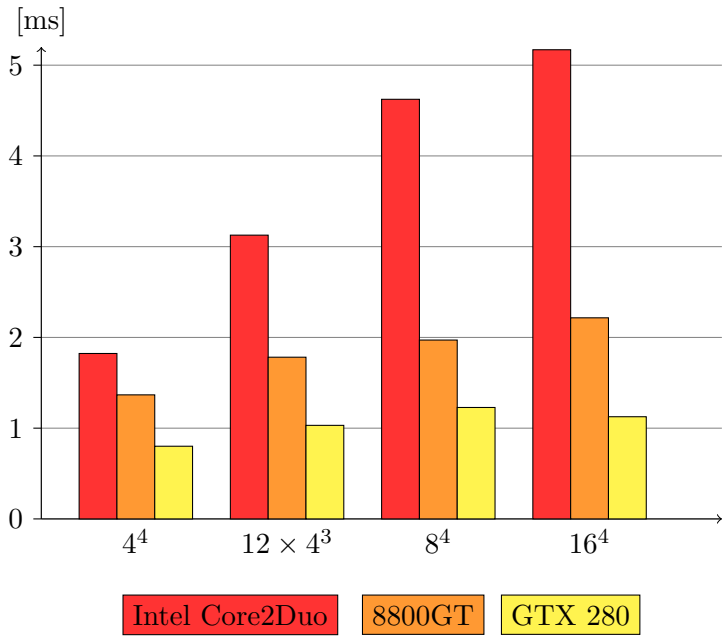

Figure 5.5: Calculation of the low-lying eigenvalues of the Wilson-Dirac operator. On the y-axis it is shown the runtime to calculate the low-lying eigenvalues and eigenmodes of the Wilson-Dirac operator. The time is normalized to the lattice volume and the number of eigenvalues found.

large improvement in performance because the Wilson-Dirac kernel is heavily dependent on memory access and the difference in performance for coalesced and non-coalesced access is huge. The use of textures also gave a significant speedup because the effect of the lookup tables which gave almost random access patterns could be almost completely suppressed. The reconstruction for the gauge fields reduces the total memory loads for the kernel and therefore gives a significant speedup. The bar labeled “miscellaneous” summarizes all other optimizations like unrolling of calculations and the grouping of source code to save registers and so on.

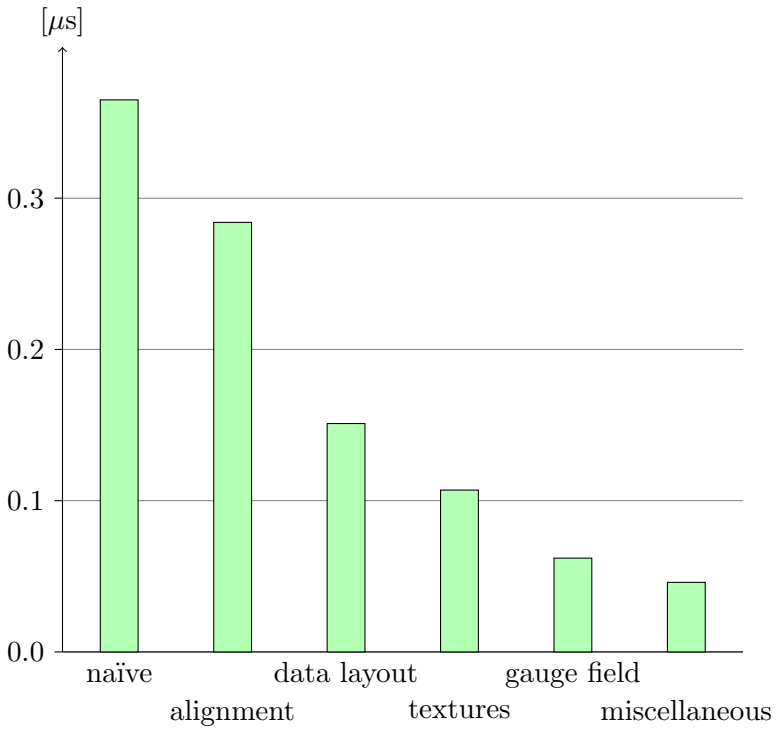

Figure 5.6: A symbolic synopsis of the progress and the impact of different optimization methods on the performance of the Wilson-Dirac operator. The lattice size was set to 16^4. For a detailed explanation see the text.

5.3 Conclusions and outlook

In this work I have shown an efficient implementation of the Wilson-Dirac operator on graphics processing units. I also integrated this into the Neuberger operator and implemented an algorithm to calculate the low-lying eigenvalues of the Wilson-Dirac operator.

We can see, the GPU implementation compared to the CPU is around the factor 12 faster for the NVIDIA GeForce 8800GT and 22 for the GeForce GTX 280. This speedup is obtained for a wide range of lattice volumes. Prior results[BBB$^+$08] show that this speedup is conserved for even larger lattice volumes. This has to be shown in the future for the implementation shown in this work.

Because of the large ratio of computing power to memory bandwidth of around 10:1 for the current generation GPUs and the problems in lattice QCD which are usually memory bound, the essential thing is to optimize for memory bandwidth. We have seen, that the effort to reconstruct SU(3) matrices at runtime gives promising results although the reconstruction schemes investigated do not give the improvement in performance initially indicated. In the future, optimal reconstruction schemes have to be investigated.

With the implementation of the Neuberger operator it could be shown that the speedup of the Wilson-Dirac operator on the GPU can be conserved in a larger framework. We can see, that the speedup for both the GeForce 8800GT and the GeForce GTX 280 is around the same value than for the Wilson-Dirac operator.

Interestingly, the volume dependence for the Neuberger operator is different for the GPU as the execution time per lattice site and degree of the polynomial approximation slightly reduces for large volumes as opposed to the CPU implementation. If this trend can be confirmed for even larger lattice volumes has to be shown in the future.

For the calculation of the low-lying eigenvalues of the Wilson-Dirac operator, a speedup of around the factor 4 for the GeForce GTX 280 could be obtained which gives a lot of space for further improvements. Especially, a lot more effort has to be put into the optimization of the linear algebra operations. Also, new CUDA features like mapped memory have to be taken into account to reduce ineffective communications from host to device which is the clear bottleneck for the current algorithm.

All in all, problems of lattice QCD calculations can be mapped very efficiently to graphics processing units. Although programs have to be optimized for the hardware and GPUs are not as flexible as the CPU, the heterogeneous interplay between both could be used in the future to improve programs for lattice QCD computations with complex code running on the CPU and code exposing parallelism running on the GPU.

Appendix A

Projection to half-spinors

The components of the projected spinor

$$\phi = (1 - s\gamma_\mu)\psi$$

with $s = \pm 1$, $\mu = 1, \ldots, 4$ can be given explicitly. In the following ψ_k and ϕ_k denote the k^{th} Dirac component of the corresponding spinor.

$s = +1, \quad \mu = 0$	$s = -1, \quad \mu = 0$
$\phi_1 = \psi_1 + \psi_3$	$\phi_1 = \psi_1 - \psi_3$
$\phi_2 = \psi_2 + \psi_4$	$\phi_2 = \psi_2 - \psi_4$
$\phi_3 = \phi_1$	$\phi_3 = -\phi_1$
$\phi_4 = \phi_2$	$\phi_4 = -\phi_2$

$s = +1, \quad \mu = 1$

$\phi_1 = \psi_1 + i\psi_4$

$\phi_2 = \psi_2 + i\psi_3$

$\phi_3 = -i\phi_2$

$\phi_4 = -i\phi_1$

$s = -1, \quad \mu = 1$

$\phi_1 = \psi_1 - i\psi_4$

$\phi_2 = \psi_2 - i\psi_3$

$\phi_3 = i\phi_2$

$\phi_4 = i\phi_1$

$s = +1, \quad \mu = 2$

$\phi_1 = \psi_1 + \psi_4$

$\phi_2 = \psi_2 - \psi_3$

$\phi_3 = -\phi_2$

$\phi_4 = \phi_1$

$s = -1, \quad \mu = 2$

$\phi_1 = \psi_1 - \psi_4$

$\phi_2 = \psi_2 + \psi_3$

$\phi_3 = \phi_2$

$\phi_4 = -\phi_1$

$s = +1, \quad \mu = 3$

$\phi_1 = \psi_1 + i\psi_3$

$\phi_2 = \psi_2 - i\psi_4$

$\phi_3 = -i\phi_1$

$\phi_4 = i\phi_2$

$s = -1, \quad \mu = 3$

$\phi_1 = \psi_1 - i\psi_3$

$\phi_2 = \psi_2 + i\psi_4$

$\phi_3 = i\phi_1$

$\phi_4 = -i\phi_2$

Appendix B

Matrix elements in the angle representation

For the representation of the gauge field through 8 angles, the matrix elements to be reconstructed for the whole SU(3)-matrix can be given explicitly [Bro88].

With the paremeters $0 \leq \theta_1, \theta_2, \theta_3 \leq \pi/2$, $0 \leq \phi_1, \ldots, \phi_5 \leq 2\pi$ the explicit form of the matrix elements u_{ij} is given such as

$$
\begin{aligned}
u_{11} &= \cos\theta_1 \cos\theta_2 \, \mathrm{e}^{i\phi_1}, \\
u_{12} &= \sin\theta_1 \, \mathrm{e}^{i\phi_3}, \\
u_{13} &= \cos\theta_1 \sin\theta_2 \, \mathrm{e}^{i\phi_4}, \\
u_{21} &= \sin\theta_2 \sin\theta_3 \, \mathrm{e}^{-i\phi_4 - i\phi_5} - \sin\theta_1 \cos\theta_2 \cos\theta_3 \, \mathrm{e}^{i\phi_1 + i\phi_2 - i\phi_3}, \\
u_{22} &= \cos\theta_1 \cos\theta_3 \, \mathrm{e}^{i\phi_2}, \\
u_{23} &= -\cos\theta_2 \sin\theta_3 \, \mathrm{e}^{-i\phi_1 - i\phi_5} - \sin\theta_1 \sin\theta_2 \cos\theta_3 \, \mathrm{e}^{i\phi_2 - i\phi_3 + i\phi_4}, \\
u_{31} &= -\sin\theta_1 \cos\theta_2 \sin\theta_3 \, \mathrm{e}^{i\phi_1 - i\phi_3 + i\phi_5} - \sin\theta_2 \cos\theta_3 \, \mathrm{e}^{i\phi_2 - i\phi_4}, \\
u_{32} &= \cos\theta_1 \sin\theta_3 \, \mathrm{e}^{i\phi_5}, \\
u_{33} &= \cos\theta_2 \cos\theta_3 \, \mathrm{e}^{i\phi_1 - i\phi_2} - \sin\theta_1 \sin\theta_2 \sin\theta_3 \, \mathrm{e}^{-i\phi_3 + i\phi_4 + i\phi_5}.
\end{aligned}
$$

Appendix C

Compute capabilities

NVIDIA cards are divided into groups with the same hardware properties called *compute capabilities*. Cards with the same compute capabilities do have the same key properties. This methods makes it easy for programmers to optimize for certain hardware generations.

Specifications for compute capability 1.0

- The maximum number of threads per block is 512;
- The maximum size of each dimension of a grid of thread blocks is 65535;
- The number of registers per multiprocessor is 8192;
- The amount of shared memory per multiprocessor is 16KB;
- The total amount of constant memory is 64KB;
- The cache size for constant memory is 8KB per multiprocessor;
- The cache size for texture memory is between 6KB and 8KB per multiprocessor;
- The maximum number of active threads per multiprocessor is 768.

Specifications for compute capability 1.2

- The number of registers per multiprocessor is 16384;
- The maximum number of active threads per multiprocessor is 1024.

Specifications for compute capability 1.3

- Support for double-precision floating-point computation.

Compute capability 1.1 does introduce atomic functions on 32-bit words in global memory which where not relevant for this work here.

A complete definition of the compute capabilities and further hardware specifications can be found in [NVI09].

Appendix D

Glossary

A short overview of the technical terms occurred is given in this chapter. For a deeper explanation the reader should refer to [OSW+05].

Bump mapping Bump mapping is a technique to give the illusion of irregularities of the surface of an object without increasing the complexity of the object itself. Therefore, the informations of normal vectors for the light calculation will be stored in a texture mapped to this object. Surface elevation information is stored in a so-called *heightmap* and can be used to generate realistic shadows.

Lighting Lighting calculations are also called shading and are based on simplified models for the local illumination of objects in a scene. The most basic model is called *Phong shading* and includes three parts: ambient shading calculates the ambient reflection of all points in the scene, diffuse shading is the amount of diffuse or Lambertian reflection from incoming light such as from lights inside the scene and specular shading calculates the specular reflection which are small intense specular highlights of objects.

Projection matrix The projection matrix gives the coordinate transformation from the scene to the rendering plane. The three-dimensional volume of the displayed

scene will be mapped to the two-dimensional rendering plane. Depth information will be stored for further processing in special buffers.

Rendering The rendering process translates the three-dimensional data in the scene into a two-dimensional image. The rendering process will be implemented in a pipelined fashion and usually contains modeling transformations, lighting, projection to the rendering plane, clipping of unseen parts of the scene and display of the result.

Rendering plane The rendering plane is a two-dimensional region and the target of the rendering process. Usually, the rendering plane is a part of the screen itself but it can also be some two-dimensional memory region for off-screen rendering.

RGBA RGBA stands for **R**ed **G**reen **B**lue **Alpha** and is the usual format to specify colors in computer graphics. The red, green and blue component gives the ratio of the corresponding color, the alpha component contains information about the translucency of the total color and makes it possible to define transparent object.

Scene A scene is the collection of virtual objects to be visualized together with lights for illumination and the camera which represents the viewpoint. The scene usually contains the corresponding parameters of the objects presented like color, material, surface, type of lights and camera.

Texture Textures are one-, two- or three-dimensional images objects in the scene are overlaid with to enhance the degree of details without increasing the complexity of the scene.

Texture blending Modern rendering pipelines can handle and combine several textures at once to give an object different material properties. This is called texture blending. A colored wall of bricks is a good example where texture blending could be used. Different textures will be used for the structure of the bricks itself, for the global coloring and for created shadows with bump mapping for example. On a basic level, textures will be pixel-wise combined with a binary operator like addition or multiplication to achieve various effects.

Bibliography

[A+00] S. Aoki et al. Quenched Light Hadron Spectrum. *Phys. Rev. Lett.*, 84:238–241, 2000.

[B+98] Claude W. Bernard et al. Continuum limit of lattice QCD with staggered quarks in the quenched approximation: A critical role for the chiral extrapolation. *Phys. Rev. Lett.*, 81:3087–3090, 1998.

[BBB+08] Kipton Barros, Ronald Babich, Richard Brower, Michael A. Clark, and Claudio Rebbi. Blasting through lattice calculations using CUDA. *PoS*, LATTICE2008:045, 2008.

[Bro88] J. B. Bronzan. Parametrization of SU(3). *Phys. Rev.*, D38:1994, 1988.

[BS91] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik, 25.* 1991.

[BSK76] T. Banks, Leonard Susskind, and John Kogut. Strong-coupling calculations of lattice gauge theories: (1 + 1)-dimensional exercises. *Phys. Rev. D*, 13(4):1043–1053, Feb 1976.

[E+07] Gyozo I. Egri et al. Lattice QCD as a video game. *Comput. Phys. Commun.*, 177:631–639, 2007.

[Fro03] Andreas Frommer. *Skript zur Vorlesung "Iterationsverfahren"*, 2003.

[FS95] Vadim Furman and Yigal Shamir. Axial symmetries in lattice QCD with Kaplan fermions. *Nucl. Phys.*, B439:54–78, 1995.

[GHLW03] Leonardo Giusti, C. Hoelbling, M. Lüscher, and H. Wittig. Numerical techniques for lattice QCD in the epsilon- regime. *Comput. Phys. Commun.*, 153:31–51, 2003.

[Gre97] Anne Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[GSZ90] K. Gallivan, A. Sameh, and Z. Zlatev. Solving general sparse linear systems using conjugate gradient-type methods. In *ICS '90: Proceedings of the 4th international conference on Supercomputing*, pages 132–139, New York, NY, USA, 1990. ACM.

[GW73] D. J. Gross and Frank Wilczek. Ultraviolet Behavior of Non-Abelian Gauge Theories. *Phys. Rev. Lett.*, 30:1343–1346, 1973.

[GW82] Paul H. Ginsparg and Kenneth G. Wilson. A remnant of chiral symmetry on the lattice. *Phys. Rev. D*, 25(10):2649–2657, May 1982.

[Har08] Mark Harris. *Parallel prefix sum (scan) with CUDA*. NVIDIA Corporation, January 2008.

[HJL99] Pilar Hernández, Karl Jansen, and Martin Lüscher. Locality properties of Neuberger's lattice Dirac operator. *Nucl. Phys.*, B552:363–378, 1999.

[IBP08] Khaled Z. Ibrahim, François Bodin, and Olivier Pène. Fine-grained parallelization of lattice QCD kernel routine on GPUs. *J. Parallel Distrib. Comput.*, 68(10):1350–1359, 2008.

[Kap92] David B. Kaplan. A Method for simulating chiral fermions on the lattice. *Phys. Lett.*, B288:342–347, 1992.

[KS75] John Kogut and Leonard Susskind. Hamiltonian formulation of Wilson's lattice gauge theories. *Phys. Rev. D*, 11(2):395–408, Jan 1975.

[KS96] Thomas Kalkreuter and Hubert Simma. An Accelerated conjugate gradient algorithm to compute low lying eigenvalues: A Study for the Dirac operator in SU(2) lattice QCD. *Comput. Phys. Commun.*, 93:33–47, 1996.

[Lüs98] Martin Lüscher. Exact chiral symmetry on the lattice and the Ginsparg-Wilson relation. *Phys. Lett.*, B428:342–345, 1998.

[MM94] I. Montvay and G. Münster. *Quantum fields on a lattice*. Cambridge Univ. Press, 1994.

[Neu98a] Herbert Neuberger. Exactly massless quarks on the lattice. *Phys. Lett.*, B417:141–144, 1998.

[Neu98b] Herbert Neuberger. More about exactly massless quarks on the lattice. *Phys. Lett.*, B427:353–355, 1998.

[NN81] Holger B. Nielsen and M. Ninomiya. No Go Theorem for regularizing chiral fermions. *Phys. Lett.*, B105:219, 1981.

[NVI09] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 2009.

[OSW+05] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[P7585] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985.

[Pol73] H. David Politzer. Reliable Perturbative Results for Strong Interactions? *Phys. Rev. Lett.*, 30(26):1346–1349, Jun 1973.

[PTVF92] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

[She94] Jonathan R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. 1994.

[SW85] B. Sheikholeslami and R. Wohlert. Improved Continuum Limit Lattice Action for QCD with Wilson Fermions. *Nucl. Phys.*, B259:572, 1985.

[Sym83] K. Symanzik. Continuum Limit and Improved Action in Lattice Theories. 2. O(N) Nonlinear Sigma Model in Perturbation Theory. *Nucl. Phys.*, B226:205, 1983.

[Wil74] Kenneth G. Wilson. Confinement of quarks. *Phys. Rev. D*, 10(8):2445–2459, Oct 1974.

[Wit08] Hartmut Wittig. QCD on the lattice. 2008. In *Landolt-Boernstein I 21A: Elementary particles* 5.