MASTER IN HIGH PERFORMANCE COMPUTING

Lattice Quantum Chromodynamics on Intel® Xeon Phi™based supercomputers

Supervisor:
PROF. DR. CARSTEN URBACH

Candidate:
Peter LABUS

2nd EDITION
2015–2016

Master's Programme in High Performance Computing, ICTP & SISSA Trieste, Italy.

`https://github.com/plabus/qphix`
`https://github.com/plabus/qphix-codegen`

The research for this master's thesis project was carried out under the supervision of Prof. Dr. Carsten Urbach with the financial support of the SISSA Research Fellowship "Lattice QCD calculations of Hadronic properties on Intel-MIC based supercomputers", within a total of seven months, from June to December 2016.

December 2016

Preface

The aim of this master's thesis project was to expand the QPhiX library for twisted-mass fermions with and without clover term. To this end, I continued work initiated by Mario Schröck et al. [63].

In writing this thesis, I was following two main goals. Firstly, I wanted to stress the intricate interplay of the four pillars of High Performance Computing: Algorithms, Hardware, Software and Performance Evaluation. Surely, algorithmic development is utterly important in Scientific Computing, in particular in LQCD, where it even outweighed the improvements made in Hardware architecture in the last decade—cf. the section about computational costs of LQCD. It is strongly influenced by the available hardware—think of the advent of parallel algorithms—but in turn also influenced the design of hardware itself. The IBM BlueGene series is only one of many examples in LQCD. Furthermore, there will be no benefit from the best algorithms, when one cannot implement the ideas into correct, performant, user-friendly, read- and maintainable (sometimes over several decades) software code. But again, truly outstanding HPC software cannot be written without a profound knowledge of its target hardware. Lastly, an HPC software architect and computational scientist has to be able to evaluate and benchmark the performance of a software program, in the often very heterogeneous environment of supercomputers with multiple software and hardware layers.

My second goal in writing this thesis was to produce a self-contained introduction into the computational aspects of LQCD and in particular, to the features of QPhiX, so the reader would be able to compile, read and understand the code of one truly amazing pearl of HPC [40].

It is a pleasure to thank S. Cozzini, R. Frezzotti, E. Gregory, B. Joó, B. Kostrzewa, S. Krieg, T. Luu, G. Martinelli, R. Percacci, S. Simula, M. Ueding, C. Urbach, M. Werner, the Intel company for providing me with a copy of [55], and the Jülich Supercomputing Center for granting me access to their KNL test cluster DEEP [8].

Peter Labus, December 2016.

Contents

Introduction

Lattice Quantum Chromodynamics (LQCD) is the discretised version of the physical theory that is believed to described the so-called strong interactions—one of the four known elementary forces in the Universe. It describes the movement and interaction (the *dynamics*) of elementary particles called quarks. Their charge is called colour (Greek *chromos*) and determines how they combine into heavier particles like protons and neutrons which then themselves form the nucleus of the atom. The particles that are responsible for this sort of binding mechanism are called gluons because they stick or *glue* the quarks together. Interestingly enough, these particles themselves are charged and interact not only with the quarks but also amongst each other. This has the consequence that the equations describing the dynamics are non-linear and the theory cannot be easily solved with analytic methods. However, discretising the underlying continuous space-time structure and looking only onto a finite subset of points provides an excellent framework to study the theory on the computer. The discrete version of space-time assumes the form of a grid or *lattice*—which explains the origin of the name of the theory.

Mathematically, QCD is fully described by an integral (the so-called path integral or partition function) which can be manipulated to extract information from the theory and calculate observable quantities which could then be compared with data from experiments. Numerical methods to estimate integrals have a long history. Standard methods turn out to be particularly expensive when the dimensionality of the domain of integration is high. For these kind of integrals a sophisticated but simple technique which is based on pure randomness turns out to be most efficient. It is known under the name of Monte Carlo Integration. In essence, one lets the computer roll a – very clever – dice many times to select the portions of relevance of the integral and then sums only those. This can be shown to be a sufficiently fast converging estimate of the full integral.

In LQCD the preferred variant of this method is called *Hybrid* or *Hamiltonian Monte Carlo* (HMC) algorithm [21]. It combines two of the most influential ideas in Scientific Computing of the 20th century—the *Metropolis Monte Carlo* algorithm [50] and *Molecular Dynamics* [14]. The word Hamiltonian thereby refers to the specific system of equations that is used for the Molecular Dynamics (MD) Integration. MD is the procedure of integrating a system of differential equations (i.e. *equations of motion*) by means of discretising the differential operators, which transforms the equations into simple algebraic ones. We will come back to the HMC algorithm in more detail in the next chapter.

The equations of motions which are integrated during the MD part of the Monte Carlo algorithm describe how the quarks move or *propagate* in space-time under the influence of the gluons. As it turns out, this is described by the inverse of an intricate differential operator (the Dirac operator or *dslash*) in the continuum theory, which however becomes a very sparse matrix in the lattice formulation. It is described by a (discretisation dependent) nearest-neighbour stencil operator which we will present shortly. The appearance of this matrix is the reason why one has to solve very many sparse systems of linear equations during the MD integration. This is best done using iterative solvers to which we will give an introduction in the next chapter as well.

Lattice QCD is one of the most computationally expensive fields in Scientific Computing and uses large fractions of the available supercomputer resources worldwide. Walltimes for the generation of Monte Carlo configurations can easily amount for up to a year on large parts of capability supercomputers. In particular, several new important results in hadron physics are expected to have demands of hundreds of Teraflop-years or even Petaflop-years [61]. This is why LQCD software is frequently implemented very early on, on new available hardware and highly optimised and efficient implementations of the best known algorithms are of utmost importance. After introducing the main algorithms used in practical LQCD simulations in the next chapter, we will discuss the scaling of the simulation cost in terms of fundamental physical parameters of the theory.

In this chapter, we want to give a brief overview over the mathematical framework and structure of the continuous quantum field theory of QCD. First and foremost, this serves as a means to familiarise the reader with the objects, entities and notions of the mathematical theory, so the structure of the discretised theory becomes clearer. Then, we want to give a slightly more detailed outline how one actually makes the transition to the lattice formulation of QCD in order to simulate the theory on a computer. This should motivate the basic procedure one will have to apply to extract usable data from the lattice theory and in particular point out ambiguities one has to face when discretising the continuum theory. Along the way, we will introduce the objects that eventually will have to be represented through data structures in the software implementation.

The theory of QCD is most compactly and conveniently represented in form of a partition function[1]

$$\mathcal{Z} = \int \mathcal{D}\psi\, \mathcal{D}\bar{\psi}\, \mathcal{D}A\; e^{-S_{QCD}[\psi,\bar{\psi},A]} \tag{1}$$

which is a *Feynman path integral* over all possible configurations of the fermion fields $\psi(x)$ and the gauge fields $A(x)$ which are both functions of the space-time point x themselves. The former will represent the aforementioned quarks and the latter the gluons. As for now, space-time is a four-dimensional, continuous manifold and thus the integral is formally infinite dimensional. This will change, once the uncountable many points are reduced to finitely many when passing to a lattice with finite extension.

[1]We will directly work in Euclidean signature where space-time is the vector space $\mathbb{R}^4$ with Euclidean inner product.

The name partition function was borrowed from statistical physics and in fact we will make use of a probabilistic interpretation of the path integral in order to calculate $\mathcal{Z}$. The most convenient feature of the partition function is the fact that it can generate observable quantities of the theory in a very elegant way. We will not outline the exact procedure here, but only state that all physical observable are encoded within (vacuum) expectation values of so-called n-point functions of the form

$$\langle \mathcal{O}_1 \mathcal{O}_2 \ldots \mathcal{O}_n \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}\psi \, \mathcal{D}\bar{\psi} \, \mathcal{D}A \;\; \mathcal{O}_1 \mathcal{O}_2 \ldots \mathcal{O}_n[\psi, \bar{\psi}, A] \; e^{-S_{QCD}[\psi,\bar{\psi},A]} \,, \tag{2}$$

where $\mathcal{O}_1 \mathcal{O}_2 \ldots \mathcal{O}_n$ are products of operators of the quantum field theory. The important point to note, is the fact that all of these integrals have a very similar form and will in fact be evaluated in a very similar way.

The actual information of the theory is encoded within the *action* S_{QCD} which is also the weight in the *Boltzmann factor* $e^{-S_{QCD}}$ appearing in both the partition as well as the n-point functions. In continuum's QCD it reads

$$S_{QCD} = S_F[\psi, \bar{\psi}, A] + S_G[A] \,, \tag{3}$$

$$S_F = \sum_{f=1}^{N_F} \int \mathrm{d}^4 x \left[\overline{\psi}^{(f)}(x) \left(\not{D}(x) + m^{(f)} \right) \psi^{(f)}(x) \right] , \tag{4}$$

$$S_G = \frac{1}{2g^2} \int \mathrm{d}^4 x \; \mathrm{Tr}\, F^2(x) \,, \tag{5}$$

where the first part S_F is the fermionic and the second S_G is the (gauge) bosonic contribution. Here the affine differential operator $\not{D} = \sum_{\mu=1}^{4} \gamma_\mu (\partial_\mu + i A_\mu)$ is the Dirac *dslash* operator mentioned earlier.[2] The four 4×4 matrices γ_μ are the gamma-matrices which arise from the property that fermions have spin $\frac{1}{2}$. We will seem them later explicitly. The first part of the *dslash* operator (the partial space-time derivative) describes the propagation of the fermions in space-time, whilst the second term encodes the interaction of the quarks with the gluons. Finally, $m^{(f)}$ is the mass of a quark of *flavour* f. The index f simply enumerates through the N_F different types of quarks, of which six are known experimentally. In LQCD, one frequently simulates a number of flavours varying from two to four.

The above notation is very compact and hides a lot of the information in order to be more readable. Let us have a closer look at the components of the fermionic and bosonic fields first. The quarks are described by so-called Dirac *four-spinor fields*

$$\overline{\psi}^{(f)}{}_c^{\alpha}(x) \;\; \text{and} \;\; \psi^{(f)}{}_c^{\alpha}(x) \,, \tag{6}$$

which are defined for every flavour f at every space-time point x. In addition, they carry two vector-like indices, the Dirac or *spin* index $\alpha = 1, 2, 3, 4$ and the *colour* index $c = 1, 2, 3$. Each of the 12 components for each space-time point is a complex number. The bar denotes Dirac conjugation $\overline{\psi} = \psi^\dagger \gamma_0$. In the following we will often suppress these indices and adapt a matrix/vector notation.

The gluons on the other hand are described by the gauge fields

$$A_\mu^{cd}(x) \,, \tag{7}$$

[2] As it is customary in the continuum theory, we will count one-based here, and pass to zero-based counting in the lattice theory, since this way of counting is used more frequently in computer science.

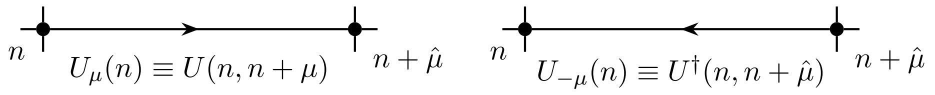
n
$U_\mu(n) \equiv U(n, n+\mu)$
$n+\hat{\mu}$
n
$U_{-\mu}(n) \equiv U^\dagger(n, n+\hat{\mu})$
$n+\hat{\mu}$

Figure 1: Forward (left) and backward (right) oriented link variables, which replace the gauge fields on the lattice, adapted from [35].

which again, are define at every space-time point x. They carry a Lorentz index $\mu = 1, 2, 3, 4$, and in addition two colour indices c and d. In fact, they are traceless, hermitian 3×3 matrices in colour space and thus can be parametrised by eight real numbers. Choosing a basis $T_i^{cd} \in \mathfrak{su}(3)$, where $i = 1, 2, \dots, 8$, for these matrices, the gauge fields can be parametrised by the eight real colour components $A_\mu^{(i)}$ as follows

$$A_\mu^{cd}(x) = \sum_{i=1}^{8} \sum_{b=1}^{3} A_\mu^{(i)\,cb}\, T_i^{bd} \,. \tag{8}$$

With all indices written out the fermionic part of the action of QCD assumes the following form

$$S_F = \sum_{f=1}^{N_F} \int \mathrm{d}^4x \sum_{\mu=1}^{4} \sum_{\alpha,\beta=1}^{4} \sum_{c,d=1}^{3} \left\{ \overline{\psi}^{(f)\,\alpha}_{\;\;c}(x) \left[(\gamma_\mu)^{\alpha\beta} (\partial^\mu \delta_{cd} + i A^\mu_{cd}) + m^{(f)} \delta_{\alpha\beta} \delta_{cd} \right] \psi^{(f)\,\alpha}_{\;\;c}(x) \right\} . \tag{9}$$

Similarly, the bosonic part, that encodes the propagation and self-interactions of the gluons assumes the form

$$S_G = \frac{1}{4g^2} \int \mathrm{d}^4x \, \sum_{i=1}^{8} F_{\mu\nu}^{(i)}(x)\, F_{\mu\nu}^{(i)}(x) \,, \tag{10}$$

$$F_{\mu\nu}^{(i)}(x) = \partial_\mu A_\nu^{(i)}(x) - \partial_\nu A_\mu^{(i)}(x) - \sum_{j=1}^{3} \sum_{k=1}^{3} f_{ijk}\, A_\mu^{(j)}(x)\, A_\nu^{(k)}(x) \,, \tag{11}$$

with real coefficients (the structure constants of the Lie algebra) f_{ijk}. Again, the presentation up to now only serves the purpose to introduce to the quantities and structures of interest for later use in the lattice theory and its implementation on the computer.

In order to proceed with the discretisation of the continuum theory, we essentially have to follow a two-step programme. First the space-time continuum has to be replaced with a discrete four-dimensional grid with a lattice spacing a separating two neighbouring points in one direction. Secondly, we have to make sense of the action and integrals on this grid as well. To discretise the action itself, two important requirements have to be met. On the one-hand side, we have to guarantee that in the limit $a \to 0$, that is the *naïve continuum limit*, the lattice action approaches the continuum action smoothly. On the other hand, we need to insure that a certain invariance of the continuum's formulation, the so-called *gauge symmetry*, is preserved, which allows to model interactions with a *local* quantum field theory, paying a depth in terms of introducing a redundancy into the theory. We shall not be concerned with the mathematical details of gauge field theories here. The interested reader may find further details in any book about quantum field theory (cf. *e.g.* [79]).

The first step is straight-forward. We introduce a lattice as the collection of points

$$\Lambda = \{n = (n_0, n_1, n_2, n_3) \,|\, n_i = 0, 1, \dots, N_i - 1\} \;, \tag{12}$$

such that each point of the lattice is labelled by a quadruple of integers and $x_\mu = an_\mu$ for all space-time dimensions $\mu = 0, 1, 2, 3$. Then integrals are replaced with sums over the sites of the lattice $\int \mathrm{d}^4x \to a^4 \sum_{n\in\Lambda}$ and the fermion and gauge fields are defined only on sites n of the lattice $\psi(n)$ and $A_\mu(n)$. Note that they still carry the same colour, spin and flavour indices as in the continuum's formulation, which we suppress here for simplicity.

It turns out, that the gauge part of the action is best discretised by exponentiating the gauge fields, and using the so-called *link variables* as the fundamental variables

$$U_\mu(n) = \exp\left(iaA_\mu(n)\right) \in SU(3) \,, \tag{13}$$

thus promoting an $\mathfrak{su}(3)$ Lie algebra-valued field into an $SU(3)$ Lie group-valued field. The link variables $U_\mu(n) \equiv U(n, n+\hat{\mu})$ live on connecting links between to lattices sites (thus the name) and carry an orientation (cf. Fig. 1). They are 3×3 special unitary colour matrices. It can be shown, that any closed loop of (oriented) products of link variables of the form

$$L(U) = \mathrm{Tr}\left(\prod_{(n,\mu)\in\mathcal{L}} U_\mu(n)\right) \,, \tag{14}$$

where $\mathcal{L}$ is a loop on the lattice, is gauge-invariant. These objects are called *Wilson loops*, and the simplest of them is the *plaquette*

$$U_{\mu\nu}(n) = U_\mu(n)\, U_\nu(n+\hat{\mu})\, U_{-\mu}(n+\hat{\mu}+\hat{\nu})\, U_{-\nu}(n+\hat{\nu}) \,, \tag{15}$$

depicted in Fig. 2. One can show, that summing over all plaquettes of the lattice, counting only one orientation per plaquette, on can resemble the continuum gauge action in the continuum limit

$$S_G[U] = \frac{2}{g^2} \sum_{n\in\Lambda} \sum_{\mu<\nu} \mathrm{Re}\ \mathrm{Tr}\left[\mathbb{1} - U_{\mu\nu}(n)\right] = \frac{a^4}{2g^2} \sum_{n\in\Lambda} \sum_{\mu,\nu} \mathrm{Tr}\ F_{\mu\nu}^2(n) + \mathcal{O}(a^2) \,. \tag{16}$$

Note however, that this procedure is ambiguous, because we could add any term of $\mathcal{O}(a)$ or higher to the action without spoiling the continuum limit $a \to 0$.

Unfortunately, the naïve, straight-forward discretisation procedure for the fermions is less successful. Here, we first have to discretise the derivative operator, for instance with a symmetric difference operator

$$\partial_\mu \psi(x) \mapsto \frac{1}{2a}\left(\psi(n+\hat{\mu}) - \psi(n-\hat{\mu})\right) . \tag{17}$$

Next, one can expand the link variables $U_\mu(n)$ for small lattice spacings a and replace the lattice gauges fields in the *dslash* operator with link variables. One can show that the resulting term is gauge-invariant and the naïve fermion lattice action becomes:

$$S_F^{\textit{naïve}} = a^4 \sum_{n\in\Lambda} \overline{\psi}(n) \left[\sum_{\mu=1}^{4} \frac{\gamma_\mu}{2a}\Big(U_\nu(n)\,\psi(n+\hat{\mu}) - U_{-\nu}(n)\,\psi(n-\hat{\mu})\Big) + m\,\psi(n)\right] . \tag{18}$$

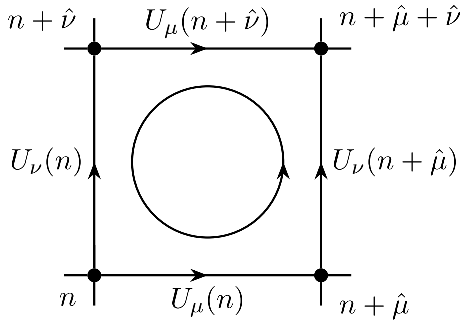

Figure 2: The plaquette is the simplest oriented product of link variables, adapted from [35].

Unfortunately, this action turns out to contain so-called *fermion doublers*, which double the number of fermions for each dimension. It thus describes 16 instead of one fermion. The original solution proposed by Wilson [80] is to add a second-derivative-like term[3]

$$S_W = -\frac{r}{2a} \sum_{n\in\Lambda} \sum_{\mu=1}^{4} \overline{\psi}(n) \Big(\psi(n+\hat{\mu}) - 2\psi(n) + \psi(n-\hat{\mu}) \Big) \,, \tag{19}$$

to the naïve fermion action which effectively introduces an additional mass term for the fermions and removes the doublers. There are, however, other problems with (pure) *Wilson Fermions*, which is why there exist many other fermionic formulations of LQCD, including *Staggered Fermions*, *Overlap Fermions* and *Domain Wall Fermions*.

Wilson-like fermions can also be improved, for instance by adding higher order terms in a, like the *clover term*, as well as other terms like the *twisted-mass term*. We will see both of the above terms in more detail in the next chapter. The goal of this thesis is to implement each of these terms, as well as both in combination into the QPhiX lattice QCD library for Intel Xeon Phi (co-)processors.

More about lattice gauge theories, and LQCD in particular, can be found in many excellent text books, *e.g.* [27, 35, 57].

[3]The *Wilson parameter* r is usually set to 1.

1. Algorithms

In the last chapter we have given an introduction to Quantum Chromodynamics, the Quantum Field Theory of the strong interactions and its discretised version which is known as Lattice QCD. The latter is suitable to study the full theory on a computer and in this chapter we want to outline the most important numerical techniques and algorithms to do so. In particular, we aim to describe all algorithms that are actually used in the QPhiX library.

As we have seen before, in order to extract data from simulating lattice QCD, which could be compared with experiments or studies using analytic techniques, one essentially has to calculate integrals of a specific kind. These integrals are derivatives of the partition function known from statistical physics, called *correlation functions*. They all resemble the same form, in that their integrand is always weighted with what (at least in the case of vanishing temperature and chemical potential) can be interpreted as a probability distribution—the *Boltzmann factor* e^{-S}. This sort of integral is usually best evaluated using *Monte Carlo* techniques, in particular those based on *Markov Chains*, because of the high dimensionality of the integrals. In order to do so, one first generates configurations that are distributed with the given probability and then calculates approximate values of the integrals as the arithmetic mean of the integrated function evaluated on a (large) set of these configurations.

Since there is, as of yet, no efficient Monte Carlo Markov Chain algorithm known, that could function with local updates for the full theory of QCD, one usually uses the *Hybrid Monte Carlo* algorithm to generate gauge configurations. Within this algorithm, (global) updates are proposed using a *Molecular Dynamics* procedure to integrate the classical equations of motion of the theory.

Both in the molecular dynamics part as well as when finally evaluating integrals, it is of great importance to be able to *apply and invert* the full fermion matrix M. In fact, most of the simulation time for lattice QCD is spend in kernels implementing the matrix-vector product of this matrix.

This is why we will first have a detailed look on how to apply the fermion matrix to its respective vector. After that we will give an introduction to a class of solvers for sparse systems of linear equations which can be used to invert the fermion matrix—so called *Krylov Subspace Iterative Solvers*. Finally, we will outline, how both these techniques come in to play when one wants to compute configurations distributed according to the Boltzmann probability, using the Hybrid Monte Carlo algorithm. This understanding will be important, to be able to generate gauge configurations on Xeon Phi's eventually.

In this chapter we will be concerned about algorithms and their theoretical properties. We will not worry about the particulars of an actual implementation, as for instance data structures, memory patterns and the like, until after having introduced the target architectures in the next chapter.

1.1 The Dirac Dslash Operator

After having pointed out in some detail the importance of the Dirac operator both for the generation of gauge configurations as well as to measure quark propagators and other observables we will now outline in some more detail how the matrix can be applied to a spinor field.

First we will describe the basic algorithm in the simplest case of Wilson fermions, already with the addition of some very simple algorithmic improvements. From that we will estimate what kind of theoretical performance we may expect. In particular, we will calculate the *arithmetic intensity* measured in floating point operations per transferred byte in memory. This will show that due to the sparseness of the matrix, the algorithm is actually memory bandwidth rather than compute bound, which would be the case for algorithms handling dense matrices.

Finally we will show how to additionally account for the twisted-mass as well as the clover term which are added to the full fermion matrix M in the action for twisted-mass fermions.

1.1.1 The Basic Algorithm

As mentioned before, the Dirac operator $\not{D}$ is a very large sparse matrix which physically describes both the propagation of the fermions as well as their interaction with the gluons. It is the part of the full fermion matrix

$$M = (4+m) - \frac{1}{2}\not{D}\,. \tag{1.1}$$

The Dirac operator connects even to odd (or red to black) sites and vice versa. This is why it is also referred to as the *hopping matrix*. The *Feynman slash* indicates that the spacetime-derivative ∂_μ in the continuum theory is contracted with the Gamma matrices γ_μ as we will see in more detail shortly.

As we have already seen the last chapter, the Dirac operator acts on fermion (*a.k.a.* quark or spinor) fields $\psi^a_\alpha(x)$ on every point x of the lattice Λ. The additional indices of the spinor field are called *colour* and *spin*. The (Latin) colour index a takes values in $\{0,1,2\}$ and the (Greek) spinor index α assumes the four values $\{0,1,2,3\}$. A spinor (that is a spinor field at one lattice point) may thus alternatively thought of as a collection of four three-component colour-vectors, or three four-component spin-vectors. A spinor has thus $3\times 4 = 12$ complex components and so the Dirac *dslash* operator is a square matrix of $24\times|\Lambda|$ side length. For a cubic lattice of dimensions $48^3\times 96$ the matrix would have roughly (254 million)^2 elements. Stored in its entity in double precision, the matrix would occupy more than 5 PB $= 5\times 10^{17}$ bytes of data.

Fortunately, it is not necessary to store the matrix entirely. Rather, one only needs to store one $SU(3)$ matrix for each link connecting two neighbouring sites of the lattices—the link variables (*a.k.a.* gauge or gluon fields) which have been introduced in the last chapter, as well. A special, unitary matrix $U_{ab} \in SU(3)$ is a 3×3 complex matrix which contains 18 real elements. However, since the matrix has to be unitary $U^\dagger_{ab} = U^{-1}_{ab}$ and special, *i.e.* $\det U_{ab} = +1$, there are actually only 8 real parameters describing the matrix fully. In order to save storage and memory bandwidth when applying the *dslash* operator, one can therefore use techniques to restore a full $SU(3)$ from only 8 or 12 parameters *on the fly*, as we will see below.

With all sums and indices written out explicitly a matrix-vector product of the *dslash* with a spinor assumes the following form

$$\begin{aligned}
\chi^a_\alpha(x) &= \sum_{y\in\Lambda}\sum_{b=0}^{2}\sum_{\beta=0}^{3} \not{D}^{ab}_{\alpha\beta}(x,y)\psi^b_\beta(y) \\
&= \sum_{b=0}^{2}\sum_{\beta=0}^{3}\sum_{\mu=0}^{3} \Big[U^{ab}(x,x+\hat{\mu})\ (\mathbb{1}-\gamma_\mu)_{\alpha\beta}\ \psi^b_\beta(x+\hat{\mu}) + \\
&\qquad U^{\dagger\,ab}(x,x-\hat{\mu})\ (\mathbb{1}+\gamma_\mu)_{\alpha\beta}\ \psi^b_\beta(x-\hat{\mu})\Big]\,,
\end{aligned} \tag{1.2}$$

where x and y are spacetime vectors with four components each, which label the sites of the lattice. The vectors $\hat{\mu} \in \{\hat{0},\hat{1},\hat{2},\hat{3}\} \equiv \{\hat{t},\hat{x},\hat{y},\hat{z}\}$ represent the unit vectors in the respective (positive) direction. Finally, the lattice indices of the matrices U label where the link is emanating and ending, respectively.

There are a few things about this formula worth mentioning. First of all, note that the resulting vector is again a spinor field and thus has 24 real components for every point of the lattice. As we see explicitly in the second equality, the summation over the lattice variable y drops out and is replaced by a summation over the four spacetime (or lattice) dimensions μ. In fact, since the matrix U is unitary, we can think of the matrix $U^\dagger = U^{-1}$ in the third line as the inverse operator which is pointing *backwards* from the point x to the point $x - \hat{\mu}$. We thus see, that the Dirac matrix only connects each site of the lattice with its eight nearest neighbouring sites, as well as the eight links which emanate from that site. In computer science, this sort of operators are called *stencil operators* and are well studied in the general context in the literature [25, 62].

Half Spinors and Projection

The first important algorithmic improvement for the evaluation of the stencil operation comes from the fact that the 4×4 matrices

$$P^{\pm}_\mu = \mathbb{1} \pm \gamma_\mu \tag{1.3}$$

act as projectors onto the spinors and can be evaluated beforehand. Indeed, choosing for instance the *DeGrand-Rossi* basis[1] of the gamma-matrices γ_μ

$$\gamma_0 = \begin{pmatrix} 0 & 0 & 0 & +i \\ 0 & 0 & +i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix} \quad \gamma_1 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & +1 & 0 \\ 0 & +1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} \tag{1.4}$$

$$\gamma_2 = \begin{pmatrix} 0 & 0 & +i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & +i & 0 & 0 \end{pmatrix} \quad \gamma_3 = \begin{pmatrix} 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & +1 \\ +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \end{pmatrix} \tag{1.5}$$

the projected spinors will have a particularly simple form. For instance, in the case of $\mu = 2$ with $\psi = (\psi_0, \psi_1, \psi_2, \psi_3)^T$ we have

$$P_2^+\psi = \begin{pmatrix} 1 & 0 & i & 0 \\ 0 & 1 & 0 & -i \\ -i & 0 & 1 & 0 \\ 0 & i & 0 & 1 \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} \psi_0 + i\psi_2 \\ \psi_1 - i\psi_3 \\ -i\psi_0 + \psi_2 \\ i\psi_1 + \psi_3 \end{pmatrix} \equiv \begin{pmatrix} h_0 \\ h_1 \\ -ih_0 \\ ih_1 \end{pmatrix} , \tag{1.6}$$

where we defined $h_0 = \psi_0 + i\psi_2$ and $h_1 = \psi_1 - i\psi_3$ (we adopted the notation from the QPhiX library [44]). That is, the two top components can be computed with one complex addition only (we will not count multiplications with $\pm i$ as floating point operations, because they merely consist of a swap of the real and imaginary part with a possible change of sign). The lower two components can even be reconstructed from the two upper ones without having to calculate anything at all. This holds true for the seven other cases as well and we have summarised the outcome of *projection* and *reconstruction* for each direction in Tab. 1.1. The physical origin of the simple structure for this kind of bases is the fact that the part of the fermion action that is given by the hopping matrix is chirally invariant. We will refer to the two-component projected (or reconstructed) spinors as *half spinors*. Note in particular, that since the reconstruction is essentially a reordering of the four components of the half spinor (h_0, h_1), one can first multiply the latter with the $SU(3)$ matrix and only then reconstruct the remaining two spinor components (r_2, r_3). This reduces the required colour matrix multiplications from four to two per site and direction.

Gauge Compression

As we will see explicitly below, the basic *dslash* algorithm is memory bandwidth bound. That is, one may want to decrease the amount of data one has to read from and write to memory, even if this comes with a trade-off in form of additional floating point operations.

One way to do this, is to reconstruct or *decompress* the gauge matrices *on the fly*, effectively having to read fewer of their elements from memory. As mentioned before, an $SU(3)$ matrix is fully parametrised by eight real parameters. From these eight parameters the nine complex numbers, which have to be available for the matrix-vector multiplication, can be recalculated.

There are two common methods used for compression. The first one compresses the 18 real numbers into 12 real parameters in the form of two 3-component complex colour vectors [26]. These two vectors $\mathbf{a}$ and $\mathbf{b}$ can be thought of as the first two rows of the matrix, which is why the

[1]The following structure will actually arise for any *chiral* representation of the gamma-matrices (cf. for instance [53]), one other popular choice being the Weyl basis.

$(\mu,\pm)$	h_0	h_1	r_2	r_3
$(0,+)$	$\psi_0+i\psi_3$	$\psi_1+i\psi_2$	$-ih_1$	$-ih_0$
$(1,+)$	$\psi_0-\psi_3$	$\psi_1+\psi_2$	h_1	$-h_0$
$(2,+)$	$\psi_0+i\psi_2$	$\psi_1-i\psi_3$	$-ih_0$	ih_1
$(3,+)$	$\psi_0+\psi_2$	$\psi_1+\psi_3$	h_0	h_1
$(0,-)$	$\psi_0-i\psi_3$	$\psi_1-i\psi_2$	ih_1	ih_0
$(1,-)$	$\psi_0+\psi_3$	$\psi_1-\psi_2$	$-h_1$	h_0
$(2,-)$	$\psi_0-i\psi_2$	$\psi_1+i\psi_3$	ih_0	$-ih_1$
$(3,-)$	$\psi_0-\psi_2$	$\psi_1-\psi_3$	$-h_0$	$-h_1$

Table 1.1: The action of the projection operators $P_\mu^\pm$ defined in the text in the DeGrand-Rossi basis. The lower half spinor (r_2, r_3) can always be reconstructed from the upper one (h_0, h_1).

technique is also called two-row compression. The vectors have to be normalised $||\mathbf{a}|| = ||\mathbf{b}|| = 1$ and orthogonal onto each other $\langle \mathbf{a}, \mathbf{b}\rangle = 0$. The third row can then be restored by calculating the complex conjugate of the cross product

$$\mathbf{c} = (\mathbf{a}\times\mathbf{b})^* . \tag{1.7}$$

Obviously, $\mathbf{c}$ will then be normalised as well and the triple $U \equiv (\mathbf{a}, \mathbf{b}, \mathbf{c})^T$ will be positively oriented (i.e. right-handed), such that $\det U = +1$. Since furthermore $\mathbf{c}$ is orthogonal on both $\mathbf{a}$ and $\mathbf{b}$ and complex conjugate to them, it follows that U is also unitary.[2]

For the first component $(\mathbf{a}\times\mathbf{b})_0^* = \mathbf{a}_1^*\mathbf{b}_2^* - \mathbf{a}_2^*\mathbf{b}_1^*$, two complex multiplications and one complex addition, that is eight floating point operations, are required. Thus, in total this method introduces an additional 24 floating point operations, saving six reads at the same time.

To decompress $SU(3)$ matrices from only eight parameters, one could use the generators of the Lie algebra $\mathfrak{su}(3)$ (the Gell-Mann matrices) and exponentiate, but this turns out to require rather many floating point operations. Another approach was proposed in [19] and modified in [22] and involves the use of trigonometric functions.

Starting again with a complex, normalised three-vector $\mathbf{a}$, one can constructed an orthonormal basis for $\mathbb{C}^3$ setting

$$\mathbf{b}' = \frac{1}{N}(0, -a_2^*, a_1^*)\,,\ N^2 = |a_1|^2 + |a_2|^2\,, \tag{1.8}$$
$$\mathbf{c}' = (\mathbf{a}\times\mathbf{b}')^*\,, \tag{1.9}$$

i.e. $\langle \mathbf{a}, \mathbf{b}'\rangle = \langle \mathbf{a}, \mathbf{c}'\rangle = \langle \mathbf{c}', \mathbf{b}'\rangle = 0$ and $||\mathbf{b}'|| = ||\mathbf{c}'|| = 1$. Since the second and third row of the final matrix $U \in SU(3)$ must be (complex) orthogonal to $\mathbf{a}$, we have $\mathbf{b}, \mathbf{c} \in \operatorname{span}\{\mathbf{b}', \mathbf{c}'\}$ and thus

[2]In case unitarity is lost due to rounding errors, it can be restore easily with a Gram-Schmidt procedure:

$$\mathbf{a}_{new} \leftarrow \mathbf{a}\,/\,||\mathbf{a}||\,, \quad \mathbf{b}_{new} \leftarrow \mathbf{b}'\,/\,||\mathbf{b}'||\,, \quad \mathbf{c} \leftarrow (\mathbf{a}_{new}\times\mathbf{b}_{new})^*\,,$$
$$\text{with}\ \ \mathbf{b}' = \mathbf{b} - \mathbf{a}_{new}\,\langle \mathbf{a}_{new}, \mathbf{b}\rangle\,.$$

Algorithm 1: Eight Parameter Gauge Decompression

```
input  : 3 complex numbers $a_1$, $a_2$, $b_0$ and 2 angles $\theta_1$, $\theta_1$
output : full $SU(3)$ matrix

1 $N^2 \leftarrow |a_1|^2 + |a_2|^2$

2 $a_0 \leftarrow \sqrt{1 - N^2}\,(\cos\theta_1, \sin\theta_1)$
3 $c_0 \leftarrow \sqrt{N^2 - |b_0|^2}\,(\cos\theta_2, \sin\theta_2)$

4 $n \leftarrow -1/N^2$
5 $b_1 \leftarrow n\,(b_0 a_0^* a_1 + c_0^* a_2^*)$
6 $b_2 \leftarrow n\,(b_0 a_0^* a_2 + c_0^* a_1^*)$
7 $c_1 \leftarrow n\,(c_0 a_0^* a_1 - b_0^* a_2^*)$
8 $c_2 \leftarrow n\,(c_0 a_0^* a_2 - b_0^* a_1^*)$
```

we can rotate $\mathbf{b}'$ and $\mathbf{c}'$ by a transformation $S \in SU(2)$ into $\mathbf{b}$ and $\mathbf{c}$:

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & s_1 & s_2 \\ 0 & -s_2^* & s_1^* \end{pmatrix} \begin{pmatrix} a_0 & a_1 & a_2 \\ 0 & -\frac{1}{N} a_2^* & -\frac{1}{N} a_1^* \\ N & -\frac{1}{N} a_0^* a_1 & -\frac{1}{N} a_0^* a_2 \end{pmatrix} . \tag{1.10}$$

From this relation one can see that $b_0 = N s_2$ and $c_0 = N s_1^*$, that is, b_0 and c_0 can be used to parametrise the matrix U. The ten (real) parameters $\{\mathbf{a}, b_0, c_0\}$ can then be further reduced to eight parameters by demanding that the first row and column of the matrix U are normalised.[3] Writing two of the complex parameters, say a_0 and c_0, in trigonometric from $z = |z|\, e^{i\theta}$, these two conditions translate into conditions for the modulus of the two complex parameters, such that

$$||\mathbf{a}|| = 1 \Leftrightarrow a_0 = \sqrt{1 - N^2}\, e^{i\theta_1} , \tag{1.11}$$

$$|a_0| + |b_0| + |c_0| = 1 \Leftrightarrow c_0 = \sqrt{N^2 - |b_0|^2}\, e^{i\theta_2} , \tag{1.12}$$

that is, a_0 and c_0 are parametrised by only one real parameter respectively. Thus a full $SU(3)$ matrix U can be reconstructed from a set $\{b_0, a_1, a_2, \theta_1, \theta_2\}$ with the help of Alg. 1. This involves in total a 88 floating point operations (additions and multiplications), one division, and two square root, two sines and two cosines evaluations[4] to save a total of 10 reads. Although, this sounds extensively many, on platforms like GPGPUs this has proven to be beneficial [22]. This is because, for this architectures, fast (hardware) functionality for trigonometric and square root functions are available and the rate of throughput per moved byte is rather high.

With the projection onto half spinors and the (optional) gauge compression in place, we can now summarise the application of the *dslash* operator onto a spinor field in Alg. 2.

1.1.2 Performance Model and Boundedness

To get an idea, how this basic algorithm will perform we can first calculate the *arithmetic intensity*, that is the number of floating point operations per byte moved from/to memory. Let us first explicitly

[3] We are following Ref. [22] here. For an approach using stereographic projection see [19].

[4] Note that some of the floating point operations for additions and multiplications may be reduced through the use of fused multiply/add functionality on Xeon and Xeon Phi processors.

Algorithm 2: Basic *dslash* Stencil

```
input : spinor field psi [position][colour][spin],
        gauge field U [position][dimension][direction][colour][colour]
output: spinor field result [position][colour][spin]

Allocate temporary arrays: h [colour][2], Uh [colour][2]
foreach site $x \in \Lambda$ do
    Zero-out result [x][:][:]
    for $dim \leftarrow 0$ to 3 do
        for $dir$ = forward, backward do
            nb = neighbour_index ($x$, $dim$, $dir$)
            Decompress link variables U [$x$][$dim$][$dir$][:][:]
            for $col \leftarrow 0$ to 2 do
                h [$col$][0] = Project_0 ($dim$, $dir$, psi [nb ][$col$][:])
                h [$col$][1] = Project_1 ($dim$, $dir$, psi [nb ][$col$][:])
            end
            if $dir == forward$ then
                Uh [$col$][0] = SU3_Multiply (U [$x$][$dim$][$dir$][$col$][:], h [:][0])
                Uh [$col$][1] = SU3_Multiply (U [$x$][$dim$][$dir$][$col$][:], h [:][1])
            else
                Uh [$col$][0] = SU3_Adj_Mult (U [$nb$][$dim$][$dir$][$col$][:], h [:][0])
                Uh [$col$][1] = SU3_Adj_Mult (U [$nb$][$dim$][$dir$][$col$][:], h [:][1])
            end
            for $col \leftarrow 0$ to 2 do
                result [$x$][$col$][0] += Uh [$col$][0]
                result [$x$][$col$][1] += Uh [$col$][1]
                result [$x$][$col$][2] += Reconstruct_2 ($dim$, $dir$, Uh [$col$][:])
                result [$x$][$col$][3] += Reconstruct_3 ($dim$, $dir$, Uh [$col$][:])
            end
        end
    end
end
```

show that the addition and multiplication of two complex numbers $z_i = (a_i, b_i) \equiv a_i + ib_i$

$$z_1 + z_2 = (a_1 + a_2, b_1 + b_2) \tag{1.13}$$
$$z_1 * z_2 = (a_1 * b_1 - a_2 * b_2, a_1 * b_2 + a_2 * b_1) \tag{1.14}$$

requires two and six floating point operations, respectively.

Let us count the number of operations needed for one of the eight directions first. For the first step of the algorithm we have to calculate h_0 and h_1, which require an addition of two colour-vectors, twice. We thus need $3 \times 2 \times 1$ complex adds which make up 12 floating point operations. Next, we have to evaluate two matrix-vector products of the form $\sum_b U^{ab} h_i^b$ for $i = 1, 2$ for every colour $a = 0, 1, 2$, that is for every row of the matrix U. For each row we need three complex multiplications and two complex additions, that is in total

$$\underbrace{2}_{h_0, h_1} \times \underbrace{3}_{\text{colours}} \times \left(\underbrace{3 \times 6}_{\text{complex mults}} + \underbrace{2 \times 2}_{\text{complex adds}} \right) = 132 \text{ flops} . \tag{1.15}$$

After the matrix multiplication, we have to reconstruct the lower half spinor, which however only involves swaps in memory (with possible sign changes) and thus does not require any floating point operations at all. That is for all eight directions we need $8 \times (12 + 132 + 0) = 1152$ floating point operations up until now. Finally, we have to accumulate the eight full (four component) spinors to the overall result spinor. Each of the eight spinors has 24 real components ($3 \times 4 \times 2$ for colour, spin and real/imaginary) and thus the 7 necessary adds amount for another 168 floating point operations. That is, the *dslash* requires 1320 floating point operations in total per site. As we have mentioned before, the two-row gauge compression requires 24 additional floating point operations,[5] which we will keep in mind to see if, in case gauge compression is used, the algorithm becomes compute bound. However, we will not add them to the arithmetic intensity, because they do not contribute to the evaluation of the matrix product.

In terms of memory transfer the algorithms demands to read eight spinors and eight link matrices per site. The latter have at most 18 ($3 \times 3 \times 2$ for colour, colour, real/imaginary) components. In addition, we need to write one full spinor with 24 components as the result. We thus have

$$\underbrace{8}_{\text{neighbours}} \times \left(\underbrace{24}_{\text{spinors read}} + \underbrace{18/12}_{\text{links read}} \right) + \underbrace{24}_{\text{spinor written}} = 360/312 \tag{1.16}$$

memory movements in total. The arithmetic intensity is then given by

$$I = \frac{F}{B}, \tag{1.17}$$

the number of floating point operations divided by the amount of memory transfer in bytes. Tab. 1.2 summarises different intensities of various precision, with and without the use of 12 parameter gauge compression.

On the other hand the maximal throughput per byte for the Xeon Phi processors can be estimated as follows. KNL (KNC) has a memory bandwidth quoted with approximately 450 (320) GB/s. The theoretical peak performance can be calculated as 6912 (2022) GFLOP/s. Thus the maximal throughput per byte is 15.36 (6.32) FLOP/Byte.[6] Since the algorithmic intensity is always

[5] We will not consider 8 parameter compression any further in this work.

[6] These (single precision) numbers have to be taken with a grain of salt what concerns both *sustained* bandwidth as well as the peak performance, cf. [44] and the next chapter.

Precision	Compression	Intensity [Byte^{-1}]
half	yes	2.16
half	no	1.83
single	yes	1.06
single	no	0.92
double	yes	0.53
double	no	0.46

Table 1.2: Arithmetic intensity for the dslash algorithm, which needs 1320 flops and read/write 360/312 Byte per site, depending if 12 parameter gauge compression is used.

less than these values, we see that the *dslash* operation will be *memory bandwidth bound*, as soon as the required (input) data does not fit into caches.

We will refine the above model in the next chapter for Xeon Phi's, taking into account hardware specific details of the memory traffic.

1.1.3 Clover Term and Twisted-Mass

After having introduced and analysed the *hopping* part of the fermion matrix M, which we referred to as the *dslash* stencil operator $\not{D}$, we will now turn to the mass-like terms.

As we have pointed out in the introduction, the discretised action of the lattice version of QCD is far from being unique. In particular, any term of $\mathcal{O}(a)$ and higher can be added, so that the continuum action remains unchanged. Considered the other way around, any term of this form is an artefact of the numerical method and only disappears in the continuum limit $a \to 0$. This limit, however, is hard to take in practice: the lattice spacing a has to be send to zero in such a way, that the physical volume remains unchanged

$$V_{\text{phys}} = L^4 = Na^4 = \text{const.} \tag{1.18}$$

That is, reducing the lattice spacing by a factor of two, demands for an increase of the number of lattice sites by a factor of $2^4 = 16$. This is why in practice simulations are always performed for finite lattice spacings and then dealt with by extrapolating towards $a = 0$.

In this section we want to introduce two terms that can be added to the lattice action in order to reduced the lattice artefacts by $\mathcal{O}(a)$. Both terms contribute to the local part of the fermion matrix as follows

$$M^{ab}_{\mu\nu} = (4+m)\delta^{ab}\delta_{\mu\nu} \pm i\mu\,\gamma_{5\mu\nu}\,\delta^{ab} + c_{sw}T^{ab}_{\mu\nu} - \frac{1}{2}\not{D}^{ab}_{\mu\nu}\,, \tag{1.19}$$

where we explicitly denoted colour and spin indices. The term proportional to μ (the *twisted-mass parameter*) was introduced in [32, 33, 34] with the motivation to remove so-called *exceptional configurations*, thus regulating the low-energy (infrared) physics. We will refer to it simply as the *twisted-mass term*.[7] For a discussion of the $\mathcal{O}(a)$ improvement of the action see, *e.g.* [35] (also cf. [65] for a detailed review of twisted-mass fermions).

[7]The twisted mass term is actually non-trivial in flavour space, that is, in its simplest version, it describes two (degenerate) fermions. Here, we will only note that this is the origin of the "$\pm$" in front of the term.

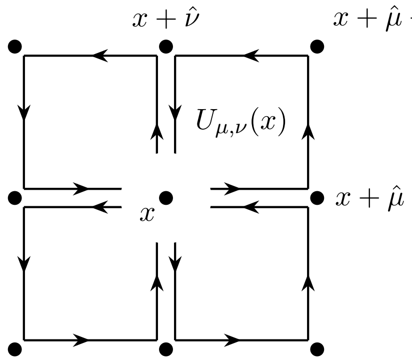

Figure 1.1: The sum of the four plaquettes in the clover term remind of the four leaves of a clover, adapted from [35].

The second new term $c_{sw}T^{ab}_{\mu\nu}$ is the so-called *clover term* [64] which reads

$$T^{ab}_{\mu\nu} = -\frac{i}{16}\,\sigma_{\mu\nu}\hat{F}^{ab}_{\mu\nu}\,, \tag{1.20}$$

where $\sigma_{\mu\nu} = \frac{1}{2}(\gamma_\mu\gamma_\nu - \gamma_\nu\gamma_\mu)$ are anti-symmetrised products of gamma-matrices and $\hat{F}^{ab}_{\mu\nu}$ is (a specific form) of the discretised ***field strength tensor*** that consists of products of link matrices:[8]

$$\begin{aligned}
\hat{F}_{\mu\nu}(x) &= (Q_{\mu\nu}(x) - Q_{\nu\mu}(x))\,,\\
Q_{\mu\nu}(x) &= U_{\mu,\nu}(x) + U_{\nu,-\mu}(x) + U_{-\mu,-\nu}(x) + U_{-\nu,\mu}(x)\,,\\
U_{\mu,\nu}(x) &= U_\mu(x)\;U_\nu(x+\hat{\mu})\;U_{-\mu}(x+\hat{\mu}+\hat{\nu})\;U_{-\nu}(x+\hat{\nu})\,,
\end{aligned} \tag{1.21}$$

The product of the four link variables in the last expression (the aforementioned *plaquette*) forms a closed, oriented loop on the lattice. The four closed loops appearing in $\hat{F}_{\mu\nu}$ emanate from the same lattice site and are reminiscent of the four leaves of a clover (cf. Fig. 1.1)—which is the origin of the name of the term.

Before we have a look on how to evaluate the local terms of the fermion matrix M, let us give a brief motivation on how (a class of terms like) the clover term can improve the action to $\mathcal{O}(a)$ and higher.

Given any operator $\mathrm{O}^{\text{cont.}}$ in the continuum action[9], one builds a (non-unique) discretised operator $\mathrm{O}^{\text{disc.}}$ making some error that will depend on the lattice spacing a and only disappear in the continuum limit

$$\mathrm{O}^{\text{disc.}}[a] = \mathrm{O}^{\text{cont.}} + \mathcal{O}(a^n)\,, \tag{1.22}$$

for some constant $n \geq 1$. An improvement of $\mathcal{O}(a)$ is equivalent of increasing n by 1, on the right-hand side. To improve the original operator, one can add a power series in the lattice spacing a to the original discretised operator

$$\mathrm{O}^{\text{disc.}}[a] + \sum_{i=1}^{N-1} c_i\,a^i\,\mathrm{O}^{\text{disc.}}_i[a] = \mathrm{O}^{\text{cont.}} + \mathcal{O}(a^N)\,, \tag{1.23}$$

[8]We suppress colour indices here, in order not to clutter the notation.

[9]Our discussion here will focus only on the operators in the action, but similar considerations apply for observables, *i.e.* operators in correlation functions, as well.

with real coefficients c_i. Here, the new operators $\mathrm{O}_i^{\text{disc.}}$ should respect the symmetries of the original (discretised) operator and derive from a (higher dimensional[10]) continuum operator $\mathrm{O}_i^{\text{disc.}} = \mathrm{O}_i^{\text{cont.}} + \mathcal{O}(a^{N-i})$. That is, the higher dimensional the operator, the more "inaccurate" the discretisation may be in terms of a. Also note, that this procedure is note unique.

Following [35], let us illustrate this procedure with the simple example of a one-dimensional derivative operator $\mathrm{O}^{\text{cont.}} = \frac{\mathrm{d}}{\mathrm{d}x}$, acting on the "scalar field" $f(x)$. Starting with a symmetric difference as the discretised operator, one can use the Taylor expansion of $f(x)$ around $x+a$ to obtain the initial error done when discretising

$$\frac{f(x+a)-f(x-a)}{2a} = f'(x) + \mathcal{O}(a^2)\,. \tag{1.24}$$

In particular, since the initial $\mathrm{O}^{\text{disc.}}$ is symmetric in a, we can only ever have even powers of the lattice spacing on the right-hand side and it is easy to see that the first correction term is $\frac{1}{3!}f'''(x)$. Thus, in order to improve the first discretised operator of the first derivative, we have to add a term $-\frac{1}{6}a^2 D^{(3)} f(x)$ to the right-hand side, where $D^{(3)}f = f''' + \mathcal{O}(a^2)$. Again with the Taylor expansion one can find such a (non-unique) term

$$D^{(3)} f(x) = \frac{f(x+2a) - 2f(x+a) + 2f(x-a) - f(x-2a)}{2a^3} \tag{1.25}$$

such that the sum of both discretised operators is now $f'(x) + \mathcal{O}(a^4)$.

Using an equivalent procedure in lattice QCD (the *Symanzik improvement program*) was advocated in [46, 73, 74]. To lowest order, there turns out to be only one continuum operator, of which the clover term, as we will use it here, is one possible discretisation.

Let us now turn to the application of the local term of the fermion matrix, which we will denote[11] by $A = (4+\alpha)\mathbb{1} \pm i\mu\gamma_5 + c_{SW}T$, to a spinor. In the DeGrand-Rossi basis A is block-diagonal[12]

$$A = \begin{pmatrix} (\alpha \pm i\mu)\,\mathbb{1}_6 + c_{SW}B_0 & 0 \\ 0 & (\alpha \mp i\mu)\,\mathbb{1}_6 + c_{SW}B_1 \end{pmatrix}\,, \tag{1.26}$$

with blocks of dimension $(3\,\text{colour} \times 2\,\text{spin})^2$. That is, once again, the components of the upper and lower half spinors do not mix during the matrix multiplication. The 6×6 clover blocks B_0 and B_1 are hermitian (and traceless)

$$B_0 = \frac{1}{16}\begin{pmatrix} -\hat{F}_{12} + \hat{F}_{34} & i\hat{F}_{13} + \hat{F}_{14} - \hat{F}_{23} + i\hat{F}_{24} \\ -i\hat{F}_{13} + \hat{F}_{14} - \hat{F}_{23} - i\hat{F}_{24} & \hat{F}_{12} - \hat{F}_{34} \end{pmatrix}\,, \tag{1.27}$$

$$B_1 = \frac{1}{16}\begin{pmatrix} -\hat{F}_{12} - \hat{F}_{34} & i\hat{F}_{13} - \hat{F}_{14} - \hat{F}_{23} - i\hat{F}_{24} \\ -i\hat{F}_{13} - \hat{F}_{14} - \hat{F}_{23} + i\hat{F}_{24} & \hat{F}_{12} + \hat{F}_{34} \end{pmatrix}\,, \tag{1.28}$$

which follows from the unitarity of $U_\mu(x)$ and the fact that $Q_{\mu,\nu} = Q_{-\mu,-\nu}$. However, the full matrix A is hermitian if and only if $\mu = 0$. In particular, when including the twisted-mass term, A^{-1} will in general not have upper and lower triangular parts that are hermitian conjugate to each other. This will be of importance later on, when we will introduce even-odd preconditioning. In

[10]Higher in mass dimensions $[a^{-1}]$, the traditional from of counting dimensionality in high energy physics.

[11]By a slight abuse of notation we will also refer to A as the clover term, irrespective of the value of μ.

[12]Again, this is also true for the Weyl basis.

Precision	Hermitian	Intensity [Byte^{-1}]
half	yes	2.85
half	no	1.64
single	yes	1.43
single	no	0.82
double	yes	0.71
double	no	0.41

Table 1.3: Arithmetic intensity for the mass term multiplication for various floating point precisions, with and without using hermiticity of the blocks.

case the matrices A and A^{-1} are not hermitian, the number of elements one has to read, in order to perform the matrix-vector multiplication, will essentially double.

In order to calculate the product of A and a full spinor, for each block and each of the six rows, we have to calculate the vector product of a complex six-vector. This requires six complex multiplications and five complex additions:[13]

$$\underbrace{2}_{\text{blocks}} \times \underbrace{6}_{\text{rows}} \times \left(\underbrace{6 \times 6}_{\text{complex mults}} + \underbrace{5 \times 2}_{\text{complex adds}} \right) = 552 \text{ flops}\,. \tag{1.29}$$

Furthermore, we have to read and write a full (24 component) spinor from/to memory. Lastly, we have to read the clover matrix itself. In case it is hermitian, we only need to read six (real) diagonal components and 15 complex off-diagonal components (say, the ones of the upper triangle) for each of the two blocks. In case the matrix is not hermitian however, we need to read two full 6×6 complex blocks. That is, in total we need to read and write 96 or 168 elements respectively. In Tab. 1.3 we summarise the resulting arithmetic intensity for the clover block multiplication alone, for various precisions and in both aforementioned cases.

Note that in the hermitian case, the arithmetic intensity is always higher than the respective one of the *dslash* operator, independently whether gauge compression is used or not. The exact opposite is true, when having to read the full complex blocks of the clover term A.

1.2 Iterative Solvers

As we have mentioned before, in order to calculate observables and generate configuration in lattice QCD it will be necessary to invert the full fermion matrix M and solve linear systems of the form $Mx = b$ for x with some given right-hand side b.

Historically, this could always be done with a class of algorithms called *direct*, where the matrix in question (or the linear system) is manipulated directly. Examples of such algorithms are the *Gaussian Elimination* and the *QR factorisation*. Unfortunately, this class of algorithms usually requires a number of $\mathcal{O}(n)$ steps, each of which takes $\mathcal{O}(n^2)$ work to complete, where n is the side length of the matrix.[14] Furthermore, they usually require the matrix to be stored entirely, both of

[13]In case the matrix is hermitian, one of the six multiplications is between a real and a complex number, which requires 2 floating point operations. The total number of flops required in this case is thus 548.

[14]Famous exceptions of direct solvers with better asymptotic scaling are the ones by Strassen [70], and Coppersmith & Winograd [23].

which conditions are unsatisfactory in the case of the huge fermion matrix, we are dealing with.

Fortunately, there is another type of algorithms known that solely relies on the ability to calculate products of the form Mx. These algorithms are called *iterative*, because they approximate the true solution better and better in each step. An important subclass of iterative solvers are the so-called *Krylov subspace methods*. They rely on the construction of an iteratively growing subspace of the form $\mathrm{span}\,(b, Mb, M^2b, M^3b, \dots)$—the *Krylov sequence*. In this subspace a solution vector is formed, such that the residuum to the true solution is minimised.

The main advantages of this method is that any sort of symmetry of the matrix can be used to optimise the “black box”, which is the matrix multiplication Mx. In particular, the sparsity of a matrix can be used to reduce the asymptotic scaling of this multiplication. In a direct solve, on the contrary, an initial sparsity pattern is usually destroyed. In this way, the asymptotic scaling behaviour of an iterative solver may demand only $\mathcal{O}(1)$ steps, each of which may require only $\mathcal{O}(n)$ flops to complete. In this case, the iterative solver scales two orders of magnitude better than a typical direct solver.

In this section, we would like to introduce three of such iterative algorithms, exclusively focusing on the ones implemented in the QPhiX library. This will be the *conjugate gradient* method [37] which was the first of its kind, the *stabilised biconjugate gradient* method [77] that improves the numerical stability of the original *BiCG* algorithm which uses two subspaces to circumvent the limitations of CG, and finally a variant of the *Richardson Iteration*. We will conclude this section with a first glimpse on the vast subject of algorithmic improvements to iterative solvers in the form of *preconditioning*.

Our presentation will leave out at least one very important iterative algorithm—the *generalized minimal residual method* (GMRES) [60]. More about iterative methods may be found in the excellent reference [59].

1.2.1 Conjugate Gradient Method

The conjugate gradient method was the first iterative method to solve hermitian (or symmetric in the real case) systems of linear equations $Mx = b$ where $M \in \mathbb{C}^{m\times m}$ is a positive definite complex matrix. It was proposed in 1952 by Hestenes and Stiefel [37] and convergences very rapidly for systems with well separated eigenvalues or small spectral condition numbers k. It can be viewed as an iterative procedure to minimise a quadratic function $\varphi(x)$ on $\mathbb{C}^m$, which corresponds to a particular norm of the residuum vector $e_n = x_\star - x_n$, which measures the distance of the exact solution $x_\star$ to the iterative one x_n at step n.

Since M is hermitian and positive definite, all its eigenvalues are real and strictly positive. In particular, $\langle x, Mx\rangle > 0$, $\forall x \neq 0$ and $||x||_M \equiv \sqrt{\langle x, Mx\rangle}$ defines a norm (the *M-norm*) on $\mathbb{C}^m$. It is with respect to this norm that the residuum vector e_n is minimised by the CG method, or equivalently the function

$$\varphi(x_n) \equiv \frac{1}{2}\langle x_n, Mx_n\rangle - \langle x_n, b\rangle = \frac{1}{2}||e_n||^2 + \text{const.}, \tag{1.30}$$

where the last equality follows directly from the definitions. The essence of a CG iteration is now the update step

$$x_n = x_{n-1} + \alpha_n\, p_{n-1}\,, \tag{1.31}$$

which generates the unique iterative solution in the Krylov subspace $\mathcal{K}_n$ that minimises the norm of the residuum. We will refer to the coefficient α_n as the *step length* and the vector p_{n-1} as the *search direction* of the optimisation procedure.

Algorithm 3: Conjugate Gradient

```
1 $x_0 = 0,\ r_0 = b,\ p_0 = r_0$
2 for $n = 1, 2, \ldots$ until convergence do
3     $\alpha_n = \langle r_{n-1}, r_{n-1}\rangle / \langle p_{n-1}, M\, p_{n-1}\rangle$
4     $x_n = x_{n-1} + \alpha_n\, p_{n-1}$
5     $r_n = r_{n-1} - \alpha_n\, M\, p_{n-1}$
6     $\beta_n = \langle r_n, r_n\rangle / \langle r_{n-1}, r_{n-1}\rangle$
7     $p_n = r_n + \beta_n p_{n-1}$
8 end
```

It is readily shown by induction that the spaces spanned by the sequence of iterative solutions, residuals and search direction vectors each by themselves are identical to the Krylov subspace that is generated by the matrix multiplication Mp_{n-1} in each and every iteration step [75]:

$$\begin{aligned}\mathcal{K}_n &= \text{span}\left(b, Mb, \ldots, M^{n-1}b\right) \\ &= \text{span}\left(x_1, x_2, \ldots, x_n\right) \\ &= \text{span}\left(r_0, r_1, \ldots, r_{n-1}\right) \\ &= \text{span}\left(p_0, p_1, \ldots, p_{n-1}\right). \end{aligned} \tag{1.32}$$

Furthermore, the residual vectors r_n and search directions p_n are orthogonal in the following way:

$$\langle r_n, r_j\rangle = 0,\ \forall j < n\,, \tag{1.33}$$

$$\langle p_n, Mp_j\rangle = 0,\ \forall j < n\,. \tag{1.34}$$

This is guaranteed by the particular forms of the coefficients α_n and β_n. These last three properties make the conjugate gradient method so powerful, because they imply the monotonic and unique convergence of the algorithm, such that $x_n = x_\star$ is reached for some $n \leq m$ (cf. [75] for details).

Although this is strictly true only for infinite precision, and does not apply using floating point arithmetics, for well-behaved (possibly preconditioned) systems, convergence to the desired precision is often achieved with $n \ll m$ iterations. In particular it can be shown that for a given spectral condition number $k = \lambda_{\text{max}}/\lambda_{\text{min}}$ the norm of the residual at each step n is bound by

$$\frac{||e_n||_A}{||e_0||_A} \leq 2\left(\frac{\sqrt{k}-1}{\sqrt{k}+1}\right)^n \sim \left(1 - \frac{2}{\sqrt{k}}\right)^n, \tag{1.35}$$

where the last relation applies in the limit $k \to \infty$. Thus for large condition numbers the CG method should converge in at most $\mathcal{O}(\sqrt{k})$ iterations, each of which take at most $\mathcal{O}(m^2)$ steps (dense matrix multiplication). However, as we have seen before, the *dslash* application scales only linearly in the number of sites N of the lattice.

1.2.2 Biconjugate Gradient Stabilized Method

Although the conjugate gradient method is very powerful and converges, at least for well-conditioned systems, in a fast and smooth (monotonic) manner, it has the considerable disadvantages to be only applicable to real symmetric or complex hermitian (self-adjoint) linear systems.[15]

[15] In fact, *dslash* only satisfies γ_5-hermiticity $\not{D} = \gamma_5 \not{D}^\dagger \gamma_5$.

Fortunately, there is a number of iterative algorithms that can deal with non-hermitian problems, as well. In the domain of Krylov subspace solvers, there are essentially two approaches to overcome the requirement of self-adjointness. Thereby, one either has to let go of the advantage of three-term recurrence (*i.e.* the fact that the solution vector x_n can be iteratively constructed from the three vectors x_{n-1}, r_{n-1} and p_{n-1}) or one has to construct an additional (orthogonal) Krylov subspace. In the former category the GMRES algorithm with its $(n+1)$-term recurrence is the standard example. We will focus here on the latter case, to which the *Biconjugate gradient* (BiCG) [30] algorithm and its variants belong.

However, before introducing the BiCG algorithm let us note that the restriction of self-adjointness can be overcome even with some variation of the CG method. To that end, one applies the CG algorithm to the *normal equations*

$$M^\dagger M\, x = M^\dagger b \tag{1.36}$$

with new right-hand side $b' = M^\dagger b$. This algorithm is usually referred to as CNG or CNGR. Since M is non-singular, the matrix $M^\dagger M$ is hermitian and positive definite, and CG will converge as described in the last section. The main disadvantage of this technique, however, is the fact that the condition number of the original system is squared. That is, if M has condition number k, the condition number of the new system $M^\dagger M$ will be k^2, such that the number of iterations will only be bound by $\mathcal{O}(k)$.

In the BiCG algorithm on the other hand, the two dual linear systems $Mx = b$ and $M^\dagger \hat{x} = \hat{b}$ are solved simultaneously. To achieve this, two orthogonal Krylov spaces

$$\mathcal{K}_n = \mathrm{span}\,(b, Mb, \ldots, M^{n-1}b) \tag{1.37}$$

$$\mathcal{L}_n = \mathrm{span}\,(\hat{b}, M^\dagger\hat{b}, \ldots, (M^\dagger)^{n-1}\hat{b}) \tag{1.38}$$

are spanned such that the residual vector of the one system is perpendicular to the other subspace $r_n \perp \mathcal{L}_n$. Here $\hat{b}$ has to satisfy $\langle \hat{b}, b\rangle = 1$ and one usually chooses $\hat{b} = b/||b||$.

Algorithm 4: Biconjugate Gradient

```
1  $x_0 = 0,\ r_0 = b,\ p_0 = r_0,\ \hat{r}_0 = b,\ \hat{p}_0 = r_0$
2  for $n = 1, 2, \ldots$ until convergence do
3      $\alpha_n = \langle \hat{r}_{n-1}, r_{n-1}\rangle / \langle \hat{p}_{n-1}, M\, p_{n-1}\rangle$
4      $x_n = x_{n-1} + \alpha_n\, p_{n-1}$
5      $r_n = r_{n-1} - \alpha_n\, M\, p_{n-1}$
6      $\hat{r}_n = \hat{r}_{n-1} - \alpha_n\, \hat{p}_{n-1}\, M^\dagger$
7      $\beta_n = \langle \hat{r}_n, r_n\rangle / \langle \hat{r}_{n-1}, r_{n-1}\rangle$
8      $p_n = r_n + \beta_n\, p_{n-1}$
9      $\hat{p}_n = \hat{r}_n + \beta_n\, \hat{p}_{n-1}$
10 end
```

Note the considerable similarities to the CG method. Specifically, the three-term recurrence form of the algorithm is preserved, which keeps the storage requirements under control, and in particular independent of the number of iterations. It however has the disadvantage that one can not achieve $r_n \perp M\mathcal{K}_n$ in each and every step, but only $r_n \perp \mathcal{L}_n$, and as a consequence the algorithm does not minimise the two-norm $||r_n||_2$ monotonically [75]. The situation is reversed within the GMRES algorithm, where the norm of the residual is smoothly reduced, but one pays the prices in terms of having to store the entire Krylov subspace that has been generated. Another disadvantage of the BiCG method is the need to implement an additional matrix product for the hermitian conjugate matrix $M^\dagger$, although the resulting vectors enter the algorithm only indirectly via the scalar coefficients α_n and β_n.

One of the many proposed algorithms to circumvent the two major problems of BiCG—the erratic, non-monotonous convergence and the additional matrix product—is the *Biconjugate gradient stabilized method* (BiCGStab) [77]. It can be seen as a refinement of the *conjugate gradient squared* (CGS) method [69], where one essentially combines two BiCG steps in such a way that the appearance of the hermitian matrix $M^\dagger$ can be avoided. However, the convergence is still at least as erratic as in the original BiCG algorithm. One can further improve on this fact using the observation that the residual and directional vectors satisfy a recurrence relation

$$r_n = P_n(M)\, r_0\,, \tag{1.39}$$
$$p_{n+1} = T_n(M)\, r_0\,, \tag{1.40}$$

and similar for the hatted vectors replacing M with $M^\dagger$, where $P_n(M)$ and $T_n(M)$ are polynomials in M of degree n. Similarly, one finds recurrence relations directly form the BiCG algorithm for the polynomials themselves

$$P_n(M) = P_{n-1}(M) - \alpha_n\, M\, T_{n-1}(M)\,, \tag{1.41}$$
$$T_n(M) = P_n(M) + \beta_{n+1}\, T_{n-1}(M)\,. \tag{1.42}$$
$$\tag{1.43}$$

The idea of BiCGStab now is to modify the recurrence relation for the residual vector r_n and the directional vector p_n as follows

$$r_n = Q_n(M)\, P_n(M)\, r_0\,, \tag{1.44}$$
$$p_{n+1} = Q_n(M)\, T_n(M)\, r_0\,, \tag{1.45}$$

introducing a new polynomial $Q_n(M) = \prod_{i=1}^n (\mathbb{1} - \omega_i M)$, in such a way, that suitable choices for the coefficients ω_n enable faster and smoother convergence. The details of the derivation of the optimal choice of these coefficients can for instance be found in [59]. Here, we will only give the final algorithm 5 without further explanations.

1.2.3 Modified Richardson Iteration

The *Modified Richardson Iteration*, originally proposed in 1911 [56], obtains the solution to a linear system M as the fixed point to a class of iterative recurrence relations. This is based on the observation that the original linear system can be rewritten in the form $x = x + \tau(b - Mx)$, for any non-vanishing real value τ. In this way one can *define* the n^{th} iterative solution as the result of the left-hand side evaluated on the solution of the previous step:

$$x_n = x_{n-1} + \tau(b - Mx_{n-1})\,. \tag{1.46}$$

In this way, a *fixed point*, $x_n = x_{n-1}$, of the recurrence relation must also satisfy the original linear system $Mx_n = b$.

Algorithm 5: Biconjugate Gradient Stabilized

```
1  $x_0 = 0,\ r_0 = b,\ p_0 = r_0,\ \hat{r}_0$ arbitrary
2  for $n = 1, 2, \ldots$ until convergence do
3      $\alpha_n = \langle \hat{r}_0, r_{n-1}\rangle / \langle \hat{r}_0, M p_{n-1}\rangle$
4      $s_n = r_{n-1} - \alpha_n M\, p_{n-1}$
5      $\omega_n = \langle s_n, M s_n\rangle / \langle M s_n, M s_n\rangle$
6      $x_n = x_{n-1} + \alpha_n p_{n-1} + \omega_n s_n$
7      $r_n = s_n - \omega_n M s_n$
8      $\beta_n = \frac{\langle \hat{r}_0, r_n\rangle}{\langle \hat{r}_0, r_{n-1}\rangle} \times \frac{\alpha_n}{\omega_n}$
9      $p_n = r_n + \beta_n (p_{n-1} - \omega_n M p_{n-1})$
10 end
```

Convergence will in general be achieved under the following circumstances. From the recurrence relation we see that the residuum to the exact solution $e_n = x_\star - x_n$ will satisfy

$$e_n = (\mathbb{1} - \tau M)\, e_{n-1}\,, \tag{1.47}$$

which implies $||e_n|| \leq ||\mathbb{1} - \tau M||\, ||e_{n-1}||$ for any norm. In particular,

$$\frac{||e_n||}{||e_0||} = ||\mathbb{1} - \tau M||^n\,, \tag{1.48}$$

will converge to zero in the limit $n \to \infty$ whenever the norm satisfies $||\mathbb{1} - \tau M|| < 1$. In the case of a hermitian matrix M, all eigenvalues are real and positive and the last condition can be satisfied by demanding $|1 - \tau\lambda_i| < 1$ for all eigenvalues λ_i. This can clearly be satisfied by choosing $0 < \tau < 2/\lambda_{\mathrm{max}}$.

Iterative solvers of the more general form

$$x_n = G\, x_{n-1} + f\,, \tag{1.49}$$

for some linear operator G are studied in the literature (cf. *e.g.* [59]). From quite general considerations it can be shown, that the optimal value for τ in the case of the modified Richardson iteration is given by [59]

$$\tau_{\mathrm{opt}} = \frac{2}{\lambda_{\mathrm{min}} + \lambda_{\mathrm{max}}}\,. \tag{1.50}$$

This sort of iterative solver has great practical value in particular in the context of mixed floating point precision, as discussed in greater detail below. In this case, a low-precision solution x_{n-1} may be found by means of some *inner solver* and then "improved" through a modified Richardson iteration using the high-precision matrix and vector operations, achieving an overall improved rate of convergence.

1.2.4 Preconditioning

When solving a linear system of equation with an iterative method the rate of convergence of the algorithm will in general depend on the condition number of the matrix involved. Most iterative solvers will increase their rate of convergence when the condition number of the matrix is decreased, as we have seen explicitly in the case of the CG method.

This can be achieved with a technique called preconditioning, for which the original matrix M is transformed by a linear transformation P (the *preconditioner*), such that the resulting linear operator $\tilde{M} \equiv PM$ has a smaller condition number. Instead of solving the system $Mx = b$, one then solves the better behaved system

$$\tilde{M}y = (MP)(P^{-1}x) = b \tag{1.51}$$

for the vector y, such that the solution to the original system can be obtained from $x = Py$.

Preconditioning of linear systems is a very powerful tool, but also a very complex matter. Here, we will only focus on two simple (but potent) methods, which will be used later in QPhiX.

Even-Odd Preconditioning

The first algorithm of preconditioning we want to introduce is called even-odd preconditioning [28]. The main idea of this method is to separate the lattice into an even and an odd sublattice, which are interlaced such that even sites only have odd neighbouring sites and vice versa. Since, starting from a rectangular lattice in two dimensions, these sublattices resemble a chess or checkerboard this method is also referred to as *checkerboarding*.

This method is very powerful, because as we will see shortly, it reduces the condition number of the checkerboarded matrix to roughly half of the original linear operator [41]. Moreover, it will also allow to solve the system on one sublattice, only. This effectively reduces the volume by a factor of two.

Having separated the lattice into an even and an odd part, the original linear operator can be written as a block matrix of the four operators connecting even to even, odd to odd, even to odd and odd to even sites, respectively:

$$M = \begin{pmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{pmatrix} . \tag{1.52}$$

Note that this merely consists of a reordering and that the sub-matrices still carry two volume indices each. This block separation will turn out to be particularly useful whenever the off-diagonal part of the original operator only connects even to odd sites and vice versa.

The only requirement for this technique to be beneficial, is the ability to invert the even-even operator relatively easily. This will be the case for all variants of the Dirac matrix we are considering here. Then, as we will see shortly, one is left having to invert a matrix connecting odd to odd sites only.

The first step of the even-odd preconditioning is to rewrite the decomposed operator as a product of triangular matrices:

$$\begin{pmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{pmatrix} = \begin{pmatrix} M_{ee} & 0 \\ M_{oe} & 1 \end{pmatrix} \begin{pmatrix} 1 & M_{ee}^{-1}M_{eo} \\ 0 & M_{oo} - M_{oe}M_{ee}^{-1}M_{eo} \end{pmatrix} . \tag{1.53}$$

This product, known as the (asymmetric) *Schur decomposition* allows for a simple inversion of the matrix M which is still a product of triangular matrices:

$$M^{-1} = \begin{pmatrix} 1 & M_{ee}^{-1}M_{eo} \\ 0 & M_{oo} - M_{oe}M_{ee}^{-1}M_{eo} \end{pmatrix}^{-1} \begin{pmatrix} M_{ee} & 0 \\ M_{oe} & 1 \end{pmatrix}^{-1} \tag{1.54}$$

$$= \begin{pmatrix} 1 & -M_{ee}^{-1}M_{eo}\tilde{M}_{oo}^{-1} \\ 0 & \tilde{M}_{oo}^{-1} \end{pmatrix} \begin{pmatrix} M_{ee}^{-1} & 0 \\ -M_{oe}M_{ee}^{-1} & 1 \end{pmatrix} , \tag{1.55}$$

where we have introduced the preconditioned odd-odd operator

$$\tilde{M}_{oo} = M_{oo} - M_{oe} M_{ee}^{-1} M_{eo} \,. \tag{1.56}$$

Now we are able to formally write the solution to the full system as follows

$$\begin{pmatrix} x_e \\ x_o \end{pmatrix} = \begin{pmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{pmatrix}^{-1} \begin{pmatrix} b_e \\ b_o \end{pmatrix} = \begin{pmatrix} M_{ee}^{-1} b_e - M_{ee}^{-1} M_{eo} \tilde{M}_{oo}^{-1} \tilde{b}_o \\ \tilde{M}_{oo}^{-1} \tilde{b}_o \end{pmatrix} , \tag{1.57}$$

where we have introduced the newly prepared (odd) source term

$$\tilde{b}_o = -M_{oe} M_{ee}^{-1} b_e + b_o \,. \tag{1.58}$$

To solve the full linear system with even-odd preconditioning one is thus led to the following algorithmic procedure:

Algorithm 6: Even-Odd Preconditioning

```
input  : Matrix $M$, source vector $b$
output : Solution vector $x$
1 Invert $M_{ee}$
2 Prepare the source $\tilde{b}_o = -M_{oe} M_{ee}^{-1} b_e + b_o$
3 Solve $\tilde{M}_{oo} x_o = \tilde{b}_o$
4 Reconstruct even sites $x_e = M_{ee}^{-1}(b_e - M_{eo} x_o)$
```

This method is obviously only beneficial as long as the even-even part of the operator can be inverted cheaply. In this case, however, almost all the time of the algorithm will be spend inverting a linear system, which only involves odd sites. This is again done with an iterative solver involving the preconditioned operator only, which we will give below explicitly for all the considered cases. As long as one does not include a clover term in the Dirac matrix, the even-even part can be inverted analytically (even with twisted-mass) and one does not have to perform step 1 of the algorithm. As soon as one adds the clover term, M_{ee} is best inverted numerically, using a direct solve such as a LU decomposition.

In the case of Wilson fermions the even-even and odd-odd parts are identical and proportional to the identity matrix. Then we have $M_{ee} = (4 + m)\,\mathbb{1} \equiv \alpha\,\mathbb{1}$ and $M_{eo} = -\frac{1}{2}\,D_{eo}$, such that the inverse is simply given by $M_{ee}^{-1} = \frac{1}{\alpha}\mathbb{1}$ and the preconditioned odd-odd operator (cf. [43]) by

$$\tilde{M}_{oo} = \alpha\,\mathbb{1} - \frac{1}{4\alpha}\,D_{oe} D_{eo} \,. \tag{1.59}$$

For later convenience we note that the action of the operator on some (odd) spinor field χ can be obtained by applying two linear algebra routines

$$\psi = \not{D}\,\chi \,, \tag{1.60}$$

$$\tilde{M}_{oo}\,\chi = a\,\chi - b\,\not{D}\,\psi \,, \tag{1.61}$$

with $a = \alpha$, $b = 1/4\alpha$ and $\not{D}$ being the appropriate version of the *dslash* operator.

The case of twisted-mass fermions with clover term is slightly more involved. We will be particularly interested in the two-flavour degenerate mass operators, which read

$$M_{ee}^{\pm} = \alpha\,\mathbb{1} \pm i\mu\gamma_5 + c_{sw}T\,, \tag{1.62}$$

$$M_{eo} = -\frac{1}{2}\,D_{eo}\,. \tag{1.63}$$

The preconditioned operator then assumes the form

$$\tilde{M}_{oo}^{\pm} = (\alpha\,\mathbb{1} \pm i\mu\gamma_5 + c_{sw}T_{oo}) - \frac{1}{4}\,D_{oe}(\alpha\,\mathbb{1} \pm i\mu\gamma_5 + c_{sw}T_{ee})^{-1}D_{eo}\,, \tag{1.64}$$

with T_{oo} and T_{ee} being the odd-odd and even-even part of the clover term, respectively. Again, we can construct this operator with two linear algebra routines as

$$\psi = A^{-1}\not{D}\,\chi\,, \tag{1.65}$$

$$\tilde{M}_{oo}\,\chi = A\,\chi - b\,\not{D}\,\psi\,. \tag{1.66}$$

with $b = 1/4$, A and A^{-1} being the sum of the mass and clover term and its inverse, which are evaluate on opposing sublattices.

In the case of $\mu \neq 0$, the inverse of the clover term can for instance be obtained with the method of *LU decomposition with pivoting*.[16] The idea is to transform the matrix A into an upper-triangular matrix. To do so, one uses *Gaussian Elimination*, where one subtracts multiples of a given row (the $(i+1)^{\text{th}}$ in the i^{th} step) from all the subsequent rows. Then each iteration is essentially the multiplication of a unit lower-triangular matrix with A such that

$$L_{n-1}\,\cdots\,L_2\,L_1\,A = U, \tag{1.67}$$

where n is the dimension of the matrix A. By inverting[17] the product of transformations $L \equiv L_1^{-1}L_2^{-1}\cdots L_{n-1}^{-1}$ we obtain another unit lower-triangular matrix and thus arrive at an LU decomposition

$$A = L\,U\,. \tag{1.68}$$

In practice, at each and every step one has to choose the row and column with the biggest element and apply Gaussian Elimination to the respective sub-matrix. In this way, one guarantees numerical stability of the algorithm, because one circumvents the subtraction of large numbers. This selection scheme is called *pivoting*.

Given an LU decomposition, it is simple to solve a linear system $A\,x = b$. To this end, one solves the system $L\,y = b$ first and subsequently the system $U\,x = y$. This is done very easily using the method of forward- and backward substitution, respectively. This way, one obtains the full inverse of A by solving the multi-system $A\,X = B$, with B being the identity matrix $\mathbb{1}$. For more details on Gaussian Elimination and pivoting consult the excellent lecture series [75].

To conclude the section we will give an outline how the reduction of the condition number comes about in the simplest case of Wilson fermions. Making use of the identity

$$\det\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \det(A)\det(D - CA^{-1}B)\,, \tag{1.69}$$

[16] An alternative would be Cholesky factorization which is used in QDP++.

[17] The inversion actually turns out to be trivial and does not require any floating point operations.

one can rewrite the determinant of the full linear operator as follows

$$\det(\mathbb{1} - \kappa \not{D}) = \frac{1}{\alpha} \det M = \det(\mathbb{1} - \kappa^2 D_{oe} D_{eo}) \,, \tag{1.70}$$

where we divided by α and $\kappa = 1/2\alpha = 1/2(4+m) < 1$. Obviously, to every eigenvalue ν of $\not{D}$ there is an eigenvalue ν^2 of $D_{oe} D_{eo}$. Let $\nu = 1/(\kappa + \varepsilon)$ be an eigenvalue of $\not{D}$, such that the smallest eigenvalue λ of M will be

$$\lambda = 1 - \kappa\nu = 1 - \frac{\kappa}{\kappa + \varepsilon} \simeq \frac{\varepsilon}{\kappa} \,. \tag{1.71}$$

The smallest eigenvalue of the preconditioned operator, on the other hand, is

$$\tilde{\lambda} = 1 - \kappa^2 \nu^2 = 1 - \frac{\kappa^2}{(\kappa + \varepsilon)^2} \simeq \frac{2\varepsilon}{\kappa} \,. \tag{1.72}$$

That is, the ratio of the spectral condition numbers $k(M) = \lambda_{max}(M)/\lambda_{min}(M)$ scales as

$$\frac{k(\tilde{M})}{k(M)} \simeq \frac{\lambda_{min}}{\tilde{\lambda}_{min}} = \frac{1}{2} \,. \tag{1.73}$$

Mixed Precision Preconditioning

As we have described in the introduction, preconditioning essentially replaces the original system of linear equations $Mx - b = 0$ with the preconditioned system $P(Mx - b) = 0$, such that the condition number is reduced $k(PM) < k(M)$. This can obviously be achieved, whenever the exact inverse is known $P = M^{-1}$, such that $k(PM) = 1$ is very well behaved.

This observation motivates the following idea: what if instead of the exact inverse one uses an approximate inverse $PM \approx \mathbb{1}$, *i.e.* an approximate solution to the linear system in question? Since the matrix-product, as well as the condition number are continuous functions, the condition number of the preconditioned system will still be drastically reduced. The power of this methods then relies solely on the ability to find an approximate solution (that is one with lower numerical precision than desired) relatively cost-efficiently.

In practice, this is indeed feasible by evaluating vector-vector and matrix-vector products within the iterative solvers using different floating point precisions. This is because, all these routines are memory bandwidth-bound for sparse systems (as we know for vector-vector products and have explicitly shown for the *dslash* operator) and thus may benefit from the higher amount of data being transferred from main memory or lying in caches. Historically, GPGPUs additionally had significantly higher throughput in single precision compared to double precision [39, 67]. The idea of so-called *mixed precision solvers* is thus to evaluate most of the costly linear algebra routines with (cheaper) low-precision floating point operations, in such a way, that the worse convergence behaviour that results from more significant rounding errors, is counteracted upon.

There are essentially two different approaches used in practical implementations [22, 39, 67, 72]: *defect-correction* also known as *iterative refinement* [49], where the system is solved to some inner tolerance $\varepsilon_{in} < \varepsilon$ using low-precision operations and then updated with high-precision arithmetics to obtain an overall precision ε; and *reliable updates* [66], where the frequency of those high-precision updates is controlled dynamically.

The simplest approach using mixed precision Krylov solvers is iterative refinement, which we outline in Alg. 7. There, one updates an inner solution vector $\tilde{x}_n$ by solving the linear system

Algorithm 7: Iterative Refinement

```
1 Variables with tilde represent low-precision types and operations.
  $x_0 = 0,\ r_0 = b,\ \tilde{r}_0 = r_0$
2 for $n = 1, 2, \ldots$ until $||r_n|| < \varepsilon$ do
3     Solve $\tilde{M}\,\tilde{x}_n = \tilde{r}_{n-1}$ to precision $\varepsilon_{in}$
4     $x_n = x_{n-1} + \tilde{x}_n$ (accumulate in high-precision)
5     $r_n = b - M x_n$ (calculate true residuum)
6     $\tilde{r}_n = r_n$ (truncate)
7 end
```

$\tilde{M}\,\tilde{x}_n = \tilde{r}_{n-1}$ with an inner solver (that may rely on the same algorithm used for the outer solver) to some inner precision ε_{in} with the aid of low-precision arithmetics.[18] Then, the iterative solution x_n is updated using high-precision operations, and the true residual vector is calculated in high-precision, as well. Although easy to implement, the defect-correction method has a big disadvantage: whenever an iterative algorithm is used for the inner solve, the Krylov subspace that has previously been calculated is discarded after each and every iteration. This may result in such an increase in the number of total iterations that the solver converges slower than it had without preconditioning.

A solution to this problem may be achieved as follows. Using low-precision arithmetics with one single solver will lead to rounding errors that will eventually destroy the precious orthogonality relations that make iterative solvers so powerful. That is, in order not to have to restart the inner solver (the low-precision part), it has to be corrected every so often with high-precision operations (the *reliable updates*). In particular, one has to replace the iterative residual with the true one as to rectify the numerical drift, and accumulate the iterative result vector into a high-precision buffer, in order to account for rounding errors. To guide the frequency of the high-precision updates one can test, if the norm of the current iterative residual has decrease by more than a factor δ with respect to the maximum of all previous residual norms. In order to guarantee convergence we obviously have to choose $\delta > \mathrm{ulp}_{in}$. On the other hand we want $\delta < 1$ in order to see benefits of the faster low-precision operations. In practical implementations δ may be chosen between 10^{-1} and 10^{-2} [22]. We summarise the full algorithm in Alg. 8.

1.3 Hybrid Monte Carlo

As we have mentioned already in the introduction, to calculate the integrals frequently appearing in LQCD, it is best to use Monte Carlo techniques, because they generically scale much better with the dimensionality of the integral in question than other techniques based on *quadrature*. In the former approach the integrand $f(x)$ is split into a part which is interpreted as a probability distribution $\mathcal{P}(x)$ and a remainder $g(x)$

$$I = \int \mathrm{d}^d x\; f(x) = \int \mathrm{d}^d x\; g(x)\,\mathcal{P}(x)\,. \tag{1.74}$$

[18]In order for the algorithm to converge, the spectral radius of the matrix $\rho(M) = \max(\lambda_1, \lambda_2, \ldots, \lambda_n)$ must be smaller than the *unit of least precision* used in the inner solve $\rho(M) < \mathrm{ulp}_{in}^{-1}$ and evidently $\varepsilon_{in} > \mathrm{ulp}_{in}$.

Algorithm 8: Reliable Updates

```
1  Variables with tilde represent low-precision types and operations.
   $x_0 = 0,\ r_0 = b,\ \tilde{x}_0 = 0,\ \tilde{r}_0 = r_0,\ n = 0$
2  for $m = 1, 2, \dots$ until $||r_n|| < \varepsilon$ do
3      Update $\tilde{x}_m \leftarrow \tilde{x}_{m-1}$ and $\tilde{r}_m \leftarrow \tilde{r}_{m-1}$ in low-precision
4      if $||\tilde{r}_n|| < \delta\,(\max_{k<n} ||\tilde{r}_k||)$ then
5          $x_n = x_{n-1} + \tilde{x}_m$ (accumulate in high-precision)
6          $r_n = b - Mx_n$ (calculate true residuum)
7          $\tilde{x}_m = 0$
8          $\tilde{r}_m = r_n$ (truncate)
9          $n \leftarrow n + 1$
10     end
11 end
```

Then, sampling a large number N of points $\{x\}$ which are distributed according to $\mathcal{P}(x)$, allows to approximate the above integral by calculating the arithmetic mean of the remainder function, evaluated on that set of points:

$$I \approx \sum_{\{x_i\}\in\mathcal{P}}^{N} g(x_i)\,. \tag{1.75}$$

Although the error of such an estimate decreases only $\propto N^{-1/2}$, it is by far the best known method for integrals of very large dimensionality.

In 1953 Metropolis et al [50] proposed an algorithm which can sample a *Markov Chain* of configurations x distributed according to any given $\mathcal{P}(x)$ by means of a random walk through configuration space. Its original domain of application was the calculation of statistical properties of many-body systems. At about the same time Alder and Wainwright [14] proposed an alternative algorithm which integrates the discretised Hamiltonian equations of motion of the molecular system in order to extract properties of its dynamics. About forty years later, Duane, Kennedy, Pendleton and Roweth unified these two ideas in their landmark paper [21] into the *Hybrid Monte Carlo* algorithm (HMC), in order to increase the acceptance rate of the Metropolis algorithm to speed up calculations in lattice field theories.

The main idea goes as follows: Instead of exploring the configuration space with a random work, a suitable Hamilton function H is derived from the probability distribution $\mathcal{P}$, such that new states are proposed according to their evolution under the Hamiltonian dynamics given by H. But since H is a conserved quantity under its own dynamics, exact integration will yield a new state which will be accepted with probability one. Although this is no longer true, when integrating discretised equations of motion using finite precision, the algorithm still gives much higher acceptance ratios and explores the configuration space much faster than the Metropolis Monte Carlo algorithm.

In this section, we want to introduce the basic HMC algorithm in full generality and give some details of its most important properties. To this end, we will briefly review Hamiltonian dynamics and its numerical integration first, then describe, how to derive a Hamiltonian from a given probability distribution, and to use the Metropolis accepted/reject step in order to account for inaccuracies of the numerical integration. Finally, we will outline how to adapt this algorithm to LQCD, particularly focusing on how the application and inversion of the fermion matrix and the *dslash* operator show up during the process.

Hamiltonian dynamics is an equivalent formulation of Newton's second law of classical mechanics, that uses position coordinates q_i and momenta p_i as fundamental variables, where $i = 1, 2, \dots, d$ and consists of $2d$ first order partial differential equations (*a.k.a.* equations of motion) instead of d second order partial differential equations in the Newtonian case. The dynamics can be derived from a Hamilton function or Hamiltonian $H(p, q)$ as follows

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = +\frac{\partial H}{\partial p_i}\,, \tag{1.76}$$

$$\frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial H}{\partial q_i}\,, \tag{1.77}$$

which can be combined in *symplectic* from as

$$\frac{\mathrm{d}z}{\mathrm{d}t} = J\,\nabla H(z)\,,\ \text{where } J = \begin{pmatrix} \mathbb{0}_d & \mathbb{1}_d \\ -\mathbb{1}_d & \mathbb{0}_d \end{pmatrix}\,. \tag{1.78}$$

Here, $z = (q, p)$ combines positions and momenta into one vector, and $\nabla H(z)$ is the gradient of H. Hamiltonian dynamics can then be viewed as a one-parameter linear transformation A_H^t: $\mathbb{R} \times \mathbb{R}^{2d} \mapsto \mathbb{R}^{2d}$ such that $A_H^t(p_0, q_0) = (p_t, q_t)$, mapping an initial state (p_0, q_0) at time t_0 into some time-evolved state (p_t, q_t) at time $t_0 + t$. In particular, this map has an inverse which is given by $(A_H^t)^{-1} = A_H^{-t}$. This is the same as to say, that Hamiltonian dynamics is time-reversible. Furthermore, it preserves the volume in phase-space, which is known as Liouville's theorem. This follows from the more general property of the Hamiltonian equations to be symplectic, *i.e.*

$$B^T J B = J\,, \tag{1.79}$$

where B is the Jacobian matrix of an arbitrary canonical transformation. This implies $\det^2 B = 1$, and, in turn, that the volume is preserved under the map A_H^t for any time t. The two properties of reversibility and volume preservation will turn out to be essential to satisfy so-called *detailed balance* for the Metropolis transition probability, to be discussed later.

The most important property of the Hamiltonian for the HMC algorithm however is the fact that it is conserved along trajectories of the Hamiltonian equations:[19]

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \sum_{i=1}^{d}\left(\frac{\partial H}{\partial p_i}\dot{p}_i + \frac{\partial H}{\partial q_i}\dot{q}_i\right) = \sum_{i=1}^{d}\left(-\frac{\partial H}{\partial p_i}\frac{\partial H}{\partial q_i} + \frac{\partial H}{\partial q_i}\frac{\partial H}{\partial p_i}\right) = 0\,. \tag{1.80}$$

The idea of HMC is now to apply the Metropolis Monte Carlo algorithm to a probability distribution $P' \propto e^{-H(p,q)}$. In this way, when evaluating the acceptance probability for a state $s' = (q', p')$ obtained by integrating the Hamiltonian equations from some initial state $s = (q, p)$, which is given by $T(s|s') = \min(1, e^{-H'+H})$, we always have $T(s'|s) = 1$, because $H(p, q)$ is preserved along the trajectory. Although, this is no longer true in finite precision, one can always correct for numerical inaccuracy by an accept/reject step. This Metropolis step makes the algorithm exact.

[19]Strictly speaking, we assume that the Hamiltonian does not have an explicit time dependence $\frac{\partial H}{\partial t} = 0$, but there exists a constant of motion in any case.

Clearly, in the above scheme one cannot set $\mathcal{P} = \mathcal{P}'$, because in this case one would sample only configurations with some given (initial) probability. Rather, one has to promote the random variables X in $\mathcal{P}(X)$ to the position variables q and introduced fiducial momentum variables p which will be sampled independently of q at the beginning of every Monte Carlo step. We will see this in more detail shortly.

First, however, let us have a look on how to integrate Hamilton's equations numerically. With out loss of generality, we can assume that the Hamilton function is separable into a potential $U(q)$ and a kinetic term $K(p)$

$$H(q,p) = U(q) + K(p)\,, \tag{1.81}$$

which we can always arrange for, in the algorithm. Then, we have to replace differential operators with finite differences to obtain a discretisation of the differential equations. The simplest such discretisation results from a Taylor expansion

$$p_i(t+\varepsilon) = p_i(t) + \varepsilon\,\frac{\mathrm{d}p_i}{\mathrm{d}t}(t) = p_i(t) - \varepsilon\,\frac{\partial U}{\partial q_i}(q_i(t))\,, \tag{1.82}$$

$$q_i(t+\varepsilon) = q_i(t) + \varepsilon\,\frac{\mathrm{d}q_i}{\mathrm{d}t}(t) = q_i(t) + \varepsilon\,\frac{\partial K}{\partial p_i}(p_i(t))\,, \tag{1.83}$$

where we used Hamilton's equations in the second step. This scheme is known as *Euler's method.* It is neither time-reversible nor symplectic. As a consequence, this method is not numerically stable.

It can however be modified by changing the step dependence within the second update, in the following way:

$$p_i(t+\varepsilon) = p_i(t) - \varepsilon\,\frac{\partial U}{\partial q_i}(q_i(t))\,, \tag{1.84}$$

$$q_i(t+\varepsilon) = q_i(t) + \varepsilon\,\frac{\partial K}{\partial p_i}(p_i(t+\varepsilon))\,. \tag{1.85}$$

This scheme is known as *semi-implicit* or *symplectic* Euler's method. It preserves volume, because it utilises only so-called *shear transformations* where in each step only one variable is changed by an amount proportional to the respective other variable.

The modified Euler's method is an example of a *first-order* symplectic integrator. This means, that each step of the trajectory has a *local error* of $\mathcal{O}(\varepsilon^2)$. It can be shown to accumulate to a *global error* of $\mathcal{O}(\varepsilon)$ after simulating τ/ε steps with length ε.

Examples of second-order symplectic integrators are the Verlet type integrators and in particular the *leapfrog method* in which updates of momenta and positions are staggered with half step-size interlacing. In this case, one step looks as follows:

$$p_i\left(t+\varepsilon/2\right) = p_i(t) - \frac{\varepsilon}{2}\,\frac{\partial U}{\partial q_i}(q_i(t))\,, \tag{1.86}$$

$$q_i(t+\varepsilon) = q_i(t) + \varepsilon\,\frac{\partial K}{\partial p_i}\left(p_i\left(t+\varepsilon/2\right)\right)\,, \tag{1.87}$$

$$p_i\left(t+\varepsilon\right) = p_i\left(t+\varepsilon/2\right) - \frac{\varepsilon}{2}\,\frac{\partial U}{\partial q_i}(q_i(t+\varepsilon))\,. \tag{1.88}$$

Note that the last half-step can be combined with the next one, forming a full step. That is, only the first and the last step are half steps. The only difference to the symplectic Euler's method is the fact, that positions are updated at full multiples of the step-size, whilst momenta are updated "in-between". The improved global error of $\mathcal{O}(\varepsilon^2)$ is a consequence of reversibility of the leapfrog algorithm [51]. There are also third- and fourth-order algorithms [31, 58].

Now, assume we want to sample a probability distribution of the form

$$\mathcal{P}(X) = \frac{1}{\mathcal{Z}}\, e^{-S(X)} \,, \tag{1.89}$$

where $\mathcal{Z}$ is some normalisation constant. This is the form of a canonical partition function, which we also find in the case of QCD. The first step consists in promoting the action $S(X)$ to the potential of the Hamiltonian $U(q) = S(X)|_{X=q}$. Next, we have to introduce an auxiliary set of momentum variables p. The most common practice is to choose a canonical kinetic term as the function $K(p)$

$$K(p) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i} \,, \tag{1.90}$$

where the "masses" m_i can be tuned for better convergence. In this way we have d independent random variables with Gaussian distribution, which have zero mean and variance m_i. We then introduce the Hamiltonian $H(q,p) = U(q) + K(p)$ and apply the Metropolis algorithm to the distribution[20]

$$\mathcal{P}'(q,p) = \frac{1}{\mathcal{Z}}\, e^{-H(p,q)} \,. \tag{1.92}$$

This works as follows: At the beginning of each step, we draw d independent Gaussian variables p_i. Then, we perform a reversible, volume-preserving (symplectic) integration with L steps of size ε according to Hamilton's equation with the above Hamiltonian H. Lastly, we calculate the resulting change ΔH and accept or reject the step with probability $\propto \min(1, e^{-\Delta H})$, which will correct for numerical errors, in case the integration drifts to far from the exact solution. We summarise the HMC algorithm in Alg 9.

Note, that the re-sampling of p_i at the beginning of each trajectory, *i.e.* each Monte Carlo step, is essential to prevent the sampling of configurations with only nearly constant probability. It allows to have large changes in term of the action S, maintaining the Hamiltonian H approximately constant, and thus the acceptance rate high. This allows for a much faster and more efficient exploration of the phase space as compared to the Metropolis random work. In this way, the momentum variables do resemble some sort of physical momenta, in that high kinetic energy at the beginning of a trajectory will result in a final configuration, which is far away form the initial one, with respect to the action (or probability).

What is left to show, is that the HMC algorithm indeed leaves the canonical distribution function $\mathcal{P}(q,p)$ invariant. To this end, we will first show that *detailed balance* is satisfied. This is the case, whenever the probability of passing from one state s to another state s' is the same as passing from s' to s

$$\mathcal{P}(s)\, T(s|s') = \mathcal{P}(s')\, T(s'|s) \,,\ \forall s, s' \tag{1.93}$$

[20]Note, that since $\int \mathrm{d}^d p\, e^{-K(p)} = \prod_{i=1}^{d} (2\pi m_i)^{1/2}$ we have

$$\mathcal{P}(X)\big|_{X=q} = \int \mathrm{d}^d p\, \mathcal{P}'(q,p) + \text{const.} \tag{1.91}$$

The constant can be absorbed into a redefining of $\mathcal{Z}$, and we will thus drop the prime in what follows.

```
Algorithm 9: Generic Hybrid Monte Carlo Algorithm
    input  : Distribution $\mathcal{P}(X)$, desired number of states $N$
    output: $N$ states distributed according to $\mathcal{P}$
1 Introduce position & momentum variables $p$, $q$
2 Introduce potential & kinetic energy $K(p), U(q)$
3 Re-write distribution as $\mathcal{P} = \frac{1}{\mathcal{Z}} \exp(-H(p,q))$
4 for $n \leftarrow 1$ to $N$ do
5     Generate initial momentum normally distributed
6     Integrate Hamiltonian Dynamics with symplectic integrator
7     Accept/reject with probability $\min(1, e^{-H_{new}+H_{old}})$
8     Store state vector $q$
9 end
```

where $T(s|s')$ is the transition kernel. To see this, one can partition the phase space into small subsets $\{S_i\}$ such that the image of every S_i under the Hamiltonian dynamics is S'_i. Then $\{S'_i\}$ will also form a partition of phase space, because this map is bijective, due to reversibility and volume-preservation. We are left to show that

$$\mathcal{P}(S_i)\, T(S_i|S'_j) = \mathcal{P}(S'_j)\, T(S'_j|S_i)\,. \tag{1.94}$$

This is satisfied by construction for any $i \neq j$. Otherwise is can easily been seen, that the above choice of the transition kernel $T(S|S') = \min\left(1, e^{-H(S')+H(S)}\right)$ leads to an identity

$$\frac{1}{\mathcal{Z}}\, e^{-H(S)} \min\left(1, e^{-H(S')+H(S)}\right) = \frac{1}{\mathcal{Z}}\, e^{-H(S')} \min\left(1, e^{-H(S)+H(S')}\right). \tag{1.95}$$

Finally, let S_k be canonically distributed. We want to show that its image after the accept/reject step is still canonically distributed. To this end we have to sum the probability for the states to transition to any other state (which may be the same) or to be rejected $R(S_k)$.

$$\begin{aligned}
&\mathcal{P}(S_k)R(S_k) + \sum_i \mathcal{P}(S'_i)\, T(S_k|S'_i) \\
&= \mathcal{P}(S_k)R(S_k) + \sum_i \mathcal{P}(S_k)\, T(S'_i|S_k) \\
&= \mathcal{P}(S_k)R(S_k) + \mathcal{P}(S_k) \sum_i T(S'_i|S_k) \\
&= \mathcal{P}(S_k)R(S_k) + \mathcal{P}(S_k)(1 - R(S_k)) \\
&= \mathcal{P}(S_k)\,,
\end{aligned} \tag{1.96}$$

where detailed balance was of crucial importance in the first step.

Apart from the Markov chain being reversible—which is the case when detailed balance is satisfied—it is crucial for any Monte Carlo algorithm to be *ergodic*. This roughly translates into the requirement that any state in configuration space must be reachable in a finite number of Monte

Carlo steps from any other state. It is a well-known fact, that generic Hamiltonian dynamical systems are neither ergodic nor even integrable [48]. However, the obstacle of ergodicity in practical applications is that trajectories may end up being exactly periodic, such that the evolution gets stuck in a subset of states. One solution to this problem is to randomly change both the number of steps L as well as the step size ε within small intervals occasionally [47].

For the later presentation of the computational costs of LQCD, let us state the scaling behaviour of HMC in terms of the dimensionality of the phase space d as well as the optimal acceptance rate. A more detailed discussion may be found in [51] and references therein. It is well known, that the Metropolis algorithm scales asymptotically with d^2 which is far superior to any method of quadrature. However, the HMC algorithm can be shown to scale even better, namely with $d^{5/4}$ [24]. Furthermore, as we pointed out before it has a much higher ideal acceptance rate than the Metropolis algorithm. One can show, that the ideal rate of acceptance for Metropolis is 23%, whereas it is 65% for the HMC algorithm [51].

Let us now turn to the case of LQCD. We have already seen the kind of integrals one is interested in calculating. In particular, the partition function takes the form

$$\mathcal{Z} = \int \mathcal{D}\psi\, \mathcal{D}\bar{\psi}\, \mathcal{D}A\; e^{-S_G[A]-S_F[\psi,\bar{\psi},A]}\,, \tag{1.97}$$

where it is possible to integrate out the fermion fields analytically. Using the property that fermion fields anti-commute with each other (they are *Grassmann valued*) one can show that the Gaussian integral of the fermion part of the action is proportional to its determinant. For two flavours (*up* and *down*) one can thus rewrite the partition function as

$$\mathcal{Z} = \int \mathcal{D}A\; e^{-S_G[A]} \det(M_u)\det(M_d)\,. \tag{1.98}$$

In order to interpret the integrand as a probability distribution, it has to be real and non-negative. Since the fermion matrix M can be shown to be γ_5-hermitian, *i.e.* $M^\dagger = \gamma_5 M \gamma_5$, reality follows:

$$(\det M)^* = (\det M^T)^* = \det M^\dagger = \det(\gamma_5 M \gamma_5) = \det M\,, \tag{1.99}$$

because of $\gamma_5^2 = \mathbb{1}$. The simplest possibility to guarantee positivity is to use pairs of degenerate fermions (here $M_u = M_d = M$) which for later convenience we can write as

$$0 \leq (\det M)^2 = (\det M)(\det M^\dagger) = \det(MM^\dagger)\,. \tag{1.100}$$

The trick is now, to rewrite the determinant as another Gaussian integral, although one of bosonic (that is *commuting*) fields. Using the identity $\det A = \frac{1}{\det A^{-1}}$ for any invertible matrix A, we can write

$$\mathcal{Z} = \int \mathcal{D}A\; e^{-S_G[A]} \int \mathcal{D}\phi^\dagger\, \mathcal{D}\phi\; e^{-\phi^\dagger (MM^\dagger)^{-1}\phi}\,, \tag{1.101}$$

where ϕ is a complex-valued scalar field that carrier the same index structure as the fermion fields ψ. It is called a *pseudo-fermion field*, because it behaves like a fermion.

Note, that the fermion matrix M is a functional of the gauge fields A_μ. As a consequence, even though the gauge bosonic part of the action is *ultra-local*, *i.e.* all terms in the action only consist of products of neighbouring links, the fermionic part is not. Quite on the contrary, because of the appearance of the inverse of the fermion matrix, it will involve all kinds of products of links that are very far apart. That does not only make it more expensive to evaluate the difference of the action after proposing a Monte Carlo move, it also makes it very unlikely, that a configuration reached by random work will be accepted during the Metropolis step. This is why we want to use the HMC algorithm to sample a distribution

$$\mathcal{P}_{QCD} \propto e^{-S_G[A]-\phi^\dagger(MM^\dagger)^{-1}\phi}\,. \tag{1.102}$$

As a first step, one samples the fermions by themselves. This is done be rewriting $M^{-1}\phi = \chi$. Then one samples the Gaussian $e^{-\chi^\dagger\chi}$ and obtains the pseudo-fermions by calculating $\phi = M\chi$.

Then we have to propose a new configuration of the gauge links by integrating the Hamiltonian dynamics. This is best done by using the lattice gauge fields as position variables $q_\mu(n) = A_\mu(n)$. They, however, live in the algebra $\mathfrak{su}(3)$ as mentioned earlier and need to be exponentiated in order to obtain link variables,

$$U_\mu(n) = \exp\left(i\sum_{i=1}^{8}\omega_i T_i\right). \tag{1.103}$$

Here the ω_i are the eight real coefficients of an algebra element and the T_i the generators, which are traceless, hermitian 3×3 matrices. Then, also the conjugate momentum variables $p_\mu(n)$ will be traceless, hermitian matrices and we will use the kinetic function

$$K(p_\mu) = \frac{1}{2}\sum_{n,\mu,i}\left(p_\mu^{(i)}(n)\right)^2 = \sum_{n,\mu}\operatorname{Tr} p_\mu^2(n)\,. \tag{1.104}$$

Hence as the second step, one samples the momentum variables according to their Gaussian distribution given by $e^{-\operatorname{Tr} p^2}$. Finally, we have to integrate the equations of motion for the Hamilton function $H(q,p) = S_G(q) + \phi^\dagger\left(MM^\dagger(q)\right)^{-1}\phi + \operatorname{Tr} p^2$. The most involved part thereof is the evaluation of the force term $F[U,\phi] = \frac{\partial H}{\partial q}$ which assumes the following form:

$$F[U,\phi] = \sum_{i=1}^{8} T_i\,\nabla^{(i)}\left(S_G[U] + \phi^\dagger(MM^\dagger)^{-1}\phi\right). \tag{1.105}$$

It is again an element of $\mathfrak{su}(3)$. The fermionic contribution will read [35]

$$\nabla^{(i)}\left(\phi^\dagger(MM^\dagger)^{-1}\phi\right) = -\xi^\dagger\left(\frac{\partial M}{\partial\omega_i}M^\dagger + M\frac{\partial M^\dagger}{\partial\omega_i}\right)\xi\,, \tag{1.106}$$

$$\text{with }\ \xi = (MM^\dagger)^{-1}\phi\,. \tag{1.107}$$

The derivatives of the fermion matrix are very similar to the *dslash* operator itself and read in the case of Wilson fermions

$$\frac{\partial M(n,m)}{\partial\omega_i^\mu(k)} = -\frac{i}{2a}(1-\gamma_\mu)T_iU_\mu(k)\delta_{n+\hat\mu,m}\delta_{n,k} + \frac{i}{2a}(1+\gamma_\mu)U_\mu(k)^\dagger T_i\delta_{n-\hat\mu,m}\delta_{m,k}\,. \tag{1.108}$$

Algorithm 10: HMC Algorithm for LQCD with Leapfrog Integration

```
input  : Initial gauge configuration $U_0$, force functions $F[U,\phi]$
output : New gauge configuration $U_n$
1  Generate normally distributed random spinor field $\chi$ and calculate pseudo-fermions field
     $\phi = M\chi$
2  Sample initial momentum $p_0$ configuration according to $e^{-\mathrm{Tr}\,p^2}$
3  Initial step:
4      $p_{\frac{1}{2}} = p_0 - \frac{\varepsilon}{2} F[U,\phi]\big|_{U_0}$
5  foreach intermediate step $k \leftarrow 1$ to $n-1$ do
6      $U_k = \exp\left(i\,\varepsilon\, p_{k-\frac{1}{2}}\right) U_{k-1}$
7      $p_{k+\frac{1}{2}} = p_{k-\frac{1}{2}} - \varepsilon\, F[U,\phi]\big|_{U_k}$
8  end
9  Final step:
10     $U_n = \exp\left(i\,\varepsilon\, p_{n-\frac{1}{2}}\right) U_{n-1}$
11     $p_n = p_{n-\frac{1}{2}} - \frac{\varepsilon}{2} F[U,\phi]\big|_{U_n}$
12 Accept when uniform random number $r \in [0,1)$ is smaller than
     $\exp\left[\mathrm{Tr}\,p^2 - \mathrm{Tr}\,p'^2 + S_G[U] - S_G[U'] + \phi^\dagger\left((MM^\dagger)^{-1} - (M'M'^\dagger)^{-1}\right)\phi\right]$
```

The computation of the fermion forces is the most expensive part of the whole HMC algorithm. In particular, after every update of the gauge fields, the fermion matrix M changes. The update of the momentum fields however, requires the inversion of the matrix $MM^\dagger$ by means of an iterative solver. The typical number of time steps per trajectory is of $\mathcal{O}(100)$. That is, during every Monte Carlo step, one needs to perform a number of inversions of the same order. We summarise the HMC algorithm for the special case of LQCD with leapfrog integration in Alg. 10.

1.4 Computational Complexity of LQCD

After having introduced all basic tools to carry out a simulation in lattice QCD, we can make an estimate of its computation complexity. As mentioned previously, the fermion matrix M is a square matrix with side length roughly given by the number of lattice sites. Thus, it is a sparse matrix of size $(L/a)^4 \times (L/a)^4$. Here, L is the physical length of one lattice dimension, and a is the lattice spacing. To make physically sensible calculations, L should be much larger than the largest length scale of the dynamics, which is given by the inverse of the pion mass m_π^{-1}. Similarly, a should be much smaller than the smallest dynamical length scale, which is given by Λ_{QCD}^{-1}

$$L \gg m_\pi^{-1} \sim 10^{-15}\,\mathrm{m}\,, \tag{1.109}$$

$$a \ll \Lambda_{QCD}^{-1} \sim 10^{-16}\,\mathrm{m}\,. \tag{1.110}$$

Hence, we should ideally have $L/a \gtrsim 100$, which results in M being $10^8 \times 10^8$.

The fermion matrix is of the form $M = am_f\mathbb{1} + \not{D}$ and so the smallest and largest eigenvalues of the product matrix $MM^\dagger$, relevant for the HMC algorithm, can be estimated as

$$\lambda_{min} \sim (am_f)^2 , \quad \lambda_{max} \in \mathcal{O}(10) . \tag{1.111}$$

For the simplest case of two degenerate quarks we have $m_{u,d}/\Lambda_{QCD} \sim 10^{-2}$, which together with the above requirement, leads to $am_f \lesssim 10^{-3}$.

Recalling that the HMC algorithm scales with $d^{5/4}$, where d is the dimensionality of the distribution to be sampled, and the fact that it scales with the number of lattice sites $d \sim V$, we can estimate the scaling in terms of the lattice extent as L^5. Taking into account, that the number of iterations of the solver scales with the condition number $k = \lambda_{max}/\lambda_{min}$, we arrive at the following estimate for the computational cost of LQCD, cf. Ukawa [76] from 2001:

$$2.8 \left(\frac{\#\text{conf}}{1000}\right) \left(\frac{m_\pi/m_\rho}{0.6}\right)^{-6} \left(\frac{L}{3\,\text{fm}}\right)^{5} \left(\frac{a}{2\,\text{GeV}}\right)^{-7} . \tag{1.112}$$

With this estimate, the costs to obtain 100 well-decorrelated configurations for a lattice with $L = 6\,\text{fm}$ and $a = 0.04\,\text{fm}$ would amount to approximately 10^7 TFlops $\cdot$ year $\sim 10^{26}$ ops.

Eleven years later, the same estimate was made by Schaefer [61], taking into account new algorithmic developments, like multi time-scale and higher order symplectic integrators during the molecular dynamics integration, preconditioning with Hasenbusch decomposition, and deflation. Then the above estimate reduces to approximately 10^3 TFlops $\cdot$ year $\sim 10^{22}$ ops, which is an improvement of four orders of magnitude.

Within the same time frame, there were similar improvements made in the domain of computer hardware. Judging from the front runner of the TOP500 [71], which was ASCI White in November 2001 with 12 TFlops and Titan Cray XK7 in November 2012 with 27 112 TFlops, this amounts to a factor of 2204.

But the history of lattice QCD is not short of developments in innovative computer hardware itself. Noteworthy are certainly the APE project (within the INFN collaboration), which started with a 250 MFlops machine in 1988 and featured the APEnext 10 TFlops machine in 2006 [15, 20], as well as the series of BlueGene computers which were developed by IBM in collaboration with lattice physicists like Peter Boyle [17, 36].

This page intentionally left blank.

2. Hardware

In this chapter, we want to introduce the target hardware of the QPhiX library, the Intel Xeon Phi processor/coprocessor family. We will in particular focus on the second generation product, with the code name *Knights Landing* (KNL), which is the successor of the coprocessor card *Knights Corner* (KNC). First announced in 2011, the Intel Xeon Phi family is a many integrated core architecture that hosts of the order of 60–70 CPU cores in one processor or coprocessors card, and can in many ways be considered as Intel's alternative to *General-purpose Graphics Processing Units* (GPGPUs or short GPUs).

With the end of the increasing CPU frequency era in the early 2000's, CPU manufacturers started to produce CPUs with more than one core and/or thread. However, *Moore's law* continued to be valid up to 2016 and so the increase of transistor counts was used to increase the number of cores inside one CPU. In turn, software had to use parallelism on several levels to make use of the improved capabilities of new hardware in order to extract performance.

This is in particular true for Xeon Phi processors, which offer a large amount of parallelism on the level of both single cores, *i.e. Single Instruction, Multiple Data* (SIMD), as well as on the level of threads in a shared memory environment. This is further complicated by the fact, that the gap between the throughput of CPUs (the number of floating point operations that can be executed per second) and the bandwidth of all layers of memory grows more and more.

From this considerations one finds three main points which have to be respected in order to extract performance out of (any) parallel environment. Software must scale up to many threads (that is as many as 288 for KNL) inside a shared memory system, it must make use of vectorisation in form of SIMD and it must minimise memory traffic through data locality/affinity, which will improve the reuse of data in the faster CPU-affine caches. To this end, one uses techniques such as cache blocking and tiling, loop unrolling and (multi-)loop interchange, as well as prefetching, unit-stride, memory-contiguous data alignment and the use of (arrays of) structure of arrays (ASA/SoA) to improve vectorisation efficiency. We will see many of these techniques in much more detail later on.

In this chapter we want to give an introduction to the specific architectural design of the KNL processor. As with the presentation of the algorithms in the last chapter, we want to present the different layers of the hardware starting from the lowest lying core and tile structure, then continuing to the interplay of cores and threads in form of the interconnecting mesh and the particulars of KNL's memory structure, and ending with some remarks about the internode features when connecting several processors or cards. With this knowledge in mind, we will present some refinements of the above optimisation considerations and present some software solutions/APIs which are available for multi-core hardware systems. We will also revisited the arithmetic intensity for the *dslash* operator to create a simple performance model taking hardware considerations into account.

2.1 Architecture

KNL is a processor with up to 72 cores which exists also as a coprocessor version, whilst KNC was only available as a PCIe card. It is structured in so-called tiles which posses two CPUs each. Each tile has an L2 cache of 1MB which is private to the tile, but shared amongst the two cores. The tiles are connected via a 2D mesh, as opposed to a 1D ring in the KNC version. In addition, KNL's are shipped with two sorts of memory. One high bandwidth memory called MCDRAM (multi-channel dynamic random access memory) of $16\,\mathrm{GB}$ size and a conventional DDR-4 memory with up to $2400\,\mathrm{MHz}$ frequency and $384\,\mathrm{GB}$ size. It can be connected to the network via an on-package interconnect called Intel Omni-Path Fabric which offers two ports with $100\,\mathrm{GB/s}$ bandwidth each. It also fully integrates a new Instruction Set Architecture (ISA) called AVX-512 which supports SIMD vectors of 512-bit size and most notably incorporates new hardware instructions to gather (load) and scatter (store) non-memory contiguous vectors efficiently. KNC instead was using the Intel Initial Many-core Instructions (IMCI) which are likely to be discontinued. Furthermore, KNL is binary compatible with all previous ISA's of Intel and in particular supports the legacy ISA's SSE, MMX, x87, AVX and AVX-2. This allows for a direct use of all C/C++ and Fortran features, as well as OpenMP and MPI APIs, which is a huge practical advantage compared to the situation for GPUs, in particular for Fortran programmers.

When designing a successor for the KNC, the main goal was to remove the offload overhead which introduces a considerable bottleneck given by the latency and bandwidth limitations of the PCIe bus. It was therefore necessary to design a stand-alone processor which would be able to boot an operating system and connect to the network directly. To achieve this four major design decisions had to be made [55]. First of all, the single thread performance had to be improved considerably, since all the code would be run on the same device. It should be able to execute instructions out-of-order (which KNC could not), but be power-efficient for parallel and vectorised code as well. Secondly, it should be binary compatible to all previous ISA, so that OSes, debuggers, infrastructure, legacy libraries *etc.* could be used, even without recompilation. Thirdly, it should have a much larger memory capacity than KNC, ideally something of the order of $5\,\mathrm{GB}$ per core, which is about the upper limit that certain HPC applications demand. But in addition it should posses a high-bandwidth memory (the MCDRAM) in order to be able to transfer enough data to and from the large vector registers. Fourth and lastly, it was necessary, to improve the tile interconnect in form of a 2D mesh to reduce latency and bandwidth, so caches could be kept coherent with velocities much faster than the already high speeds provided by the MCDRAM. The design of the KNL is depicted in Fig. 2.1.

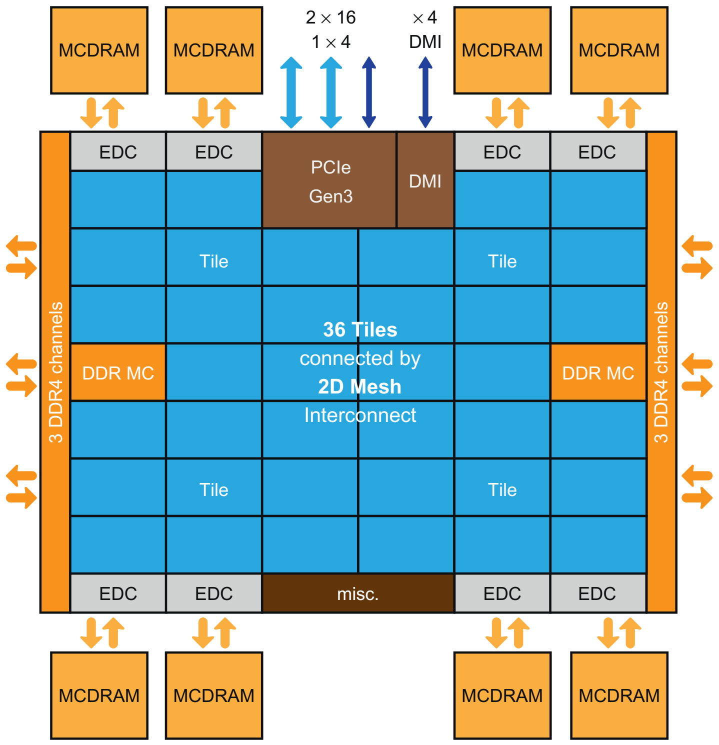

Figure 2.1: Schematic layout of the Xeon Phi Knights Landing processor. Reproduced from [55].

2.1.1 Tiles and Cores

The tiles are the basic building blocks of the KNL (Fig. 2.2). They consist of two cores with two vector processing units (VPUs) each and a 1MB shared L2 cache. They also host the Bus Interface Unit (BIU) which controls cache coherency of the L2 inside the tile, as well as the Caching-Home Agent (CHA) which stores a part of the distributed cache line hash, which is used to keep the L2 cache coherent over the whole tile mesh. The CHA is also part of the on-die interconnect mesh itself, which we will revisit shortly.

Each core is a 2-wide (that is it can execute two instructions per cycle) out-of-order core which was derived from the Intel Atom core with codename Silvermont. However, many new features have been added. In particular it has four hardware hyper-threads (also called simultaneous multi-threads (SMT)) which help to hide instruction and cache latencies via the out-of-order execution. It has the double number of inflight instructions and deeper out-of-order buffers (the Reorder Buffer (ROB) has 72 slots). It also has larger and faster L1 and L2 caches. In particular the L1 data cache, which is private to the core has a size of 32 kB. It posses two additional 64 B load ports to feed the two VPUs each core owns. They are themselves tightly integrated into the out-of-order pipeline. In addition, the L2 bandwidth was doubled, and each tile can now read 1 cache line and write 1/2 cache line per cycle. The full bandwidth may even be used by one core alone, in case the other one is busy otherwise. Cache lines have a size of 64 B. Furthermore, the translation lookaside buffer (TLB), which is a cache in which virtual page addresses are translated to physical one, was increased and support for 1 GB pages was added (16 entries, in order to cover the whole of MCDRAM). Lastly, gather and scatter facilities are integrated into the hardware directly, in order not to use fetch and decode slots in the pipeline.

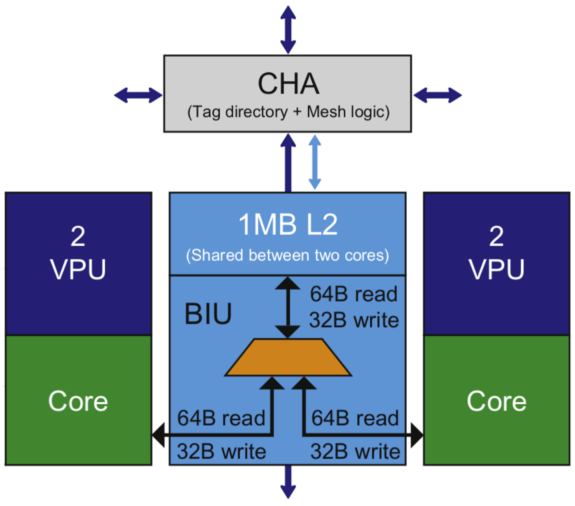

Figure 2.2: Schematic structure of a KNL tile. Reproduced from [55].

KNL is also the first hardware to implement the new AVX-512 ISA, which is announced to be supported also in the Skylake microarchitecture. It has full 512-bit SIMD support and uses 32 logical registers (ZMM0-31) which can be mapped to 256-bit or 128-bit registers (YMM0-31 and XMM0-31) for AVX-2 and AVX support, respectively. In addition, it has 8 mask registers (k0-k7) for vector predication. The instructions are separated into four categories. The first one, foundations AVX-512F, includes all the basic floating point operations, notably the fused-multiply-add (FMA), which have mostly a 6-cycle latency. The second category, conflict detection AVX-512CD, contains functionality that ensures correctness for gather-modify-scatter type sequences of instruction. They are for instance relevant when one updates arrays at index positions, that have to be looked up with the help of some other array (*e.g.* lookup tables for space-filling curves etc.). For example, they contain instructions that detect duplicates within a SIMD vector and reduce it to conflict-free subsets. The third set of instructions are special reciprocal and exponential operations AVX-512ER, which compute approximate exponentials, inverses and inverse square roots to a precision of at least 2^{-23}. The last set are prefetch instructions AVX-512PF, which include operations for software prefetches to both the L1 as well as the L2 caches.

The core itself consists of five units: the front-end unit (FEU), the allocation unit (AU), the integer execution unit (IEU), the memory execution unit (MEU) and the vector processing unit (VPU), cf. Fig. 2.3. In the FEU, instructions are fetched and decoded, branches predicted and all instructions are split in so-called micro-operations (μops) which have a 1-cycle latency. This is done in order to be able to implement complicated instructions from simple hardware facilities. In the AU, μops are prepared for out-of-order execution and resources that are needed are allocated here. These include ROB, and Rename Buffer (RB) entries, Store Data buffers, as well as gather/scatter entry tables. Finally, each μop is sent to either the Integer, Memory or Vector Processing. The IEU operates on general purpose registers R0-15, and most μops have a 1-cycle latency. However, there are important operations (e.g. multiply) with 3–5 cycle latencies. Each core possesses two IEU's. The MEU, which is also 2-wide, is responsible to fetch requests in case of the instruction cache and ITLB misses. It supports unaligned memory accesses without penalty and accesses to data, that is split between two cache lines with a 2-cycle penalty. It also hosts the gather/scatter logic.

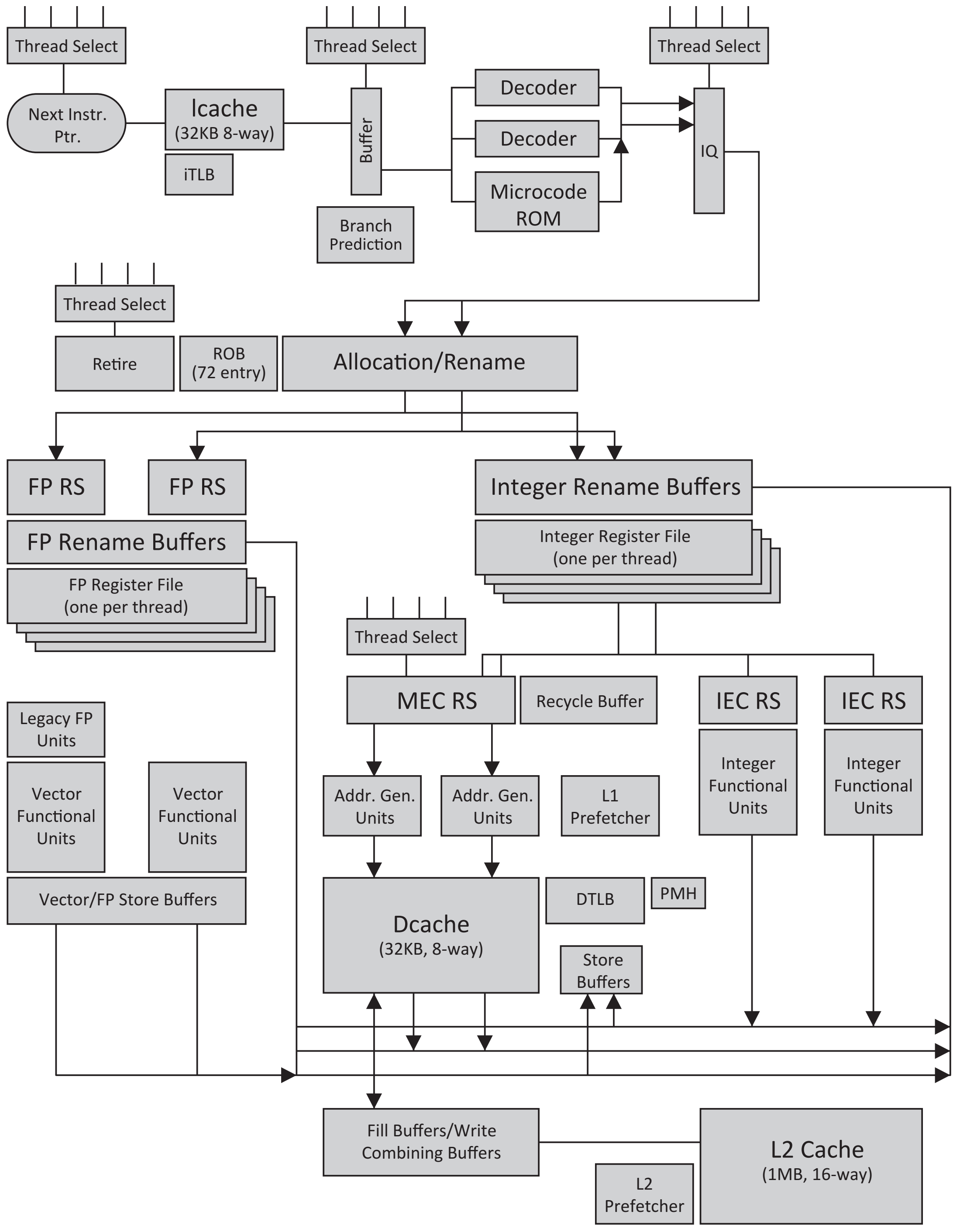

Figure 2.3: Block diagram of a KNL core. Reproduced from [55].

Finally, there are two VPUs per core which have 20 entries in their reservation station, which can be executed out-of-order with 1 μop per clock. In total, the core is 2-wide in decoding, allocating and retiring, but can execute up to 6 instructions of integer, floating point and memory operations (each unit being 2-wide as stated above) per clock.

There are some remarks about the hardware threads in order. All instructions for up to four threads flow through the same pipeline simultaneously. There are several points where threads select which instructions is moving forward in the pipeline (cf. Fig 2.3). This decision is made every single cycle and has a heuristic scheme, based on resource availability, fairness and stalling of threads. Core structures in the pipeline are shared, dynamically partitioned or duplicated amongst threads. In contrary to KNC, one KNL hardware thread can achieve maximal performance, but two or four threads often perform better, because they tend to hide latencies more efficiently. However, linear scaling of an application in the number of hyper-threads is very unlikely to be achieved.

Intel cites the peak performance of a 72 core KNL with over $3\,\mathrm{TFLOP/s}$ in double and over $6\,\mathrm{TFLOP/s}$ in single precision [55]. This is, assuming every clock the processor is executing one FMA, which is worth 2 floating point operations for 8 doubles or 16 floates, in each and every VPU. With a core frequency of $1.5\,\mathrm{GHz}$ we have

$$72 \times 2 \times (2 \times 8/16) \times 1.5 \times 10^9\,\mathrm{Hz} \approx 3.46/6.91\,\mathrm{TFLOP/s}$$

for double and single precision, respectively.

2.1.2 Interconnect Mesh and Cluster Modes

To connect the tiles of the KNL, a 2D cache-coherent interconnect mesh is used, which outperforms the 1D ring that was used in KNC by far. In order to keep the L2 cache coherent amongst all tiles of the processor, a MESIF protocol is used. It is based on a distributed tag directory, of which every tile keeps a local part in its CHA. The distribution is based on an address hash which varies depending on the cluster modes which is used. MESIF is an acronym for the five states, a given cache line may be in: Modified, Exclusive, Shared, Invalid and Forward. When issuing a request through the mesh, YX routing is used to reduce the possibility of deadlocks, caused by travelling messages. This means that messages travel vertically first, which consumes 1 clock cycle, and then horizontally, which takes 2 cycles. The aggregated bandwidth of the 2D mesh amounts to $700\,\mathrm{GB/s}$ which is about double the peak bandwidth of the MCDRAM, and which, as we will see, may be used as a high level cache.

In order to keep the communication as local as possible on the mesh, there are different so-called cluster modes, one can choose from at boot time. They divide the chip into virtual regions in different ways, so that cache messages travel only inside these regions. This reduces latency which will effectively increase the bandwidth, because hardware buffers are freed sooner and more messages can be issued in a given time frame. There are three major modes available: the all-to-all mode, the quadrant mode and the Sub-NUMA Cluster (SNC) mode. To understand the difference between these modes, one has to look how messages typically travel in case of an L2 miss. There are three main points involved in this process. The tile/core that generates the L2 cache line miss, the CHA that owns the address of this cache line in its local CHA dictionary, and finally the part of the MCD or DDR-RAM that carries the data.

The all-to-all mode is the one lowest in general performance, but it is the most general one, in that it can be used even if the DIMMs of the DDR are equipped non-symmetrically in capacity. In this mode, addresses are hashed uniformly over the whole chip, so that a request may travel from

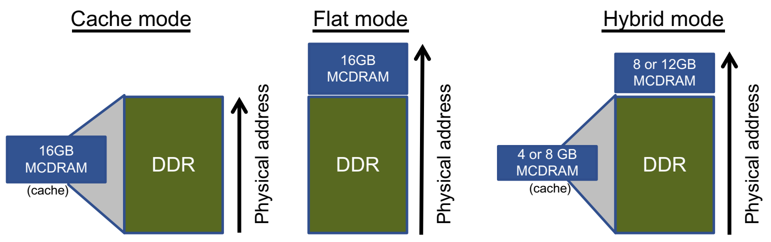

Figure 2.4: Structure of the memory modes which can be chosen at boot time. Reproduced from [55].

any tile to any other inside the entire processor, and the data may reside in any part of the memory. In the quadrant mode, which is the default mode, the chip is divided into four virtual parts in order to establish vicinity between the CHA and the memory. In this mode, the address of cache lines are hashed in such a way that CHA's carry only those addresses, that are physically located in the memory directly connected to the respective quadrant. This will reduce latency and thus increase performance without requiring any NUMA-awareness of the software. Finally, there is the SNC-4 mode, in which additional affinity of the tiles and the CHA's is established. Namely, each of the four quadrants is now promoted to a separate cache-coherent NUMA sub-cluster. That is, addresses are hashed only over a given quadrant and point only to memory located in that quadrant. This effectively subdivides (each type of) memory into four (or eight in case of addressable MCDRAM, see below) NUMA clusters. Although this mode has the smallest latency and highest bandwidth, it requires the software to by NUMA-aware in order to extract maximal performance. Note however, that any software will run in every cluster mode, and cache-coherence is always maintained over the entire chip. Different levels of affinity may however influence the performance significantly, depending on the hardware awareness of the software.

2.1.3 Memory and Memory Modes

KNL ships with two kinds of memory. A relatively small high-bandwidth memory (HBM) which is integrated on-package, and an ordinary high capacity, small bandwidth, extendable DDR type of memory, which is placed outside the package.

The MCDRAM HBM comes in 8 portions of 2 GB, each with its own memory controller (called EDC). It has an aggregated bandwidth of about 450 GB/s. It can be used either as addressable memory or as an additional high level memory-affine (as opposed to the usual low level CPU-affine) cache. The DDR-4 memory on the other hand has two dedicated memory controllers, each with three channels. In each channel one DIMM can be placed, having frequencies up to 2400 MHz. Each DIMM may have a size of up to 64 GB, making a total of $6 * 64\,\mathrm{GB} = 384\,\mathrm{GB}$. The aggregated bandwidth amounts to 90 GB/s.

These two types of memory may now be combined in three different modes, called cache, flat and hybrid (Fig 2.4). In cache mode, the entire 16 GB of MCDRAM are used as a global coherent cache to the DDR memory. It can be thought of as a giant, relatively CPU-far L3 cache, because it has the same latency as the DDR-4 memory. This is the default mode and will be beneficial for most applications, as long as they do not either block very efficiently into the L2 with a high rate of data reuse, or stream over larger than 16 GB parts of memory frequently. In this mode, the cache is invisible to software (including the OS) and is managed exclusively by the hardware.

In the flat mode, the entire MCDRAM is exposed to the OS as an addressable NUMA cluster of ordinary memory. Its NUMA distance is set higher than the one of the DDR, so that the OS does

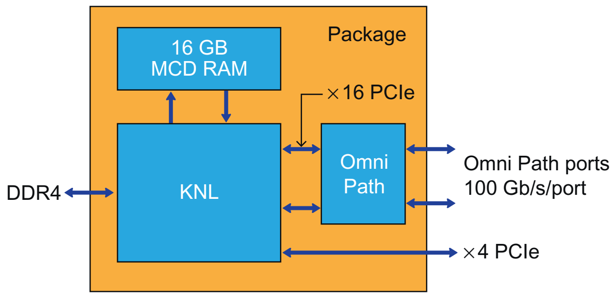

Figure 2.5: KNL package with chip, MCDRAM and Omni-Path Fabric. Reproduced from [55].

not preferably allocate into this memory. It is fully manageable by the software, and in particular provides an allocatable heap to the applications.

Hybrid mode, finally, is a combination of cache and flat mode. In this mode 25% or 50% of the HMB may be used as cache, whereas the rest remains fully addressable. This mode is in particular useful to help different kinds of software to benefit from the high-bandwidth MCDRAM.

On a practical level, it seems to be rather restrictive that both the cluster modes as well as the memory modes have to be chosen at boot time. This is because, often HPC users do not have the (root) rights to reboot the processors and enter the BIOS. This however seems to be a problem mostly caused by the fact that software, namely the OSes, can not yet deal with the possibility of hardware changes during up-time (think new tables for memory tags, routing and caching, which are normally generated during boot time). Since this is a rather new phenomenon, software is likely to adapt in the future.

2.1.4 I/O and Omni-Path Fabric

For inter-node communication and I/O the KNL processor provides a total of 36 PCIe Gen3 lanes (PCIe Root Ports) which are split into two 16x and one 4x lane. Some versions of KNL also integrate an on-package Intel Omni-Path Fabric which is connected to the die via the two 16x lanes and provides two ports out of the package with bandwidths of 100 GB/s each (Fig. 2.5).

Omni-Path is a communication architecture for HPC that is supposed to offer low latency, low power consumption and high overall bandwidth. It aims to provide tight coupling to CPU, memory and storage resources and high scalability up to the exa-scale.

2.2 General Programming Implications

We have already seen in the introduction that there are generally three points which have to be optimised in order to extract performance from the Xeon Phi processors. That is, the software has to be scalable up to many threads, it has to make use of the large vector registers using SIMD instructions (at least if the application is not memory bandwidth bound, up to the point where the bandwidth is saturated already for scalar instructions) and it has to optimise data locality and cache reuse.

In this section, we want to briefly introduce the features offered by the KNL architecture to incorporate the above three points. In particular, we will review some high level solutions concerning the software management of the MCDRAM memory, alternative APIs for parallel programming in shared memory environments, and finally some details about the deliberate use of SIMD instructions for vectorisation.

2.2.1 Managing MCDRAM

When using KNL in cache memory mode, the HBM is entirely managed by the hardware and invisible to the OS. However, in both flat and hybrid mode, the MCDRAM is addressable and can be used for dynamical allocations. The programmer has essentially three tools to choose from when managing the MCDRAM manually. He can either use the NUMA control utility (numactl), the autohbw library, or the more general and powerful memkind library [4].

numactl can be simply used to allocate all data that is managed inside the software (including the stack and data segments) to the MCDRAM. This is independent of the programming language and does not require changes to already existing software. However, it does not allow for a heap that is split between different kinds of memory or a fine-grained control (and exception handling) for different sorts of memory.

This issue is addressed by the memkind library, which is an open-source user extensible heap manager for a large class of different memory types (including DRAM, MMIO, RDMA, slabs, symmetric heaps). It is based on jemalloc, which provides a general purpose malloc facility and particularly aims towards fragmentation reduction and scalable concurrency support [2]. memkind provides two APIs, one of which (hbwmalloc.h) is considered more stable and can be used by calling the usual C malloc routines with prefix `hbw_`. The second one (memkind.h) is very powerful, but still under development. Amongst others, it allows to define customised memory types, type dependent allocation routines and explicit support for huge pages at allocation time (*e.g.* 2 MB instead of the usual 4 kB). It can also easily be used to overload the `operator new` in C++ in order to hide the implementation details in the high level code. We give a minimalistic example in Lst. 2.1, which is adapted from the memkind project.

The autohbw API finally is implemented using the memkind library and relies on the standard GNU C library (glibc) allocation routines. Thus, it can be used with Fortran, C and C++. It allows to allocate data in a certain range of size to be automatically allocated to MCDRAM. It does not require code changes, and only has to be (dynamically with `LD_PRELOAD` or statically) linked. The stack and data segments are not allocated on the HBM.

2.2.2 Vectorisation

Parallelism on the level of single CPU cores is becoming increasingly important. A vector register on both KNC and KNL has a size of 512-bit, allowing for the parallel SIMD execution of 8 doubles or 16 floats, which will cause an enormous loss in the area of HPC when not employed (efficiently). This issue will become even more severe, because 512-bit large vector registers will enter the Xeon product line via the Skylake microarchitecture [10], and are likely to grow even further to 1024-bit in the future [9].

On the other hand, however, SIMD is the area of parallel programming with the least high and low level API support for software developers. Intel tends to suggest three main strategies to utilise vectorisation for the majority of cases [55]. These are the usage of SIMD efficient libraries, compiler auto vectorisation and guided vectorisation via the use of `#pragma`'s. The latter including the openMP 4.0 `#pragma SIMD` and compiler specific directives like `#pragma ivdep`.

There are (at least) two reason why this is very unsatisfactory in practice. First of all, it does not allow to access all the facilities of the ISA. In particular, shuffle, gather&scatter, masking and more complex floating point operations cannot be handled by the programmer directly. Secondly, the necessity of having to adjust the data structures for the efficient use of SIMD, interferes with good software design. In particular, with scientific software growing to several million lines of source

```
#include <memkind.h>
#include <cstdlib>
#include <new>

template <class deriving_class>
class memkind_allocated
{
  public:
    static memkind_t getClassKind()
    {
      return MEMKIND_DEFAULT;
    }

    void* operator new(std::size_t size)
    {
      return deriving_class::operator new(
          size,
          deriving_class::getClassKind() );
    }

    void* operator new(std::size_t size, memkind_t memory_kind)
    {
      void* result_ptr = NULL;
      int allocation_result = 0;

      // This checks if deriving_class has specified alignment,
      // which is suitable to be used with posix_memalign()
      if(alignof(deriving_class) <  sizeof(void *)) {
        result_ptr = memkind_malloc(memory_kind, size);
        allocation_result = result_ptr ? 1 : 0;
      }
      else {
        allocation_result = memkind_posix_memalign(
            memory_kind,
            &result_ptr,
            alignof(deriving_class),
            size);
      }

      if(allocation_result) {
        throw std::bad_alloc();
      }

      return result_ptr;
    }
};
```

Listing 2.1: Minimal example, how to overload the operator new using memkind functionality

```
#include <sdlt/sdlt.h>

struct YourStruct
{
  double x, y, z;
};

SDLT_PRIMITIVE(YourStruct, x, y, z);
typedef sdlt::soa1d_container <YourStruct> YourContainer;

YourContainer inputContainer(count);
YourContainer::accessor<> input = inputContainer.access();
```

Listing 2.2: Minimal example, how to use Intel SDLT containers.

code, separation of algorithms from data structures and low level data manipulations becomes increasingly important.

There is an interesting project by Intel which tries to address the second issue, the SIMD Data Layout Template (SDLT). SDLT is a C++11 template library that provides abstract containers which aim to store data of Arrays of Structures (AoS), often used in an object-oriented software design, in the form of Structures of Arrays (SoA) or Arrays of Structures of Arrays (ASA) in order to improve SIMD efficiency. SoA's are more vectorisation friendly, because they allow to load contiguous data from memory to vector registers. ASA on the other hand, are SoA's (usually referred to as tiles), which are by themselves combined into an array, and fit into cache, such that data is *blocked* into cache for reuse maximisation.

SDLT is designed in a similar fashion as the Standard Template Library (STL), in that it provides containers and iterators (which are called accessors in SDLT) separately. The latter can then be used to access data in STL algorithms which is stored in these abstract containers. We give a minimal example on how to create a (1d SoA) SIMD friendly container from a `struct` in Lst. 2.2 and how to access it. The (not very elegant) use of the `SDLT_PRIMITIVE` macro is necessary because of the lack of compile-time reflection in C++11. That is, the standard does not define a way of type checking (convertibility, parent/child, base/derived, container/iterator, friends, ...), when meta-programming with classes that contain other classes etc. The two standard work-arounds for this lack is the use of code generators that parse C++ files in order to extract relationships between classes, and the use of macros. This issue, however, will likely be addressed in C++17 or C++20 [11].

The features SDLT aims to provide seem to be very import and one should hope that this issues will eventually be resolved in the C++ standard itself. There are already efforts to enable the use of (SIMD and thread) parallelism within the `<algorithm>` section of the STL. In particular, C++17 is likely to include the parallel and vectorised execution of algorithms à la `sort(par_vec, vec.begin(), vec.end())` [12].

SDLT has a few major disadvantages as a tool for the programmer to guarantee vectorisation. On a purely practical level, usage is restricted by the fact that SDLT is non-free (it ships with ICC starting from version 16 update 1) and close-source. More importantly however, it still relies on auto vectorisation done by the compiler.

The most high level option available at the moment to circumvent this issue is the use of *vector intrisics*. Intrinsics are available in C and C++ after including the header `immintrin.h`, and are

```c
#include "immintrin.h"

int main(void)
{
  float a[16] = {1.0};
  float b[16] = {2.0};
  float c[16] = {0.0};

  __mmask16 m16s = 0xAAAA;

  __m512 simd1 = _mm512_load_ps(a);
  __m512 simd2 = _mm512_load_ps(b);

  __m512 simd3 = _mm512_maskz_fmadd_ps(m16s, simd1, simd2, simd2);
  _mm512_store_ps(c, simd3);

  return 0;
}
```

Listing 2.3: Example using intrinsics to create a fused multiply-add with AVX-512 vectors of floats.

ordinary function calls with input parameters and return values. However, the compiler will not generate function calls in the binary when using intrinsics, but rather inlined SIMD instructions for the ISA used in the code. Using intrinsics as compared to assembly or inline-assembly has the advantage that the programmer does not have to take care about low level instruction scheduling and register allocation. Lst. 2.3 shows a simple example of a fused multiply and add of two arrays of 16 floats each, using a mask such that only every odd index is touched.

Today, all major compilers, including Clang, gcc and ICC, support intrinsics. Documentation for all Intel ISA's and the available intrinsics can be found online [6]. To see the code that is generated by the compiler, it suffices to compile with the option `-S`. This will create a file with suffix `.s`. For later convenience we give a list of the common suffixes of AVX-512 intrinsics in Tab. 2.1.

The QPhiX library guarantees the use of SIMD by using an explicit ASA layout for the data and by building kernels from vector intrinsics using a C++ code generator. We will study the details of this implementation in the next chapter.

Suffix	Type	Description
`_pd`	`__m512d`	8 packed doubles in 512-bit
`_ps`	`__m512s`	16 packed floats in 512-bit
`_sd`	`double`	double in lower 64-bit
`_ss`	`float`	float in lower 32-bit
`_epi32`	`__m512i`	extended packed interger of 32-bit (signed)
`_epi64`	`__m512i`	extended packed interger of 64-bit (signed)

Table 2.1: Most common data type suffixes for AVX-512 vector intrinsics.

2.2.3 Scaling

When it comes to parallel programming in a shared memory environment, thread based models are the predominant form of concurrency. There are several APIs based on thread scheduling including pthreads, C++11 threads and openMP. The latter is widely used in scientific computing and particularly in LQCD. It has a very simple to implement `#pragma` based interface, which however does not give the programmer a very fine-grained access to the scheduler. In particular, the creation and destruction of threads is entirely hidden at the software level. It also does not incorporate itself very well in an object-oriented design, because private/shared attributes do not bind to scopes. Shared variables, which are declared inside parallel regions, have to be marked `static`.

For C++ programmers there is however already a large set of concurrency features available through the standard itself. This includes threads, mutexes, atomicity and asynchronous tasks. C++17 will add upon this improvements in `std::future` and `std::experimental` with possibly executers and asynchronous operations, resumable functions (and lambdas), latches and barriers, atomic smart pointers [13], as well as the aforementioned parallel STL algorithms [12].

Apart from threads, there are a few APIs available that are build on tasks. It has been argued that concurrent programming has eventually to move from threads to tasks [45]. Amongst those APIs is the threading building blocks (TBB) library [7] and the hetero streams library [3]. The former is a free, open-source project, that has been initiated by Intel in 2006. It is a C++ template library that focusses on providing concurrent algorithm templates, as well as support facilities like a task-stealing scheduler, a concurrency-aware memory allocator, concurrent containers, portable mutexes and global timestamps. The hetero streams library on the other hand, is an open-source library that focusses on asynchronous task execution in heterogeneous platforms via streams. The Barcelona Supercomputing Center has an interesting collection of articles comparing hetero streams with CUDA streams and openCL [5].

2.3 Performance Model Revised

In the last chapter we calculated the algorithmic intensity for the *dslash* operator from the number of useful floating point operations, and established that it is generically bound by the memory bandwidth on Xeon Phi Processors. Here we want to present a simple model proposed in [44] to estimate the rooftop line of the performance we may expected on KNC and KNL, taking hardware features into account.

To this end, we have to replace the read and written bytes when calculating the arithmetic intensity by the actual memory bandwidth. In addition, we can model the hardware features of cache re-use, streaming stores and differences in read and write memory bandwidth. Streaming stores are instructions designed to continuously stream the output data to the main memory and to store it in a contiguous chunk. This has the advantage that data can be directly written to memory without having to have prior knowledge of the old content. This will save memory bandwidth needed to read the old data. Streaming store instructions are supported by a wide range of architectures including Xeon and Xeon Phi processors. Within the AVX-512 ISA in can be called via the intrinsic function `void _mm512_stream_pd(void* mem_addr, __m512d a)` passing the memory address and a reference to the data one wishes to store.

Reuse	Compression	Streaming Stores	Throughput [GB/s]		
			Half	Single	Double
0	no	no	635.9	318.0	159.0
0	no	yes	678.3	339.2	169.6
0	yes	no	726.8	363.4	181.7
0	yes	yes	782.7	391.4	195.7
7	no	no	1130.6	565.3	282.7
7	no	yes	1271.9	635.9	318.0
7	yes	no	1453.6	726.8	363.4
7	yes	yes	1695.8	848.0	423.9

Table 2.2: Estimated maximal throughput of the dslash *stencil for various tuning options and precision on KNL. We assume a memory bandwidth of 370 GB/s, which is about 85% of the STREAM bandwidth.*

Recalling that the *dslash* stencil needs 1320 floating point operations, writes one (result) spinor, and reads 8 link variable and 8 neighbouring spinors (plus possibly the resulting one) per site, we can estimate the throughput as follows:

$$F = 1320 \times B_r \times \left(\underbrace{8\,G + (8 - r_c)\,S + r_s\,S}_{\text{reads}} + \underbrace{\frac{B_r}{B_w}\,S}_{\text{writes}} \right)^{-1} .$$

Here G is the size of one link variable, which is 18 or 12 times the (precision-depending) size of one floating point number, and S is the size of a spinor, which again is 24 times the size of one float in Bytes. With the parameter r_c one can set the number of neighbouring spinor fields that may already be in cache and can be reused. It may take values between 0 and 7. This model assumes that the lowest lying cache is infinitely fast and B_r and B_w are the bandwidths between the main memory and this lowest cache (which we will assume to be MCDRAM and L1 for KNL respectively). The parameter r_s finally mimics streaming stores and is either $r_s = 0$ when they are used, or $r_s = 1$ when not. We do not include the possibility of link variable reuse, because one usually implements the even-odd preconditioned *dslash* operator. In this case, moving from an odd site to the next, non of the emanating links will be the same.

We summarise the rooftop performance for this model with the various options and precisions in Tab. 2.2. The calculations have been done for KNL, and we have set $B_r = B_w = 370\,\mathrm{GB/s}$, which is about 85% of the performance achieved in the STREAM benchmark. This is a reasonable estimate based on observations on Haswell and KNC architectures, cf. [55]. Note that the best throughput one could hope to achieve in single precision is about one third of the peak performance of the KNL, which reflects the fact that the *dslash* is memory bound, even with a large amount of cache reuse.

3. Software

In this chapter we will finally see the implementation of the *dslash* stencil and various iterative Krylov solvers, we have studied in the chapter about Algorithms, in the QPhiX library. In particular, we will remeet the three essentials of performant parallel software, which we have presented in the last chapter, in form of a cache blocked, OpenMP threading scheme, which utilises data locality and SIMDisation.

This chapter is organised as follows: First we will have a brief look what purpose QPhiX is meant to satisfy within the layered software design approach to LQCD simulations of the USQCD collaboration. Then we will have a closer lock at the structure and design of QPhiX itself, before we conclude the chapter with a detailed description of the changes and additions that were necessary to be made, in order to use QPhiX for twisted-mass fermions.

3.1 Software Layers for LQCD Simulations

QPhiX is part of a layered structure of programs which are supposed to provide a highly optimised and portable set of software to carry out simulations in LQCD on various platforms. There are four layers, cf. Fig. 3.1, which provide increasing functionality, but rely on the lower levels, respectively.

The first level provides basic functionality in form of the QCD Message Passing (QMP) parallel API which is built on top of MPI, and QLA, which makes a standard interface for linear algebra on sites or arrays of sites, available. The second level provides data parallel APIs in form of QDP/QDP++ for lattice-wide data structures, operations (in form of expression templates), as well as communication; and in form of QIO for I/O operations of lattice data. A similar software effort, including the use of SIMD vectors, was recently initiated by Peter Boyle [18]. The third level is meant to provide optimisations in form of efficient solvers and stencil operators. Examples of this layer are the QUDA library which is optimised for GPGPUs, and precisely the QPhiX library for Intel Xeon Phi's. In particular, QPhiX needs the QIO, QMP and QDP++ (for testing) libraries to be

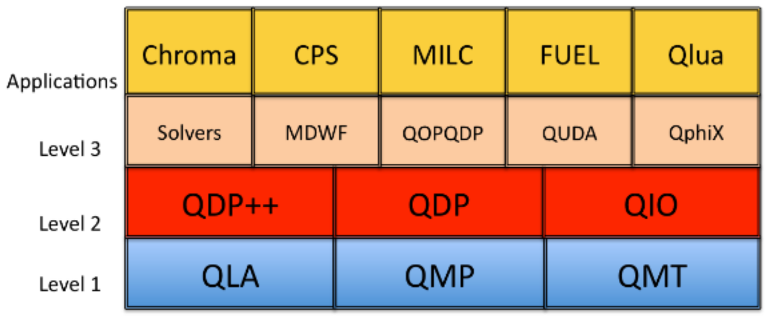

Figure 3.1: Software layers above and below QPhiX.

built.

Above this layer several software suites have been implemented which provide functionality needed for the investigation of spectroscopy, decay constants, nucleon form factors and chiral aspects of LQCD. Examples of the application layer are Chroma [29] and MILC [1].

3.2 QPhiX

QPhiX is a C++11 template library with external code generator QPhiX-codegen. It implements in essence three types of facilities. First and foremost, it provides two (checkerboarded) matrix multiplication routines, namely

$$(1)\ y = \not{D}\,x, \qquad (\,y = A^{-1}\,\not{D}\,x\,), \tag{3.1}$$

$$(2)\ z = a\,x - b\,\not{D}\,y, \quad (\,z = A\,x - b\,\not{D}\,y\,), \tag{3.2}$$

where a and b are real coefficients, $\not{D}$ is the *dslash* stencil and A is the clover term. The routines in parenthesis are used when the Wilson clover term is included. The first one is referred to as `dslash` and the second one as `achimbdpsi` in the code. From these routines, as we have seen in Eqn. (1.60) and Eqn. (1.65), one can easily build an even-odd preconditioned fermion matrix multiplication routine.

Secondly, it provides BLAS facilities, *e.g.* $y \hookleftarrow \alpha y + x$, $\rho \hookleftarrow ||r||^2$, $\rho \hookleftarrow ||x - y||^2$, etc. They are implemented by combining loops over lattice sites with the Structure of Arrays (SoA's) of the data types, and providing real and complex functors which can act on the vectors.

Thirdly, there are three iterative solvers that use the above functionality to invert the even-odd preconditioned fermion matrix (that is, its odd-odd or even-even part). Reconstruction routines to obtain the solution to the full system are not included in QPhiX itself. There is a CG solver that solves the normal system $M^\dagger M x = b$, as well as a BiCGStab and a modified Richardson iteration solver which directly solve the system $Mx = b$. The latter implements mixed precision preconditioning using iterative refinement, as we have seen it in Sec. 1.2.4.

QPhiX is heavily templated over four parameters:

1. `typename FT`
2. `int VECLEN`
3. `int SOALEN`
4. `bool COMPRESS12`

The first parameter is the floating point type and can be chosen to be half, single or double precision. The second template parameter specifies the length of the SIMD vectors for the architecture one likes to build upon, in units of floating point numbers in the precision one uses. For Xeon Phi's it will assume values of 8 for double precision and 16 for single and half precision. The third parameter specifies the length of the SoA's inside the SIMD vector, which has to be a factor of `VECLEN` and could be `SOALEN` $= 4, 8, 16$ on Xeon Phi's. The last parameter finally specifies if 12 parameter gauge compression should be used or not, cf. Sec. 1.1.1.

Since all these template parameters relate to types and the dimensions of arrays, their values have to be set at compile time. QPhiX is essentially a header-only library, as it is common for C++ software that heavily relies on templates (cf. STL, boost, etc.).[1] To be configured and build, QPhiX uses GNU Autoconf and Automake, which allows for a high level of portability and abstracts the handle of library dependencies (mainly QDP++, QMP and libxml2) into the autotools functionality.

For the convenience of the reader we summarise (most of) the file structure of QPhiX in Fig. 3.2. There are three main directories: The `lib/` directory where the main static library `libqphix_solver.a` is build, and one finds thread binding for BlueGene/Q and a generic architecture, as well as printing utilities with multi-node support. Then there is the directory `tests/` where testing and timing facilities are implemented. Finally, the main part of the source code lies under `include/qphix/`.

The main purpose of the QPhiX library is to provide the lattice site loops for the *dslash* stencil, the inter-node communication (combined with a heuristic load balanced scheduler), as well as the OpenMP threading. The implementation of the two above mentioned kernels for the bulk or body of the lattice (as well as for the face or boundary of the node-local lattice) however, will be generated in the QPhiX code generator, which we describe in more details later. Here, it is important to know that this pre-generated kernel routines, which operate on SIMD vectors, will be provided in subfolders `ARCH/generated/` where `ARCH` may be SCALAR, SSE, AVX, AVX2, AVX512 (KNL) or MIC (KNC). The appropriate routines for the various precisions and values for `VECLEN` and `SOALEN`, will be `#include`'d using a set of `#ifdef`-macros in the template specialisation files, which are found under `ARCH/`.

3.2.1 Data Structures, Tiles & Geometry

The geometry of the lattice, as well as its subdivision into tiles and blocks is defined in the file `geometry.h`. It contains the definition for the half precision type (cf. Lst. 3.1), and the functionality to (up and down) convert between half, single and double precision. It also also provides a constructor to initialise the geometry parameters such as the number of cores `NCores` used per node, the dimensions of the blocks in Y and Z direction `By` and `Bz` (used for cache blocking, see below), the number of hyper-threads (SMT threads) within one core `Sy` and `Sz`, as well as the padding for XY-planes and (XYZ) time-slices `PadXY` and `PadXYZ`. It also provides (aligned) allocation and free routines for all the introduced data types (with support for 2 MB pages).

The `class Geometry` also contains the basic type definitions for the spinors, gauges and clover terms on checkerboarded sublattices for which `Nxh` $=$ `Nx`$/2$, `Nx` being the number of sites in the X-direction. As we have mentioned in the last chapter, in order to be able to sufficiently use SIMD vector instructions, data has to be split into SIMD vectors which lie contiguously and aligned

[1] This approach is known as the *inclusion model*, and is one way to make sure the compiler understands for what template parameters a given definition should be instantiated. More details and alternative approaches can be found in [78].

```
qphix
├── include/
│   └── qphix/
│       ├── ARCH/
│       │   ├── generated/
│       │   │   └── pre-generated KERNELs
│       │   ├── KERNEL_ARCH_complete_specialization_form.h
│       │   └── KERNEL_ARCH_complete_specialization.h
│       ├── abs_solver.h
│       ├── Barrier_mic.h
│       ├── Barrier_stubs.h
│       ├── blas_c.h
│       ├── blas.h
│       ├── blas_mic.h
│       ├── blas_new_c.h
│       ├── blas_utils.h
│       ├── comm.h
│       ├── complex_functors.h
│       ├── dslash_body.h
│       ├── dslash_def.h
│       ├── dslash_generated.h
│       ├── dslash_utils.h
│       ├── face.h
│       ├── geometry.h
│       ├── invbicgstab.h
│       ├── invcg.h
│       ├── inv_richardson_multiprec.h
│       ├── linearOp.h
│       ├── minvcg.h
│       ├── print_utils.h
│       ├── qdp_packer.h
│       ├── real_functors.h
│       ├── site_loops.h
│       ├── threadbind.h
│       └── wilson.h
├── lib/
│   ├── bgq_threadbind.cc
│   ├── generic_threadbind.cc
│   └── print_utils.cc
└── tests/
    └── ...
```

Figure 3.2: The most important files of the QPhiX library. Autotool files, tests and clover specific files are left out to safe space.

```
typedef unsigned short half;

template<typename FT, int VECLEN, int SOALEN, bool COMPRESS12>
class Geometry {
  public:
    typedef T FourSpinorBlock[3][4][2][SOALEN];
    typedef T SU3MatrixBlock[8][COMPRESS12?2:3][3][2][VECLEN];

    ...
};
```

Listing 3.1: The elementary data types for spinors, gauges, and half precision. [`geometry.h`]

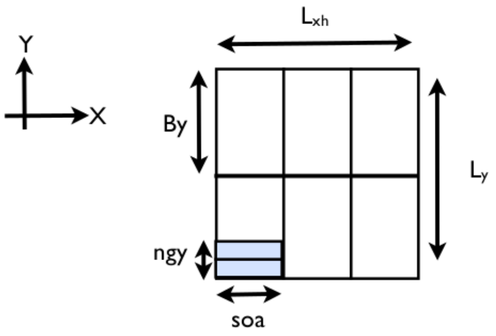

Figure 3.3: Divison of a checkerboarded XY-plane into tiles (of `ngy` $*$ `SOALEN` elements) and blocks for cache optimisation [43].

in memory. One possibility to guarantee that, is to form vectors from lattice sites in the X-direction. This has been done in [68], but it is rather restrictive, as one has to guarantee that `Nxh` is a multiple of the register length `VECLEN`. In QPhiX, SIMD vectors are formed from *tiles* in XY-planes instead. Then, only `SOALEN` sites belong to a tile in X-direction, which have `ngy` $=$ `VECLEN`/`SOALEN` different Y-values[2]. For this approach to work on has to guarantee that `ngy` divides `Ny`, and `SOALEN` divides `Nxh`. Then in each step, the (pre-generated) kernels will still process `VECLEN` number of sites, but this time from `ngy` different Y-values. The division of one XY-plane of the (checkerboarded) lattice into tiles is illustrated in Fig. 3.3.

The resulting (elementary) data structures for spinors and gauges are summarised in Lst. 3.1. A `FourSpinorBlock` carries indices (in order of increasing contiguity) for colour, spin, real/complex and SIMD of the tile. To implement a full spinor field on the entire lattice, one thus needs an array of length `Nxh` $*$ `Ny` $*$ `Nz` $*$ `Nt` / `SOALEN`. This is the Array of Structures of Arrays (ASA) we have mentioned in the last chapter.

The link variables are implemented in a slightly different way, in that they carry an index for the eight directions (backward/forward in four dimensions), in which the links emanate from each site, an index for the colour row of the $SU(3)$ matrix, the colour column, real/imaginary part and again the position inside the SIMD vector. But since gauges are reused in successive *dslash* applications, it is beneficial to repack them into full registers, so that they can be read as a single vector. Thus, the gauge field array will be of length `Nxh` $*$ `Ny` $*$ `Nz` $*$ `Nt` / `VECLEN`. Note, that

[2]On the Xeon Phi's this will be either 1, 2 or 4.

```
FourSpinorBlock[xb+Lxh/SOALEN*y+xyBase][colour][spin][RE/IM][xi];
SU3MatrixBlock[xb+(Lxh/SOALEN*y+xyBase)/ngy][direction][row][col][RE/IM][xi];
```

Listing 3.2: Accessing elements inside arrays of spinors and gauges, given a lattice site (x, y, z, t). Cf. the text for further explanation.

when gauge compression is used, each `SU3MatrixBlock` has only two rows instead of three.

To reduce the number of associativity conflict misses caused when reading vectors with less than `VECLEN` elements, one can use padding for every XY-plane, as well as for every time-slice (XYZ-hyper-plane). Then, given a lattice index (x, y, z, t), one finds the associated spinor at this site in three steps. First, one has to calculate the XY-plane and its respective `xyBase`, as follows:

$$\texttt{xyBase} = \texttt{t} * \texttt{Pxyz} + \texttt{z} * \texttt{Pxy}\,,$$

$$\text{where}\ \ \texttt{Pxy} = (\texttt{Nxh} * \texttt{Ny}/\texttt{SOALEN}) + \texttt{PadXY}\,,$$

$$\text{and}\ \ \texttt{Pxyz} = \texttt{Nz} * \texttt{Pxy} + \texttt{PadXYZ}\,.$$

Then, secondly, one has to calculate the tile `xb` and its associated vector index `xi`:

$$\texttt{xb} = \texttt{x}\,/\,\texttt{SOALEN}\,,$$

$$\texttt{xi} = \texttt{x}\,\%\,\texttt{SOALEN}\,.$$

Finally, one finds the offset in the T-direction by (cf. Fig. 3.3 again):

$$\texttt{yOffset} = \texttt{xb} + \texttt{Nxh}\,/\,\texttt{SOALEN} * \texttt{y}\,.$$

In case of the gauge fields, one has to take into account that `ngy` values in the Y-direction are repacked together. In this case, the array index becomes (cf. Lst 3.2):

$$\texttt{xb} + (\texttt{Nxh}\,/\,\texttt{SOALEN} * \texttt{y} + \texttt{xyBase})\,/\,\texttt{ngy}\,.$$

3.2.2 Cache Blocking and Load Balancing

Now we have padded and aligned data structures composed of SIMD vectors, which can be loaded contiguously into vector registers. However, the lattice still needs to be subdivided into chucks, such that these chucks can be assigned to different threads, when looping over the lattice sites.

When dividing the lattice in such chucks or blocks, there are two things that have to be kept in mind. First, the blocks should ideally fit into low-lying caches to reduce memory bandwidth/latency bottlenecks. Secondly, when processing the blocks, the utilisation of available cores/threads should be maximised. The latter will depend both on the blocking scheme itself, as well as on the block-to-core mapping.

QPhiX implements a method known as 3.5D-blocking [52]. This works as follows: The Y- and Z-direction of the lattice are subdivided into blocks of length `By` and `Bz`, respectively. The

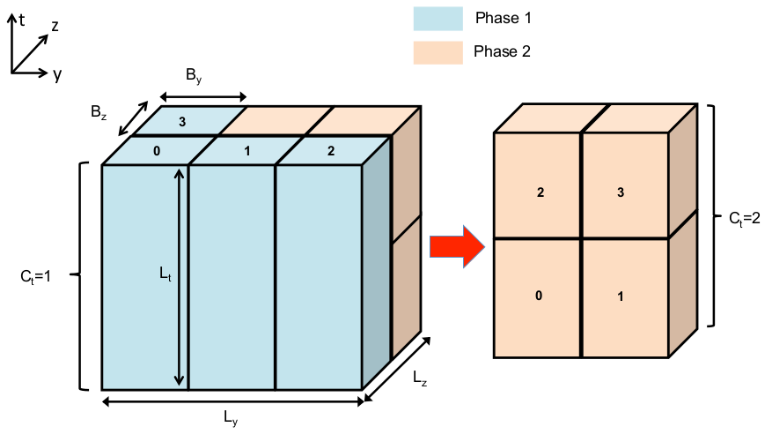

Figure 3.4: The lattice is sub-divided into blocks, which fit into L2 cache, in the Y- and Z-direction. These blocks are mapped onto the cores in multiple phase, using a heuristic load balancing scheme [43].

ideal values of these variables will depend on the architecture and can be set by the user, when constructing an instance of `class Geometry`. On dual-socket Xeon processors they are best set to 8, while for Xeon Phi's the best value for both parameters turns out to be 4. In this way, at least three (blocked) time-slices of dimension $\texttt{Nxh} * \texttt{By} * \texttt{Bz}$ will fit into the L2 cache.

Now, one has to map these $N_b = \texttt{Ny} \,/\, \texttt{By} * \texttt{Nz} \,/\, \texttt{Bz}$ blocks to cores, possibly in multiple phases. Let us denote the number of remaining blocks to be processed with N_r, where initially $N_r = N_b$. As long as $N_r \geq \texttt{Ncores}$, each cores will be assigned with one block, and the block is streamed up in the positive T-direction. As soon as $N_r < \texttt{Ncores}$, there will not be work left for all cores. To prevent cores from running idle, the blocks will be divided in the T-direction (this is the half in 3.5D, since in this case one also blocks in the time direction).

Assuming the number of remaining blocks divides the number of cores available, we can just split each block into $C_t = \texttt{Ncores}/N_r$ sub-blocks and finish the processing. However, if this is not the case, there are two possibilities to choose from. We can either divide each block into $C_t^l = \lfloor \texttt{Ncores}/N_r \rfloor$ or into $C_t^u = \lceil \texttt{Ncores}/N_r \rceil$ sub-blocks. In the former case, we will have fewer blocks than cores, so the processing will finish, but not all cores will be used in this phase. In the latter case, the core utilisation will be higher, but in the subsequent phase there will be even fewer blocks, which will have to be split in even smaller chunks.

This is where the load balancing comes into play. It is based on the heuristic assumption that in case we would split each T-slice in more than $C_t^u > T_t$ chunks, we can assume that too little work would be done, and so we will choose C_t^l instead, and finish the processing. Here, T_t is a threshold that will depend on `Nt` and the architecture, but for reasonable lattice sizes $T_t = 4$ turns out to be optimal for both Xeon and Xeon Phi processors.

Apart from an upper bound, the user may also specify a lower bound `minCt` of the number of blocks in the T-direction. This is beneficial when working with NUMA architectures (like dual-socket Xeon's), because then one can map sub-blocks to cores which are "closest". In this way, one may set this parameter to 1 for Xeon Phi's, but to 2 for Xeon's. Since the entire phase break down only depends on the geometry of the lattice and the node, it will be pre-constructed when initialising a `Geometry` object.

Fig. 3.2.2 illustrates the cache blocking scheme (with suppressed X-direction) with a two-phase processing. Here, 4 cores work on 6 blocks by first processing one entire (blue) block each,

```
// Get Core ID and SMT ID
int cid   = tid / n_threads_per_core;
int smtid = tid - n_threads_per_core * cid;

// Compute SMT ID for Y- and Z-direction
int smtid_z = smtid / Sy;
int smtid_y = smtid - Sy * smtid_z;
```

Listing 3.3: How to calculate core and SMT IDs, given the OpenMP thread ID. [`dslash_body.h`]

streaming over the T-direction, in Phase 1. In the second phase the two remaining blocks are sub-divided into four equally sized (orange) sub-blocks along the T-direction. They can then be processed *round-robin* using all four cores. Note, that although the initial number of blocks is not divisible by the number of available cores, non of the cores runs idle during the entire processing.

3.2.3 SMT Threading and Lattice Traversal

We have seen in the last chapter, that each core inside a Xeon Phi processor has up to 4 hardware threads called hyper-threads or SMT threads. QPhiX treats these threads inside a core as a grid of dimension $\mathtt{Sy} * \mathtt{Sz}$, both of which variables can be set by the user. The number of OpenMP threads will then be equal to $\mathtt{Ncores} * \mathtt{Sy} * \mathtt{Sz}$. The thread, core and SMT IDs satisfy the relation $\mathtt{tid} = \mathtt{Sy} * \mathtt{Sz} * \mathtt{cid} + \mathtt{Sy} * \mathtt{smtid\_z} + \mathtt{smtid\_y}$ and can be calculated as shown in Lst. 3.3.

With the SMT threading, the data layout using tiles and the blocking scheme in place, we are now ready to understand the lattice traversal for the *dslash* stencil operator. It is implemented in the two routines

```
Dslash<FT,VECLEN,SOALEN,COMPRESS12>::DyzPlus

Dslash<FT,VECLEN,SOALEN,COMPRESS12>::DyzMinus
```

for *dslash* and its hermitian conjugate, respectively, in the file `dslash_body.h` (and similarly for the *dslash* stencil with clover term in `clover_dslash_body.h`). As its parameters, it takes pointers to the in- and output spinor fields, the gauge field, as well as the thread ID and the checkerboard (even or odd), cf. Lst. 3.4.

All threads loop over all the phases constructed from the geometry. If there are fewer blocks than cores, cores which are not needed, will skip to the next phase. From the thread ID, each thread will calculate its core ID, the SMT IDs, the block, it has to process, as well as the starting value in the time direction. Every thread then streams over its range of T values in its block. For each given time slice, it traverses the XYZ volume in the following way: Every core splits its respective XY-planes between the SMT threads, such that every SMT thread processes a tile of size $\mathtt{SOALEN} * \mathtt{ngy}$, at any given step. The volume is then traversed by processing chunks of size `SOALEN` in the X-direction, and increasing Y in steps of `ngy` (the Y-axis of the SMT grid). The next plane (associated with the Z-axis of the SMT grid) will then be reached, by looping over the Z-direction in strides of `Sz`.

Within the loops, the indices can be used to calculated the addresses of neighbouring spinor blocks, the output spinor and the gauge field, as well as the offsets to the neighbouring spinors for successive blocks. They will be used to prefetch data to the L2 cache, in case that software prefetching is turned on. In addition, communication for the faces is handled. Finally, all this

```
template<typename FT, int VECLEN, int SOALEN, bool COMPRESS12>
void Dslash<FT,VECLEN,SOALEN,COMPRESS12>::DyzPlus(
    int tid, const FourSpinorBlock* psi,
    FourSpinorBlock* result, const SU3MatrixBlock* u, int cb)
{
  // Loop over phases
  for(int ph = 0; ph < num_phases; ph++) {
    int nActiveCores = phase.Cyz * phase.Ct;
    if ( cid >= nActiveCores ) continue;

    // Stream over blocks of time slices
    for(int ct = 0; ct < Nct; ct++) {

      // Handle B/Cs here

      // Loop over z
      for(int cz = smtid_z; cz < Bz; cz += Sz) {

        // calculate spinor base address for:
        // x & y neighbours
        // backward/forward z neighbour
        // backward/forward t neighbour
        // output

        // Loop over y
        for(int cy = nyg * smtid_y; cy < By; cy += nyg*Sy) {

          // cx loops over the soalen partial vectors
          for(int cx = 0; cx < nvecs; cx++) {

            // calculate base address for gauges
            // and various offset to successive fields
            // for L2 prefetches

            // Call the pre-generate kernel on the tile
            // with the appropriate pointers
            dslash_plus_vec<FT,VECLEN,SOALEN,COMPRESS12>(...);

          }
        } // End for over scanlines y
      } // End for over scalines z

      // Call a barrier within a core group
      if( ct % BARRIER_TSLICES == 0 )
        barriers[ph][binfo.cid_t]->wait(binfo.group_tid);

    } // end for over t
  } // phases
}
```

Listing 3.4: The skeleton of the lattice traversal for the dslash stencil. Here, indices of neighbouring spinor and gauges, output addresses, and offsets for L2 prefetches are computed and the pre-generated service kernel, which operates on tiles is called. There, the main dslash computation is carried out. (We have suppressed all QMP communication done here.) [`dslash_body.h`]

```
template<typename FT, int V, int S, bool C>
void axpy(const double alpha, const FourSpinorBlock* x,
    FourSpinorBlock* y, const Geometry<FT,V,S,C>& geom,
    int n_blas_simt)
{
  AXPYFunctor<FT,V,S,C> f(alpha, x, y);
  siteLoopNoReduction<FT,V,S,C, AXPYFunctor<FT,V,S,C> >
    (f, geom, n_blas_simt);
}
```

Listing 3.5: Structure of BLAS routines, for the example of an axpy. Every operation has its own functor, which is constructed and fed to the site loop facility template functions, which also handle reductions. [`blas_new_c.h`]

data is passed to the service kernel, in which the actual calculation will take place. The necessary kernels are pre-generated with QPhiX-codegen, which we will cover shortly.

3.2.4 BLAS Linear Algebra

As we have seen earlier, in order to implement an iterative solver, one not only needs a matrix-vector multiplication, but also a number of vector linear algebra routines, such as scalar products and vector norm squared calculations. All of these operations have a very low arithmetic intensity, and are thus memory bound. This is why it turns out to be beneficial to fuse several kernels into one, such as the calculation of a vector difference and the norm squared of the resulting residual vector.

QPhiX implements all BLAS linear algebra routines by constructing a functor for every operation that acts on a single array element (which by itself consists of $(3*4*2*\texttt{SOALEN})\,/\,\texttt{VECLEN}$ SIMD vectors), and providing separate side loop facilities with and without reduction of a variable. This is illustrated in Lst. 3.5.

Every functor is a template class which provides an inline member function

```
inline void func(int block, double* reduction),
```

which loops over a single element of a spinor array, given by `block`. The second argument only appears when a reduction of a variable is necessary (as for instance when calculating a norm). The loops over the `blocks` are *auto-vectorised*, which is guaranteed with `#pragma SIMD`, and compiler hints about their memory alignment. The number of SMT threads used inside every single BLAS routine can be set when calling the routine, and is auto-tuned inside the iterative solvers, see below. It shows, that best throughput is usually achieved with only one or two hyper-threads, whereas the *dslash* stencil benefits from up to four.

Finally, the lattice traversal and possibly a reduction is managed inside the loop template functions in the file `site_loops.h`. Here, each thread calculates the range of array elements it has to process, much in the same way, as we have outline above for the *dslash* stencil. Then each thread loops over its local volume and calls the member function `func` of the passed functor on each array element. In the case of reductions, the site loop will also handle the message passing communication.

3.2.5 Even-Odd Operators & Iterative Solvers

Now that we have seen all the low-level functionality, which makes up the major part of QPhiX, let us turn to the high-level routines and interfaces, a user is likely to access.

```
template<typename FT, int VECLEN, int SOALEN, bool COMPRESS12>
class EvenOddLinearOperator {
  public:
    virtual void operator() (FourSpinorBlock *result,
        const FourSpinorBlock* in_spinor, int isign) = 0;
    virtual Geometry<FT,VECLEN,SOALEN,COMPRESS12>&
      getGeometry(void) = 0;
};
```

Listing 3.6: An abstract even-odd operator should be applicable to a spinor field and know about its geometry. Here and in what follows, `isign==1` *stands for the normal operator, and* `isign==-1` *for the application of its hermitian conjugate. [*`linearOp.h`*]*

```
void EvenOddWilsonOperator::operator()
  (FourSpinorBlock *result, const FourSpinorBlock* in, int isign)
{
  D->dslash(tmp, in, u[1], isign, 1);
  D->dslashAChiMinusBDPsi(result, tmp, in, u[0],
       mass_factor_alpha, mass_factor_beta, isign, 0);
}
```

Listing 3.7: Implementation of the Wilson even-odd preconditioned fermion matrix. `D` *is an instance of the* `Dslash` *class,* `u` *are gauge field pointers, and the last argument in the function calls is the checkerboard. [*`wilson.h`*]*

As we have explained in the chapter about Algorithms, inverting the fermion matrix in LQCD is best approached using an iterative solver for the even-odd preconditioned linear operator $\tilde{M}_{oo}$, possibly in mixed precision. This operator in turn, can be constructed using the two low-level routines mentioned in the beginning of this section.

However, to use any iterative Krylov subspace solver, one really only needs to be able to apply a matrix to a vector. This is why QPhiX implements an abstract `EvenOddLinearOperator` which has to overload `operator()`, that is, must be applicable to a spinor field, and has a *getter* for its `Geometry`, cf. Lst. 3.6. This approach turns out to be highly beneficial, when it comes to extending QPhiX for other lattice actions, because it allows to use all the solver facilities, as long as a new `EvenOddLinearOperator` is provided. QPhiX itself implements two derived classes for the even-odd preconditioned fermion matrix with and without clover term, in the form of `EvenOddWilsonOperator` and `EvenOddCloverOperator`. Lst. 3.7 shows the overloaded parenthesis operator for the case without clover term. We easily recognise the even-odd preconditioned fermion matrix built from the two basic kernels, as described in Sec. 1.2.4, Eqn. (1.60).

The solvers themselves are also abstracted, and have to provide similar functionality as the linear operators, cf. Lst. 3.8. There are template classes for a solver that solves a single linear system, as well as a multi-solver, that solves a set of linear equations, in which the operators are related by a shift.[3] Note, that a child class of `AbstractSolver` does not rely on an `Even-OddLinearOperator`. In this way, a solver may be implemented using an operator without preconditioning. The CG, BiCGStab and Richardson solvers, however, use such an instance. This can be seen in the short example of the constructor for the conjugate gradient solver `InvCG` in Lst.

[3]This is useful *e.g.* to compute propagators for different quark masses simultaneously.

```
class AbstractSolver {
  public:
    virtual void operator() (
        Spinor* x,
        const Spinor *rhs,
        const double RsdTarget,
        int& niters,
        double& rsd_sq_final,
        unsigned long& site_flops,
        unsigned long& mv_apps,
        int isign,
        bool verboseP) = 0;
    virtual void tune(void) = 0;
    virtual Geometry<FT,VECLEN,SOALEN,COMPRESS12>& getGeometry() = 0;
};
```

Listing 3.8: An iterative solver inheriting from `AbstractSolver` has to provide a right-hand side and a target residual. [`abs_solver.h`]

```
class InvCG : public AbstractSolver<FT,VECLEN,SOALEN,COMPRESS12> {
  public:
    InvCG(EvenOddLinearOperator<FT,VECLEN,SOALEN,COMPRESS12>& M_, int MaxIters_)
      : M(M_), geom(M_.getGeometry()), MaxIters(MaxIters_)
  { ... }
}
```

Listing 3.9: All solvers implemented in QPhiX inherit from `AbstractSolver` and are constructed from an instance of `EvenOddLinearOperator`. [`invcg.h`]

3.9, which only take a reference to an `EvenOddLinearOperator`-object and an upper bound on the number of iterations. `InvCG` as well as the other solvers, also provide functionality to get and set the number of OpenMP threads used in the various BLAS routines within the solver, and a tuning procedure, which optimises these numbers for every single routine.

3.2.6 Inter-Node Communication

For all its inter-node communication QPhiX uses the QCD Message Passing API, which is built on top of MPI [42]. Since the SIMD vectors are laid out in the X- and Y-direction, MPI communication is done only in the Z- and T-direction.

To this end, the lattice is divided into a bulk (called the body) and the boundary (called face). The processing of the faces is split into two steps. In the first step, the spinors are projected onto half-spinors, as we have described in Sec. 1.1.1, and subsequently sent to its destination node. This step is done in overlap with the calculation of the *dslash* body. As soon as the faces are received at their destination and the body calculation is finished, they will be processed further in a second step. Then, they will be multiplied with the appropriate link variables, the lower half spinors reconstructed, and eventually accumulated to the final *dslash* result. The face data transferred for a typical multi-node simulation in LQCD is of the order of 256 kB up to few MBs per *dslash* application.

The machinery needed for both steps is implemented in `face.h` in the two functions

```
void Dslash<FT,VECLEN,SOALEN,COMPRESS12>::packFaceDir,

void Dslash<FT,VECLEN,SOALEN,COMPRESS12>::completeFaceDir.
```

These routines have a structure similar to `DyzPlus` and `DyzMinus`, and are used to process the body, as we have seen earlier. Their primary purpose is to collect the needed bases addresses and offsets for the spinors and gauges involved, and then call the service kernels, pre-generated in QPhiX-codegen. These kernels are called `face_proj_dir_plus/minus` and `face_finish_dir_plus/minus`, and we will meet them again shortly. Finally, the management of the QMP communication, in accord with body and face processing, is done in

```
void Dslash<FT,VECLEN,SOALEN,COMPRESS12>::DPsiPlus,

void Dslash<FT,VECLEN,SOALEN,COMPRESS12>::DPsiMinus.
```

which are implemented in `dslash_body`.

3.2.7 Barriers

To conclude this section, let us mention a peculiar feature, which is used to synchronise cores within the *dslash* stencil. It turns out, that using lightweight, local barriers to occasionally synchronise threads, is beneficial on Xeon Phi's.

These barriers were provided by the Intel Cooperation and can be found in the file `Barrier_mic.h`. They function as follows: During each phase, cores are structured together into groups, such that they work on the same range of T-slices. They will then be synchronised by calling a barrier at the end of every so-many T-slices (cf. the last function call in Lst. 3.4). This parameter can be tuned individually, and seems to give best results when it is set to 16 slices.

The reason why this is beneficial is the following: Since cores of one group are supposed to work on the same T-slice at the same time, one core might find a missed cache line in the L2 cache of the other (recall the tag directory based L2 cache coherency from last chapter). In case the processing between two cores of one group becomes too desynchronised, data may have to be fetched from main memory upon an L2 miss. This outweighs the costs of occasional barriers, which reduce the probability of that to happen.

3.3 QPhiX Code Generator

Now that we have seen, how QPhiX implements the data layout, the lattice traversal and the high-level functionality, let us turn to the low-level functionality that carries out the actual computation.

These functions are generated with a C++11 code generator called QPhiX-codegen. They contain vector intrinsics and act on the XY-tiles we have seen in the last section. The main purpose of the code generator was to provide a unified interface which gives access to hardware features like streaming stores, gather/scatters, load-packs etc. It also allows for an elegant inclusion of software prefetches, which were crucial to achieve performance on KNC's—all using a high-level software design which is maintainable, readable and extendable. Indeed, it is fairly simple to expand QPhiX-codegen both for different ISA's, as well as for new kernels, which are needed when using different lattice actions and the like.

```
qphix-codegen
├── ARCH/
│   └── generated KERNELs
├── address_types.h
├── codegen.cc
├── data_types.cc
├── data_types.h
├── dslash.cc
├── dslash.h
├── dslash_common.cc
├── instructions.h
├── inst_scalar.cc
├── inst_sp_vec16.cc
├── inst_sp_vec4.cc
├── inst_sp_vec8.cc
├── inst_dp_vec2.cc
├── inst_dp_vec4.cc
├── inst_dp_vec8.cc
├── Makefile
└── customMake.ARCH
```

Figure 3.5: The QPhiX code generator file structure.

Code generators have already been used in the past for LQCD, for instance in the assembly generator BAGEL [16] and the QA0 code generator [54].

3.3.1 Instructions, Addresses & Vector Registers

A list of files included in the code generator can be found in Fig. 3.5. The generator defines and uses objects of three main types: the actual C++ instructions used in the code, which will include vector intrinsics function calls, then the memory addresses for the various data types, offset, etc., and finally the vector registers, which will be variables of SIMD types in the generated code. All of these objects are instances of classes that inherit from `Instruction`, `Address` or `FVec`, respectively. These classes are implemented in `instructions.h` and `address_types.h`.

Instructions come in two types: firstly general ones, like multiplications, additions, and fused multiply adds, but also auxiliary ones like scope delimiters, if-else conditional blocks, and variable declarations; and secondly memory reference instructions. Both `Instruction`'s as well as `Address`'es have a member function `serialise()` which returns a `std::string` representing the actual C++ code. Registers on the other hand are managed with the help of `FVec` objects, which carry a name that identifies the associated SIMD register in the code. Instructions reference `Address` and `FVec` objects, and are stored in `std::vector`'s (`InstVector`) during the code generation. Each kernel is produced from one such `InstVector`, which will be serialised and written to a file at the end of the code generation.

The additional attribute of the `Address` and `Instruction class` allows to extract information for the code analysis later. In this way, one can for instance count the number of arithmetic vs. memory reference instructions, or extract addresses from `MemRefInstruction`'s in order

```cpp
class Instruction
{
  public:
    // string class return empty string
    virtual std::string serialize() const
    {
      return std::string("");
    }
    virtual bool hasAddress() const
    {
      return false;
    }
    virtual int numArithmeticInst() const
    {
      return 0;
    }
    virtual int numDeclarations() const
    {
      return 0;
    }
    virtual int numScopes() const
    {
      return 0;
    }
    virtual int numIfs() const
    {
      return 0;
    }
};

typedef vector<Instruction *> InstVector;
```

Listing 3.10: The Instruction class provides a serialize function to generate the ISA specific vector intrinsics, and several counters to generate statistics of the final code. [`instructions.h`]

```cpp
enum AddressType { GAUGE, SPINOR, CLOVER_DIAG, CLOVER_OFFDIAG, ADDRESS_OF_SCALAR
    , GENERIC_ADDRESS};

class Address
{
  public:
    Address(int isHalfType_) : halfType(isHalfType_) {}
    virtual std::string serialize(void) const = 0;
    virtual AddressType getType(void) const = 0;
    int isHalfType(void) { return halfType; }
  private:
    int halfType;
};
```

Listing 3.11: The address class also provides a serialize function for physical addresses and an interface for distinguished half precision treatment. [`address_types.h`]

```cpp
class FVec
{
  public:
    FVec(const std::string& name_);
    FVec(const FVec& v_) : name(v_.getName()), type(v_.getType()) {}
    const std::string& getName() const { return name; }
    const std::string& getType() const { return type; }
  private:
    const std::string name;
    const std::string type;
};
```

Listing 3.12: FVec objects are used to assign names to vector registers and treat them has variables in the C++ code. [`instructions.h`]

```cpp
enum MemRefType { LOAD_ALIGNED_VEC, LOAD_UNALIGNED_VEC, LOAD_MASKED_VEC,
    STORE_VEC, STREAM_VEC, STORE_MASKED_VEC, LOAD_NONVEC, L1_PREFETCH,
    NTA_PREFETCH, L2_PREFETCH, L1_EVICT, L2_EVICT, GATHER_VEC, SCATTER_VEC,
    GATHER_PREFETCH_L1, GATHER_PREFETCH_L2, GATHER_PREFETCH_NTA };

class MemRefInstruction : public Instruction
{
  public:
    // Override virtual
    virtual bool hasAddress() const
    {
      return true;
    }
    virtual const Address* getAddress() const = 0;
    virtual MemRefType getType() const = 0;
    virtual bool hasGSAddress() const
    {
      return false;
    }
};
```

Listing 3.13: Memory instructions are `Instruction`'s with additional facilities to handle their memory addresses. [`instructions.h`]

```
class FMAdd : public Instruction
{
  public:
    // ret = a * b + c
    FMAdd( const FVec& ret_, const FVec& a_, const FVec& b_,
        const FVec& c_, const std::string& mask_) :
      ret(ret_), a(a_), b(b_), c(c_), mask(mask_) {}
    std::string serialize() const;
    int numArithmeticInst() const { return 1; }
  private:
    const FVec ret;
    const FVec a;
    const FVec b;
    const FVec c;
    const std::string mask;
};

inline void fmaddFVec(InstVector& ivector, const FVec& ret, const FVec& a, const
    FVec& b, const FVec& c, std::string mask = "")
{
  ivector.push_back(new FMAdd(ret, a, b, c, mask));
}
```

Listing 3.14: Utility function for a fused multiply add instruction. [`instructions.h`]

to automatically generate prefetch instructions.

In addition, there are several utility functions, as the fused multiply add shown in Lst. 3.14, which take an `InstVector` and several `FVec`'s in order to generate an instruction that utilises these specific vector registers. Finally, the instruction is appended to the original vector and returned. The actual kernels are almost exclusively built from these utility functions.

3.3.2 Implementing Instructions

In the last section we have seen, how the code generator abstracts basic floating point instructions into high-level facility functions from which all *dslash* kernels can be built exactly once, independently of the actual ISA the hardware provides.

The specialisation to the ISA one wants to use is done through the implementation of the `serialise()` function for every instruction class and different `VECLEN` and floating point precisions, in files like `inst_dp_vec8.cc` and `inst_sp_vec16.cc`, cf. Fig. 3.5. The latter two files, implement the instructions for KNC (using IMCI) and KNL (using AVX-512). In these cases the `FVec` type is set to `__m512`, to use the ZMM registers of the Xeon Phi's, and is further specialised to floats and doubles, cf. Tab. 2.1.

Lst. 3.15 illustrates this procedure again for the example of a fused multiply add, both with and without the use of a mask. Finally, Lst. 3.16 illustrates the mechanisms of the up-conversion when using half precision type variables. Note the difference of the implementation for KNC and KNL.

3.3.3 The *dslash* Body

The facilities for the body of the *dslash* kernel are implemented in the two files `dslash.cc` and `dslash_common.cc`. Lst. 3.17 shows the part where the instructions for the computation are implemented. We have suppressed some code fragments, which are used to simulate hardware

```
string FMAdd::serialize() const
{
  if(mask.empty()) {
    return ret.getName() + " = _mm512_fmadd_pd(" + a.getName() +
      ", " + b.getName() + ", " + c.getName() + ");" ;
  }
  else {
    return ret.getName() + " = _mm512_mask_mov_pd(" +
      ret.getName() + ", " + mask + ", _mm512_fmadd_pd(" +
      a.getName() + ", " + b.getName() + ", " + c.getName() + "));" ;
  }
}
```

Listing 3.15: The actual implementation of the fused multiply add instruction using AVX-512 intrinsics. [`inst_dp_vec8.cc`]

```
string LoadBroadcast::serialize() const
{
  std::ostringstream buf;
#ifdef AVX512 // KNL
  if(!a->isHalfType()) {
    buf << v.getName() << " = _mm512_set1_pd(*" << a->serialize()
        << ");" << std::endl;
  }
  else {
    buf << v.getName() << " = _mm512_cvtpslo_pd(_mm512_set1_ps(*"
        << a->serialize() << "));" << std::endl;
  }
#else // KNC
  if(!a->isHalfType()) {
    buf << v.getName() << " = _mm512_extload_pd(" << a->serialize()
    << ", _MM_UPCONV_PD_NONE, _MM_BROADCAST_1X8, _MM_HINT_NONE);"
    << std::endl;
  }
  else {
    buf << v.getName() << " = _mm512_cvtpslo_pd(_mm512_extload_ps("
    << a->serialize()
    << ", _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE));"
    << std::endl;
  }
#endif
  return buf.str();
}
```

Listing 3.16: A load broadcast, including the case of half precision, on both Xeon Phi's KNC and KNL. [`inst_dp_vec8.cc`]

masking on platforms other than Xeon Phi's. The explicit use of masks is required when processing faces in a multi-node implementation, as we have explained above.

Apart from the additional use of the coefficient b (which is called `beta_vec` in the code), that is multiplied when accumulating the result and necessary to construct the routine Eqn. (3.2), the code resembles very much our basic algorithm Alg. 2. In particular, for every site, we have to loop over the eight nearest neighbours (four dimension times two directions) and decide whether to use a standard $SU(3)$ multiplication or its adjungated form. Then, we have to project to half spinors, load the gauge fields and multiply them with the spinors, and finally reconstruct the lower half spinor and accumulate to the result spinor. Note that the generated code will not contain any `for` loops, rather, they are unrolled automatically when generation the code.

Let us have a closer look at the matrix multiplication to illustrate the use of the utility functions we have seen earlier. When creating a kernel, declarations for all single components of the gauge and spinor fields (spin, colour, complex) will be generated. They are then repacked into arrays, so that they resemble the index structure of individual blocks as used in QPhiX. They carry names like `b_spinor` (the projected spinor) and `ub_spinor` (the spinor resulting from the matrix multiplication) and are global variables within the code generator.

Lst. 3.18 shows the matrix multiplication of one $SU(3)$ colour matrix with one spin-component `s` of the spinor (which is by itself a 3-component colour vector). This routine will then be called on a half spinor such that `s` assumes values 0 and 1. Note that no actual conjugation takes places when using an adjungated multiplication, and again that the instructions in the actual code will be a continuous stream of instructions without any loops. The actual calculation is done by using the utility functions for a complex vector multiplication and a complex vector fused multiply add, respectively. This shows that roughly 2/3 of the floating point instructions can be done using the hardware FMA operation.

3.3.4 Software Prefetches

QPhiX-codegen implements software prefetches into the L1 and L2 cache separately. In case of the L1 cache, full spinors are prefetches directly prior to their element-wise load and utilisation.

In the case of the L2 cache, prefetches have to be scheduled earlier. The aim in this case is to prefetch neighbouring spinors from successive spinor blocks, which will be processed later. To this end, up to 4 pointers `pfBase1`, `pfBase2`, …, to the successive base spinors and respective offsets `siprefdist1`, `siprefdist2`, …, to their neighbours are used to locate the prefetch destination. The same mechanism is used to generate L2 prefetches for gauges, the output spinor and clover terms. All base pointers and offsets are calculated during the lattice traversal, as we have explained above.

Other than the L1 prefetches, L2 prefetches are generated into a separate `InstVector` and then merged with the main instructions to achieve an even spacing between the prefetches.

3.3.5 Code Generation & Custom Make Files

To finally generate the code, the desired architecture `ARCH` as to be set at compile time. To this end there is a main `Makefile` in which the target is set via a variable `mode`. This will include a second makefile `customMake.ARCH` which provides a set of hardware options which are specific to the architecture. Their value can be set here and will be included in main makefile when the code is compiled. To do this, the main makefile provides a specific target for all supported architectures.

Lst. 3.20 shows the custom makefile for the KNC and KNL. Amongst others, it allows to switch

```
void dslash_body(InstVector& ivector, bool compress12,
    proj_ops* ops, recons_ops* rec_ops_bw,
    recons_ops* rec_ops_fw, FVec outspinor[4][3][2])
{
  for(int dim = 0; dim < 4; dim++) {
    for(int dir = 0; dir < 2; dir++) {

      int d = dim * 2 + dir;
      bool adjMul = (dir == 0 ? true : false);
      recons_ops rec_op =
        (dir == 0 ? rec_ops_bw[dim] : rec_ops_fw[dim]);

      ifStatement(ivector, "accumulate["+std::to_string(d)+"]");
      {
        declareFVecFromFVec(ivector, beta_vec);
        loadBroadcastScalar(ivector, beta_vec, beta_names[d],
            SpinorType);

        std::string mask;
        if(requireAllOneCheck[dim]) {
          mask = "accMask";
          declareMask(ivector, mask);
          intToMask(ivector, mask,
              "accumulate["+std::to_string(d)+"]");
        }

        project(ivector, basenames[d], offsnames[d],
            ops[d], false, mask, d);
        loadGaugeDir(ivector, d, compress12);
        matMultVec(ivector, adjMul);
        recons_add(ivector, rec_op, outspinor, mask);
      }
      endScope(ivector);
    }
  }
}
```

Listing 3.17: The code generation routine for the dslash body. Note the very similar structure to the basic dslash algorithm in Alg. 2. [`dslash.cc`]

```
void matMultVec(InstVector& ivector, bool adjMul, int s)
{
  std::string mask = "";
  for(int c1 = 0; c1 < 3; c1++) {
    if(!adjMul) {
      mulCVec(ivector, ub_spinor[s][c1], u_gauge[0][c1],
          b_spinor[s][0], mask);
      fmaddCVec(ivector, ub_spinor[s][c1], u_gauge[1][c1],
          b_spinor[s][1], ub_spinor[s][c1], mask);
      fmaddCVec(ivector, ub_spinor[s][c1], u_gauge[2][c1],
          b_spinor[s][2], ub_spinor[s][c1], mask);
    }
    else {
      mulConjCVec(ivector, ub_spinor[s][c1], u_gauge[c1][0],
          b_spinor[s][0], mask);
      fmaddConjCVec(ivector, ub_spinor[s][c1], u_gauge[c1][1],
          b_spinor[s][1], ub_spinor[s][c1], mask);
      fmaddConjCVec(ivector, ub_spinor[s][c1], u_gauge[c1][2],
          b_spinor[s][2], ub_spinor[s][c1], mask);
    }
  }
}
```

Listing 3.18: Colour matrix-vector multiplication in the QPhiX code generator. [`dslash_common.cc`]

```
void generateL2Prefetches(InstVector& ivector, bool compress12, bool chi, bool
    clover)
{
  PrefetchL2FullSpinorDirIn(ivector, "xyBase", "pfyOffs",
      "siprefdist1");
  PrefetchL2FullSpinorDirIn(ivector, "pfBase1", "offs",
      "siprefdist1");
  PrefetchL2FullSpinorDirIn(ivector, "pfBase2", "offs",
      "siprefdist2");

  {...}

  PrefetchL2FullGaugeIn(ivector, "gBase", "gOffs", "gprefdist",
      compress12);
  PrefetchL2FullSpinorOut(ivector, outBase, "offs", "siprefdist4");
}
```

Listing 3.19: L2 prefetches are generated from successive base pointers, and offset to their neighbours, for spinors, gauges and clover terms. [`dslash.cc`]

```
mode = mic

# Generate AVX512 code for KNL
AVX512 = 0

# Define compute precision (1=float, 2=double)
PRECISION ?= 1

# Enable Lower Precision if set to 1
ENABLE_LOW_PRECISION ?= 0

# Define SOA length
SOALEN ?= 8

# Enable serial spin compute in Dslash
SERIAL_SPIN = 1

# Prefetching options
# for KNC set all these to 1
# for KNL set all these to 0
PREF_L1_SPINOR_IN = 1
PREF_L2_SPINOR_IN = 1
PREF_L1_SPINOR_OUT = 1
PREF_L2_SPINOR_OUT = 1
PREF_L1_GAUGE = 1
PREF_L2_GAUGE = 1
PREF_L1_CLOVER = 1
PREF_L2_CLOVER = 1

# Gather / Scatter options
USE_LDUNPK = 1     # Use loadunpack instead of gather
USE_PKST = 1       # Use packstore instead of scatter
USE_SHUFFLES = 0   # Use loads & Shuffles to transpose spinor
                   # when SOALEN>4
NO_GPREF_L1 = 1    # Generate bunch of normal prefetches instead of
                   # one gather prefetch for L1
NO_GPREF_L2 = 1    # Generate bunch of normal prefetches instead of
                   # one gather prefetch for L2

# Enable nontemporal streaming stores
ENABLE_STREAMING_STORES ?= 1
USE_PACKED_GAUGES ?= 1     # Use 2D xy packing for Gauges
USE_PACKED_CLOVER ?= 1     # Use 2D xy packing for Clover
```

Listing 3.20: Custom Makefile for KNC and KNL. When using KNL, the AVX-512 instruction set architecture should be switched on, and software prefetches turned off. [`customMake.mic`]

on software prefetches into the L1 and L2 cache for the various data types, set the `SOALEN`, and to use streaming stores and gather/scatter instructions.

Once the code is built and run, it will generate files for each kernel in the subdirectory `ARCH/`. These files contain the body of the function, but no argument lists, prototypes, etc. They then have to be transferred into the QPhiX repository, into the direction `include/qphix/ARCH/generated/`, where they will be included via the use of template specialisation carried out through C preprocessing macros. There, the function headers will be added.

3.4 Extending QPhiX for Twisted-Mass

In order to extend QPhiX to the case of twisted-mass fermions with or without clover term, the first thing one has to change are the basic kernel routines.

This is so, because the inverse of the even-even part of the even-odd pre-conditioned matrix is no longer proportional to the identity matrix, not even without the clover term ($c_{sw} = 0$), c.f. Sec. 1.2.4, and in particular Eqn. (1.62). For this reason, the first of the kernels one has to implement, always assumes the form

$$y = A^{-1}\, \not{D}\, x\,, \tag{3.3}$$

where the specifics of the multiplication with A^{-1} are different, depending whether the clover term is used or not. The same holds true for the first term of the second routine

$$z = A\, x - b\, \not{D}\, y\,. \tag{3.4}$$

This time, the odd-odd part of the clover term is multiplied with the first input spinor. However, since A and A^{-1} are passed to the constructors in the `EvenOddLinearOperator` externally, one only has to implement the multiplication once.

3.4.1 Code Generation for pure Twisted-Mass Fermions

To understand better, what kind of functionality we have to modify and extend, let us have a closer look, how the kernels are precisely generated. Lst. 3.21 shows our expanded and restructured version of the loop over all the cases which are necessary to generate the aforementioned two kernel routines. We will focus here only on the respective body versions.

We have to distinguish all the cases with or without twisted-mass and clover term, the form of the kernel, if it applies the normal or hermitian conjugate operator, and if it uses gauge compression. Then, we create two `InstVector`'s and construct the file name for each kernel. Finally, we generate L2 prefetches into one of the instruction vectors, and the main calculation into the other. The two will be merged, and eventually written to a file.

To extend all existing facilities, we have to add two switches or flags which indicate that we want to use either pure twisted-mass or twisted-mass-clover. Then, one has to add all the utility functions which generate L1 and L2 prefetches, as well as type declarations and so forth.

Let us have a closer look, how the routine to generate the plain body of *dslash* changes, depending on the lattice action we want to use, cf. Lst. 3.22. After declaration of in- and output variables as SIMD vectors, the result spinor is zeroed-out, and projection and reconstruction operations are initialised. With all these parameters the generation of the main *dslash* kernel is

```
// DSLASH and DSLASH_ACHIMBDPSI ROUTINES
// ====================================
for(auto twisted_mass : {true, false}) {
  for(auto clover : {true, false}) {
    for(auto kernel : {"dslash", "dslash_achimbdpsi"}) {
      for(auto isPlus : {true, false}) {
        for(auto compress12 : {true, false}) {

          InstVector ivector;
          InstVector l2prefs;
          std::ostringstream filename;

          std::string tmf_prefix  = twisted_mass ? "tmf_" : "";
          std::string clov_prefix = clover ?
            "clov_"+CloverTypeName+"_" : "";
          std::string plusminus   = isPlus ? "plus" : "minus";
          int num_components  = compress12 ? 12 : 18;
          bool chi_prefetches = (kernel == "dslash_achimbdpsi") ?
            true : false;

          filename << "./" << ARCH_NAME << "/" << tmf_prefix
            << clov_prefix << kernel << "_" << plusminus << "_"
            << "body" << "_" << SpinorTypeName << "_"
            << GaugeTypeName << "_v" << VECLEN << "_s"
            << SOALEN << "_" << num_components;

          // Generate Instructions
          generateL2Prefetches(l2prefs, compress12, chi_prefetches,
              clover, twisted_mass);
          if(kernel == "dslash")
            dslash_plain_body(ivector, compress12, clover,
                twisted_mass, isPlus);
          else if(kernel == "dslash_achimbdpsi")
            dslash_achimbdpsi_body(ivector, compress12, clover,
                twisted_mass, isPlus);
          mergeIvectorWithL2Prefetches(ivector, l2prefs);
          dumpIVector(ivector, filename.str());

        } // gauge compression
      } // plus/minus
    } // kernel
  } // clover
} // twisted_mass
```

Listing 3.21: Scheduling of the code generation for the kernels acting on the body. [`dslash.cc`]

```
void dslash_plain_body(InstVector& ivector, bool compress12,
    bool clover, bool twisted_mass, bool isPlus)
{
  declare_b_Spins(ivector);
  declare_ub_Spins(ivector);
  declare_u_gaus(ivector);
  declare_misc(ivector);
  declare_outs(ivector);

  if(clover) declare_douts(ivector);

  if(clover && !twisted_mass) {
    declare_clover(ivector);
  } else if(clover && twisted_mass) {
    declare_full_clover(ivector);
  }

  FVec (*outspinor)[4][3][2];

  if(clover) {
    outspinor = &dout_spinor;
  } else {
    outspinor = &out_spinor;
  }

  zeroResult(ivector, (*outspinor)[0][0]);

  proj_ops *p_ops;
  recons_ops *rec_ops_bw;
  recons_ops *rec_ops_fw;

  if(isPlus) {
    p_ops      = proj_ops_plus;
    rec_ops_bw = rec_plus_pbeta_ops;
    rec_ops_fw = rec_minus_pbeta_ops;
  } else {
    p_ops      = proj_ops_minus;
    rec_ops_bw = rec_minus_pbeta_ops;
    rec_ops_fw = rec_plus_pbeta_ops;
  }

  dslash_body(ivector, compress12, p_ops, rec_ops_bw,
      rec_ops_fw, *outspinor);

  if(clover && !twisted_mass)
    clover_term(ivector, *outspinor, false);
  else if(clover && twisted_mass)
    full_clover_term(ivector, *outspinor, false);
  else if(!clover && twisted_mass)
    twisted_term(ivector, *outspinor, false, isPlus);
  else if(!clover && !twisted_mass) {};

  StreamFullSpinor(ivector, out_spinor, outBase, outOffs);
}
```

Listing 3.22: Complete dslash generation, depending on the lattice action and gauge compression. [`dslash_common.cc`]

carried out, which corresponds to the hopping part of the full fermion matrix, as we have seen it in the last section.

The interesting part follows now: the multiplication with the matrix A^{-1} mentioned above. In the case of pure Wilson fermion, this is essentially a constant factor, which is dealt with in the high-level functions. The case of the clover term has already been implemented, and we will come back to this case shortly. For the case of pure twisted-mass fermions we have

$$A^{-1} = (\alpha \mathbb{1} \pm i\mu\gamma_5)^{-1} = \frac{\mathbb{1} \mp i\mu\gamma_5}{\alpha + \mu^2} \,. \tag{3.5}$$

In particular, we have to implement a colour matrix times vector multiplication for a full spinor, because the lower half spinors have already been reconstructed. This is most easily done by multiplying with the matrix in the numerator first, and then rescaling with the factor in the denominator. The first part can be done with a FMA for each real and complex component, whereas the latter will be a multiplication with a variable `mu_inv`, which is passed to the kernel externally and will be calculated at some higher level. Since γ_5 is block-diagonal and simply given by $\mathrm{diag}(\mathbb{1}_2, -\mathbb{1}_2)$, the first spin component of the matrix product reads

$$(\alpha + \mu^2)\,\mathrm{Re}\,(A^{-1}\psi_0) = \mu * \mathrm{Im}\,\psi_0 + \mathrm{Re}\,\psi_0 \,, \tag{3.6}$$

$$(\alpha + \mu^2)\,\mathrm{Im}\,(A^{-1}\psi_0) = \mu * \mathrm{Re}\,\psi_0 - \mathrm{Im}\,\psi_0 \,. \tag{3.7}$$

A part of the full implementation is shown in Lst. 3.23, and has been written by Mario Schröck, as well as the rest of the machinery for pure twisted-mass fermions [63].

3.4.2 Data Types & Multiplication of Twisted-Mass Clover

The situation is a little bit more complicated in the case of the clover term. Here we have two major differences with respect to the Wilson action: First of all, we necessarily have to simulate two degenerate fermions at the same time, which differ by the sign in front of the twisted-mass term

$$A = \alpha \mathbb{1} \pm i\mu\gamma_5 + c_{sw}T \,, \tag{3.8}$$

and thus we need two clover terms and two clover inverses. Strictly speaking, this is already true when $c_{sw} = 0$, but there, A^{-1} is analytically related to A, which is not true for $c_{sw} \neq 0$.

Secondly, due to the imaginary addition of μ to the diagonal, A is no longer hermitian in the case of twisted-mass fermions. In particular, the inverse of the clover term A^{-1}, will not even respect the particular form of A, such that the upper and lower triangular part of the matrix are not self-adjoint.

With the first problem we can deal in QPhiX at the level of the service functions, because each kernel still only involves one clover term, and one inverse. The second issue however, forces us to change the data layout of the clover term, and with it all the low-level utilities and the main matrix multiplication.

Recall that the clover term is composed out of two 6×6 blocks (*i.e.* half spinor $\times$ colour vector). In QPhiX, these blocks are stored in form of the 6 real diagonal elements and the 15 upper triangular complex elements, cf. Lst. 3.24. In order not to have to introduce distinct data structures for the twisted-mass clover term and its inverse, we decided to use a data layout, in which we store

```
void twisted_term(InstVector& ivector, FVec in_spinor[4][3][2],
    bool face, bool isPlus, std::string mask)
{
  declareFVecFromFVec(ivector,mu_vec);
  declareFVecFromFVec(ivector,mu_inv_vec);
  loadBroadcastScalar(ivector,mu_vec,mu_name,SpinorType);
  loadBroadcastScalar(ivector,mu_inv_vec,mu_inv_name,SpinorType);

  for(int col = 0; col < 3; col++) {
    for(int spin = 0; spin < 2; spin++) {
      FVec *sp = in_spinor[spin][col];
      FVec *tmout = out_spinor[spin][col];

      if(isPlus){
        fmaddFVec(ivector,tmp_1_re,mu_vec,sp[IM],sp[RE],mask);
        fnmaddFVec(ivector,tmp_1_im,mu_vec,sp[RE],sp[IM],mask);
      }
      else{
        fnmaddFVec(ivector,tmp_1_re,mu_vec,sp[IM],sp[RE],mask);
        fmaddFVec(ivector,tmp_1_im,mu_vec,sp[RE],sp[IM],mask);
      }

      if(face) {
        fmaddFVec(ivector,tmout[RE],mu_inv_vec,tmp_1_re,
            tmout[RE],mask);
        fmaddFVec(ivector,tmout[IM],mu_inv_vec,tmp_1_im,
            tmout[IM],mask);
      }
      else {
        mulFVec(ivector,sp[RE],mu_inv_vec,tmp_1_re,mask);
        mulFVec(ivector,sp[IM],mu_inv_vec,tmp_1_im,mask);
      }
    }
  }
  {...}
}
```

Listing 3.23: Implementation of the twisted-mass term. First, the coefficients are promoted to SIMD vectors, then the multiplication is carried out in two steps. Face values are accumulated to final result. [`dslash_common.cc`]

```
typedef struct {
  FT diag1[6][VECLEN];
  FT off_diag1[15][2][VECLEN];
  FT diag2[6][VECLEN];
  FT off_diag2[15][2][VECLEN];
} Clover;
```

Listing 3.24: Wilson clover term with its two blocks of 6 real diagonal and 15 complex off-diagonal elements. Here, tiles are repacked as in the case of the gauges. [`data_types.h`]

```
typedef struct {
  FT block1[6][6][2][VECLEN];
  FT block2[6][6][2][VECLEN];
} FullClover;
```

Listing 3.25: Twisted-mass clover term with two full 6×6 complex blocks. [`data_types.h`]

two full 6×6 complex blocks. This new data structure is called `FullClover` and we show it in Lst. 3.25.

With this new data structure, we implemented the clover-term-spinor multiplication in the routine `full_clover_term`, which is needed *after* the plain *dslash*, as we have seen in Lst. 3.22. Here, we loop over the two blocks, prefetch one full clover block into the L1 and load it into the vector register. The spinor components should already be in the registers, and thus do not have to be prefetched and loaded. For each of the blocks, we calculate spin and colour index for both input and output spinor. Finally, the multiplication is carried out with complex FMA's or a standard complex multiplication, in case one starts to accumulate to a new spinor component. Note that the face version always uses FMA's, because it is an accumulation to the calculation of the body, which has already been carried out by that time. The full implementation can be found in Lst. 3.26.

3.4.3 Declarations & L1/L2 Prefetches

To use the clover multiplication generator, we also had to implement low-level utility functions, `Address` classes and `FVec`'s to provide facilities for declaration, loads and prefetches to L1 and L2 of `FullClover` terms.

This turns out to be a bit tedious and requires rather good understanding of the code, but much of the existing functionality could be mimicked. We show the example of the L1 prefetch machinery in Lst. 3.27. Here, the implementation of a new `FullCloverAddress` class was skipped to save space.

3.4.4 Extending the High-Level Facilities

With the expansion of the code generator, new kernels could be generated and imported into the QPhiX file structure.

The extension of the QPhiX library turned out to be rather straight-forward due to the well-organised software design. For both cases we have implemented one new `EvenOddLinearOp-erator` class each, which can directly be used for the iterative solvers as well.

The first one, `EvenOddTMWilsonOperator`, has two additional member variables, `dou-ble Mu` and `bool mu_plus`, which allow the user to set the sign and value of the twisted-mass parameter μ. The inverse parameter is then automatically calculated and passed to the kernel acting on the tiles. For the second one, `EvenOddTMCloverOperator`, we had to integrate the new data structure, which we used already in the code generator, into QPhiX, and provide memory facility functions for array allocation and free. As mentioned before, we also had to pass two clover terms and two inverses to the constructor. The right one is then automatically deduced from the `isign` the user provides, cf. Lst. 3.28. This is possible, because one clover term is the hermitian conjugate of the other.

The lattice traversal was replicated largely unchanged. This could be a very apt starting place to refactor the library in the future. We also had to manage the `#include`'s of the newly generated

```
void full_clover_term(InstVector& ivector, FVec in_spinor[4][3][2], bool face,
    std::string mask)
{
  for(int block=0; block<2; block++) {

    PrefetchL1FullCloverFullBlockIn(ivector,clBase,clOffs,block);
    LoadFullCloverFullBlock(ivector,clov_full,clBase,clOffs,block);

    for(int sc1=0; sc1<6; sc1++) { // half-spin-colour row

      int spin_out = 2*block + sc1/3;
      int col_out  = sc1 % 3;
      FVec *clout  = out_spinor[spin_out][col_out];

      for(int sc2=0; sc2<6; sc2++) { // half-spin-colour column

        int spin_in = 2*block + sc2/3;
        int col_in  = sc2 % 3;
        FVec *clin  = in_spinor[spin_in][col_in];

        if(sc2 == 0 && !face) {
          mulCVec(ivector,clout,clov_full[sc1][sc2],clin,mask);
        } else {
          fmaddCVec(ivector,clout,clov_full[sc1][sc2],clin,clout,
              mask);
        }

      } // half-spin-colour column
    } // half-spin-colour row
  } // block
}
```

Listing 3.26: The clover-term-spinor multiplication in the non-hermitian case with twisted-mass. [`dslash_common.cc`]

```
void PrefetchL1FullCloverFullBlockIn(InstVector& ivector,
    const std::string& base, const std::string& off, int block)
{
  int nSites=VECLEN;
  for(int i=0; i<((36*nSites*sizeof(CloverBaseType)+63)/64); i++) {
    prefetchL1CloverFullBlockIn(ivector,base,off,block,
        i*(64/sizeof(CloverBaseType)));
  }
}

void prefetchL1CloverFullBlockIn(InstVector& ivector,
    std::string base, std::string off, int block, int imm)
{
  prefetchL1(ivector, new AddressImm
      (new FullClovAddress(base,block,0,0,RE,CloverType),
       imm), 0);
}
```

Listing 3.27: Example of a `FullClover` L1 prefetch using the `Address` factory chain. Here, we prefetch one block of 36 (SIMD vector) elements, aligned to 64 bits. (Version for packed clover field only.) [`data_types.cc`]

```
// The operator() that the user sees
  template<typename FT, int VECLEN, int SOALEN, bool COMPRESS12>
void TMClovDslash<FT,VECLEN,SOALEN,COMPRESS12>::dslash(FourSpinorBlock* res,
    const FourSpinorBlock* psi,
    const SU3MatrixBlock* u,
    const FullCloverBlock* invclov[2],
    int isign,
    int cb)
{
  // Call the service functions
  if(isign == 1)   DPsiPlus(u, invclov[0], psi, res, cb);
  if(isign == -1) DPsiMinus(u, invclov[1], psi, res, cb);
}
```

Listing 3.28: The dslash selects the right inverse term depending on `isign`. The same is true for the clover term itself in `dslashAChiMinusBDPsi`. [`tm_clov_dslash_body.h`]

kernels, and the specialisation files for the Xeon Phi architectures and the scalar version for testing. We could use some amount of BASH scripting to automate parts of the work. We also had to update the autotools setup, adding new configure flags to be able to compile with twisted-mass and twisted-mass-clover, independently of the old QPhiX facilities.

Finally, we adapted the existing testing and timing facilities to verify correctness of our implementation and benchmark its performance. We successfully tested against the QDP++ kernels with $\mu = 0$, after implementing new packing routines, to reorder the QDP++ to the QPhiX layout. Once an interface for tmLQCD is ready, the kernels should also be tested with $\mu \neq 0$.

4. Benchmarks

In this chapter we want to present the results of the benchmarks carried out on the KNL test cluster DEEP of the Supercomputing Center Jülich [8].

We performed numerous benchmarks on single nodes with various run-time and compile-time options. We will present the results for the best known options for all four kernels using different floating point precision first. Then, we will evaluate how different compile-time optimisation options change the performance of the code. Unfortunately, we could not perform any convincing multi-node benchmarks on the small test cluster, but we will discuss older results for the KNC's and Intel Xeon dual-socket nodes, and in particular their weak and strong scaling behaviour.

4.1 Single Node Results

The available KNL's had 64 cores with a clock frequency of 1.4 GHz. They were run in flat memory mode where all of the 16 GB HBM is allocatable. In addition, there were $6 * 16 = 96$ GB DDR memory available. The cluster mode was set to quadrant mode. These options could not be changed, but they can be checked with `numactl --hardware`. In particular, to prevent the operating system to allocate all its data in the MCDRAM, the distance to the HBM is set larger than the one to the DDR-RAM. This is why one has to use `numactl -m 1`, to allocate all of a program's data to the fast memory. Note, that `-m 1` allocates into the *second* NUMA-node. Without explicit allocation in MCDRAM, we observed a performance loss in our benchmarks of at least a factor of two.

All code was compiled with the Intel C++ compile icpc both in version 16.3 and 17.1. We could not determine significant differences in performance between this two versions. To build QPhiX and its testing facilities several library dependencies have to be resolved (cf. Fig. 4.1). They include in particular the QDP++, QMP and QIO libraries, as we have seen in the last chapter. We have built these dependencies on top of Intel MPI and used `-enable-parallel-arch=parscalar` to

```
QPhiX
├── QDP++
│   ├── libxml2 (used version 2.9.4)
│   ├── QIO
│   │   ├── QMP
│   │   └── c-lime
│   ├── libintrin
│   ├── filedb
│   └── xpathreader
└── QMP
```

Figure 4.1: Full dependencies of QPhiX. Note that QMP is used independently if testing routines are built or not.

enable OMP threading in QDP++.

We have compiled the code with the following flags:

```
CXXFLAGS="-std=c++11 -qopenmp -xMIC-AVX512
  -restrict -finline-functions -fno-alias -O3"
```

Note in particular the use of `-xMIC-AVX512` which is necessary to specify the ISA for the compiler, and without which certain architecture specific prefetch hints will not be recognised. If not specified differently, we used the `-enable-cean -enable-mm-malloc` configure options to enable Cilk Array Notation and aligned memory allocations. We also had to set the `SOALEN` parameter, and the architecture with `-enable-proc=AVX512` at configure-time.

We performed single node benchmarks for the four *dslash* kernels using Wilson, Wilson Clover, Twisted-Mass and Twisted-Mass Clover Fermions. Each kernels was compiled as a scalar version (`SOA1`) and with the values $\texttt{SOALEN} = 4, 8, 16$. The tests have been carried out with the three types of precision half, single and double. Note that there is no half precision available for the scalar version, as there is no double version when compiling with $\texttt{SOALEN} = 16$.

We used three different lattice volumes, being $32^3 \times 64$, $32^3 \times 96$ and $48^3 \times 96$. However, all results were very similar, which is why we will only display the results for the first lattice. For each given kernel, `SOALEN` and precision, the runs consisted of 3 micro-benchmarks for each value of `cb` (checkerboard) and `isign` (hermitian conjugate nor not). These 12 values measured the execution of 500 applications of the stencil operator each. We took the arithmetic mean of these values and verified that the standard deviation was considerably small.

We set the following OpenMP environment variables at start-up:

```
export KMP_AFFINITY=compact,granularity=fine
export KMP_PLACE_THREADS=64c,4t
export OMP_NUM_THREADS=256
```

The SMT threading scheme, QMP (multi-node), lattice, blocking and padding parameters, as well

Precision	Wilson	Twisted-Mass	Wilson Clover	Twisted-Mass Clover
Single	483 (8)	504 (8)	481 (8)	366 (8)
Double	232 (4)	242 (4)	236 (4)	178 (4)

Table 4.1: Highest Performance in GFlops/s for the different kernels. The number in parenthesis indicates for which value of `SOALEN` *it has been achieved.*

as the use of gauge compression could be set at run time. A typical program call could look like

```
numactl -m 1 ./wils_soa8 -geom 1 1 1 1 -c 64 -sy 1 -sz 4
      -x 32 -y 32 -z 32 -t 64 -by 4 -bz 4 -pxy 1 -pxyz 0
             -minct 1 -compress12 -i 500 -prec d -dslash
```

to call the `dslash` in double precision for 12 micro-benchmarks and 500 iterations each. We present our results in Figs. 4.2–4.5.

There are some comments in order. First of all, it is clearly visible that SIMD has a huge impact on performance. Running the code without using any SIMD vectors, the performance for single and double precision is nearly always the same. The reason for this might be that the code is instruction bound when using only scalar instructions.

Switching to vector instructions, the code becomes memory bandwidth bound, but the speed-up with respect to the scalar version is still a solid 8x in single and 4x in double precision. It is also interesting to note, that in single and double precision the `SOALEN` parameter does not seem to have a large impact, although best performance is always obtained when using SoA's where the number of elements in the X-direction fills exactly half of the register, cf. Tab. 4.1. This seems counter-intuitive, because one needs to load data in two steps into the registers, instead of one. However, the same behaviour as been observed on KNC's, cf. [44], as well.

The length of the SoA's *does* have an impact in half precision, because there, this parameter effectively controls the fraction of the register which is actually utilised. When using half precision arithmetics with the full registers, a significantly higher throughput compared to the single precision version is clearly visible for all the kernels.

Our benchmark results for the Wilson *dslash* kernel are very similar to what has been measured in [55], where KNL's with 68 rather than 64 cores where used. The throughput for the clover version of the *dslash* in the case of Wilson Fermions is on average a little bit higher than the one without clover term, as one would expect from the higher total arithmetic intensity. In single precision with gauge compression, the *combined* arithmetic intensity is $I_A = 1.06$ without and $I_A = 1.17$ with clover term, cf. Tabs. 1.2 and 1.3. The same holds true for the Twisted-Mass Clover term routine, which performs about 24% worse than the respective Wilson version. This is rather close to the 18% loss one expects from the pure difference in the arithmetic intensities.

We also observe that the throughput values we estimated in our performance model, which took hardware considerations into account, come rather close to the values we have measured in our benchmarks,[1] cf. Tab. 2.2. In particular, we see that we achieve about 60% of the rooftop

[1]Excluding obviously the cases of half precision, because the respective arithmetic operations are not implemented in hardware for Xeon Phi's.

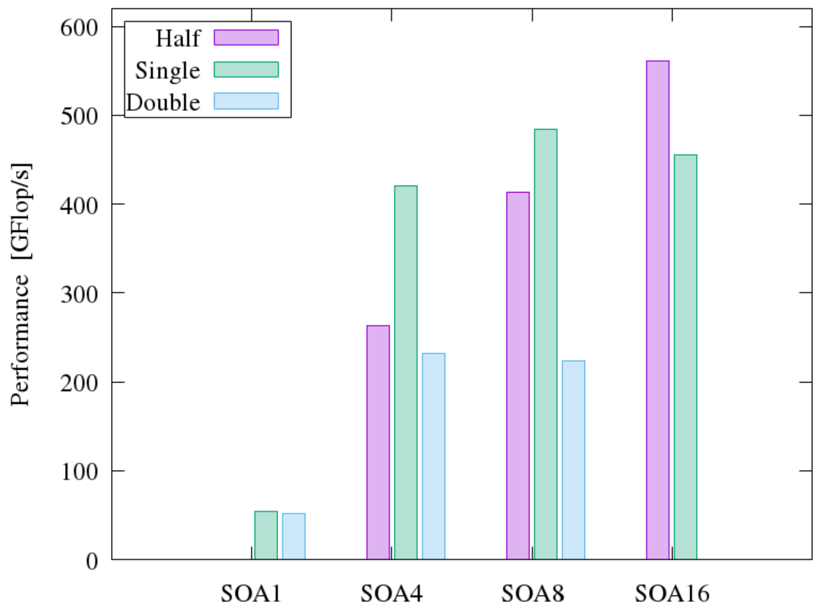

Figure 4.2: Single node performance on KNL for Wilson dslash

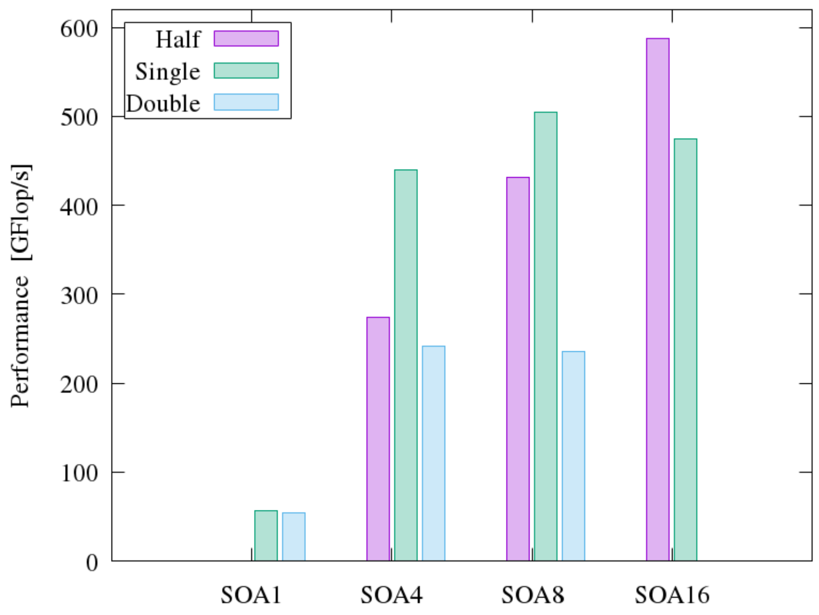

Figure 4.3: Single node performance on KNL for Twisted-Mass dslash

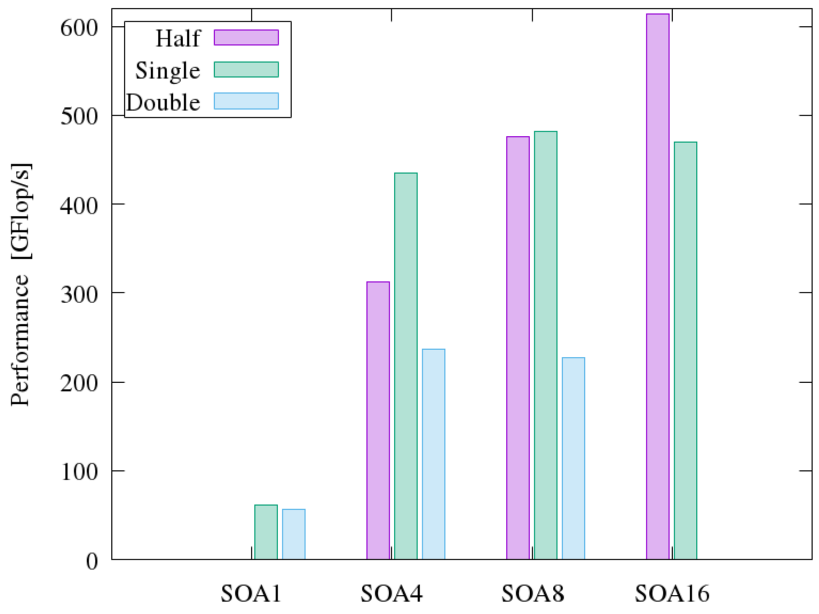

Figure 4.4: Single node performance on KNL for Wilson Clover dslash

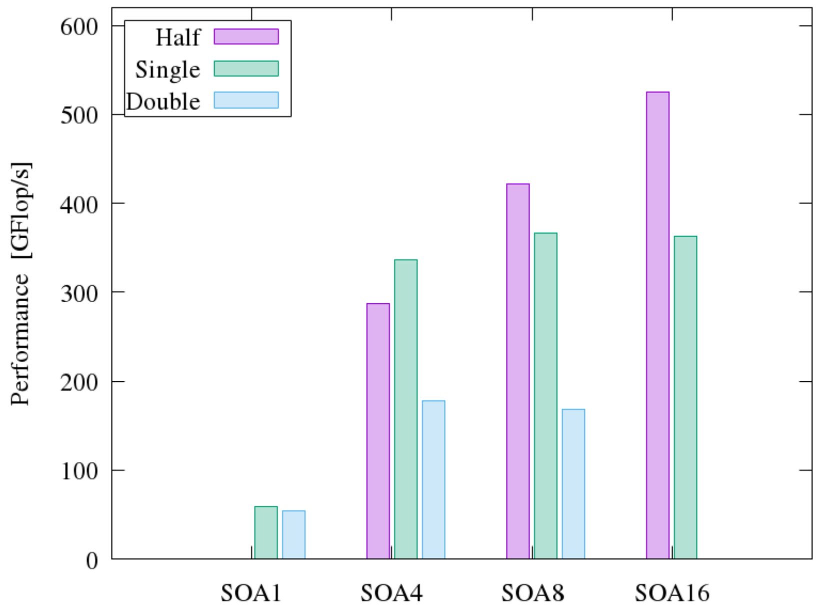

Figure 4.5: Single node performance on KNL for Twisted-Mass Clover dslash

performance for 7 reused spinors, and are significantly above the one without reuse—always considering the use of gauge compression and streaming stores.

4.2 Testing the Kernel Options

To evaluate the impact of the various compile- and run-time options QPhiX has to offer, we carried out benchmarks in single precision and `SOALEN` $= 16$ (unless stated otherwise) with a number of different tuning options. The results on KNL for the Wilson and Twisted-Mass Clover *dslash* are displayed in Figs. 4.6 and 4.7. We supplement these results with older benchmarks obtained on KNC for the Wilson *dslash*, cf. Fig. 4.8.

In the case of the KNL we made several interesting observations. First of all, the impact of repacking gauge and clover fields into full vectors results in a speed-up of about 5% to 7%. However, the *explicit* use of streaming stores does not change the performance in any visible way. We also observed that all software prefetches and in particular L2 prefetches are much better handled by the hardware itself. We also do not observed any gain from introducing light thread barriers for local core synchronisation. We *do* observe a slight improvement in throughput when spinor fields are repacked into tiles with `SOALEN` $= 8$, though (cf. also the results of the last section). The greatest impact, however, has the algorithmic improvement obtained through 2-row gauge compression.

The situation for the KNC, on the other hand, was quite different. Here, managing prefetches for L1 and L2 caches within the software was quite beneficial, as were explicit barriers for thread synchronisation. Also in that case, we observed great gains from both the use of 2D tiles as well as 2-row gauge compression.

Although these results are a little bit disappointing from the software development point of view, they suggested that low level aspects of the software execution are much better handle by the hardware itself in the case of KNL's. In particular, the programmer has to spend much less effort to be able to use low level hardware features such as streaming stores, cache prefetches and the synchronisation of hardware threads. It also shows, however, that slight variations in the data layout (here in form of re-packing and 2D spinor tiles) can give the last few percentages of improvement in performance.

4.3 Multi-Node Results

Unfortunately, we did not have access to hardware facilities to test the scaling behaviour of the code using multiple KNL's. However, there has been extensive testing for both the Wilson kernels [43], as well as the pure Twisted-Mass kernel [63] to quantify both the weak and the strong scalability of the code on KNC's.

For the former case, strong scaling was observed for two lattices with dimensions $32^3 \times 256$ and $48^3 \times 256$ up to 16 KNC units. The bigger volume was scaling up further to 32 KNC's, but also there a topping out was already visible.

For the case of Twisted-Mass Fermions, the code was weakly scaling up to 64 KNC nodes, when using a lattice of dimensions $48^3 \times 96$. Furthermore, strong scaling for various lattice volumes could be observed on Dual Socket Xeon Haswell CPU's using AVX2 intrinsics up to 64 nodes.

In both benchmarks, a proxy provided by Intel was used to improve the MPI communication between KNC nodes. To that end, one of 61 available cores was exclusively used to manage the MPI calls. The above results, and in particular those for the Dual Socket Xeon nodes, suggest that

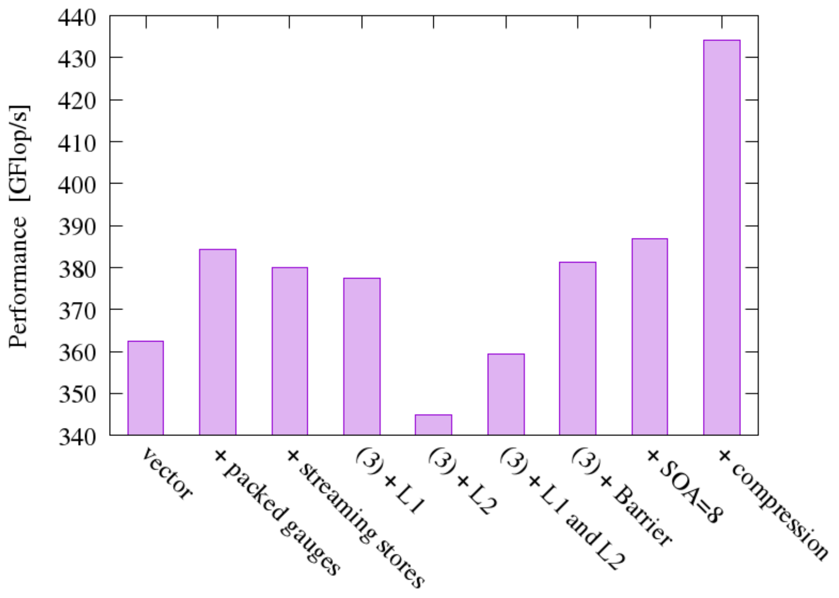

Figure 4.6: Various tuning options for Wilson dslash *on KNL*

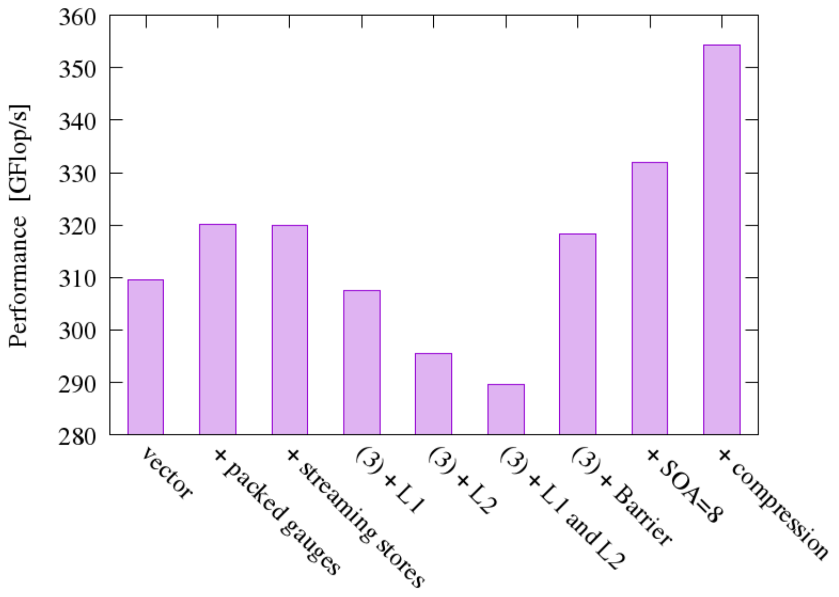

Figure 4.7: Various tuning options for Twisted-Mass Clover dslash *on KNL*

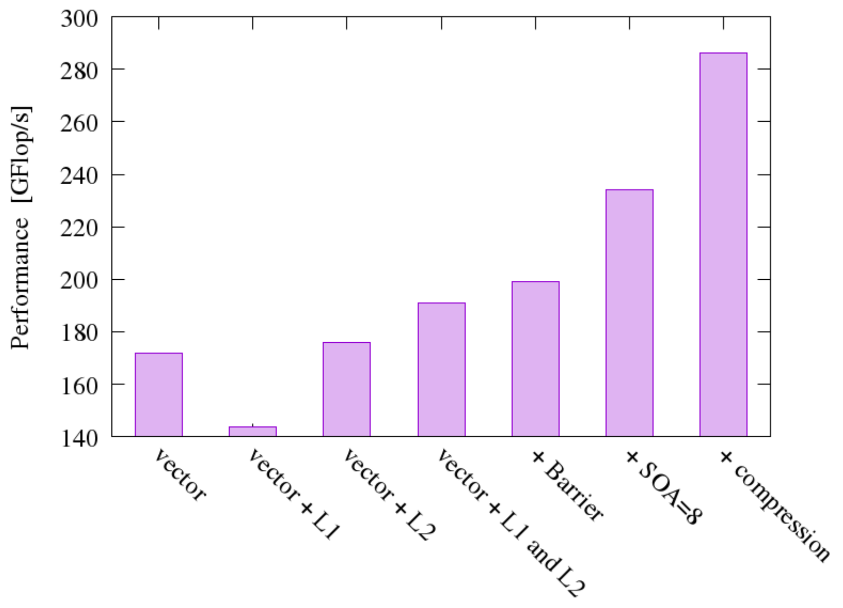

Figure 4.8: Various tuning options for Wilson dslash *on KNC, data taken from [44]. Here vector includes the use of packed gauges and streaming stores.*

good strong scalability will be achieved on KNL processor clusters, which may be interconnected using traditional InfiniBand networks.

Conclusions and Outlook

In this thesis, I extended the QPhiX library facilities to be able to deal with degenerate Twisted-Mass Fermions with Clover Term.

In this work, I gave a brief heuristic introduction to the Quantum Field Theory of Strong Interactions, and in particular its discretised version known as Lattice QCD. I discussed the fundamental degrees of freedom of the theory and their adaption on the lattice.

Subsequently, I considered the various levels of complexity needed to simulate the discretised theory on a computer. In particular, I introduced the *dslash* stencil operator, several different iterative Krylov subspace linear solvers, and the Hybrid Monte Carlo algorithm. I motivated the scaling behaviour of the algorithmic complexity of LQCD in terms of the fundamental physical parameters.

After that, I gave a description of the target hardware of the QPhiX library, the Xeon Phi (co-)processors. I particularly focused on the changes of the KNL architecture with respect to the previous KNC architecture. I reported about the structure of individual CPU's, the interconnecting mesh and the random access memory. I drew a few general implications for the HPC software development on this hardware architecture.

In a subsequent chapter, I described the software design of QPhiX and how it implements the above mentioned guide lines. I spent some time to illustrate the process of extending the library for the twisted-mass fermion action with the use of numerous code examples. In the final chapter, I reported the results of extensive benchmarking, carried out on a single KNL processors on the JSC test cluster DEEP.

Several goals for future developments are already contemplated. In particular, the QPhiX *dslash* facilities should be extended to integrate the non-degenerate twisted-mass fermion action, in order to be able to simulate fermion fields with different mass parameters, *e.g* in a setup with 2+1+1 flavours. Furthermore, since it is likely that supercomputers at the exa-scale will look similar to Xeon Phi clusters, it is desirable to be able to run full HMC simulations on KNL's. To this end, the

QPhiX library needs to be integrated into the existing software suite tmLQCD, which was natively designed to simulated twisted-mass fermions. This work has already been initiated. It would also be conceivable, to go the other way around and extend the software suite Chroma, which also targets to supply functionality for a wide range of physics applications, for twisted-mass fermions. In this case, one could benefit from the integration of QPhiX, which has already been carried out.

Apart from extending QPhiX for different formulations of LQCD, and integrating the library into established code bases, it would be worthwhile to implement newer algorithmic ideas, such as domain decomposition preconditioners, deflation and algebraic multi-grid approaches. In particular, domain decomposition could allow to block the small domains in which the lattice is decomposed into the low lying caches, in order to shift the memory bandwidth bottleneck from the main memory to the much faster caches. There has already been work in this direction for Xeon Phi's in [38].

Bibliography

[1] Mimd lattice computation (milc) collaboration, http://physics.indiana.edu/ sg/milc.html.

[2] http://jemalloc.net/, 2016.

[3] https://github.com/01org/hetero-streams, 2016.

[4] https://github.com/memkind/memkind, 2016.

[5] https://pm.bsc.es/content/hstreams-doc, 2016.

[6] https://software.intel.com/sites/landingpage/intrinsicsguide/, 2016.

[7] https://www.threadingbuildingblocks.org/, 2016.

[8] http://www.deep-er.eu/, 2016.

[9] http://www.electronicdesign.com/dsps/intels-avx-scales-1024-bit-vector-math, 2016.

[10] http://www.itworld.com/article/2985214/hardware/intels-xeon-roadmap-for-2016-leaks.html, 2016.

[11] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3814.html, 2016.

[12] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf, 2016.

[13] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0159r0.html, 2016.

[14] B. Alder and T. Wainwright. Phase transition for a hard sphere system. *The Journal of chemical physics*, 27(5):1208, 1957.

[15] F. Bodin et al. APE computers—past, present and future. *Comput. Phys. Commun.*, 147(1-2):402–409, 2002.

[16] P. A. Boyle. The bagel assembler generation library. *Computer Physics Communications*, 180(12):2739–2748, 2009.

[17] P. A. Boyle. The BlueGene/Q supercomputer. *PoS*, LATTICE2012:020, 2012.

[18] P. A. Boyle, G. Cossu, A. Yamaguchi, and A. Portelli. Grid: A next generation data parallel C++ QCD library. *PoS*, LATTICE2015:023, 2016.

[19] B. Bunk and R. Sommer. An Eight Parameter Representation of SU(3) Matrices and Its Application for Simulating Lattice QCD. *Comput. Phys. Commun.*, 40:229–232, 1986.

[20] N. Cabibbo. APE: A High Performance Processor for Lattice QCD. In *Old and New Problems in Fundamental Physics: Meeting in Honour of G.C. Wick*, pages 137–144, 1984.

[21] D. J. E. Callaway and A. Rahman. Microcanonical ensemble formulation of lattice gauge theory. *Phys. Rev. Lett.*, 49:613–616, Aug 1982.

[22] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Comput. Phys. Commun.*, 181:1517–1528, 2010.

[23] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM, 1987.

[24] M. Creutz. Global monte carlo algorithms for many-fermion systems. *Physical Review D*, 38(4):1228, 1988.

[25] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[26] P. de Forcrand, D. Lellouch, and C. Roiesnel. Optimizing a lattice qcd simulation program. *Journal of Computational Physics*, 59(2):324 – 330, 1985.

[27] T. DeGrand and C. DeTar. *Lattice methods for quantum chromodynamics.* World Scientific, 2006.

[28] T. A. DeGrand and P. Rossi. Conditioning Techniques for Dynamical Fermions. *Comput. Phys. Commun.*, 60:211–214, 1990.

[29] R. G. Edwards and B. Joo. The chroma software system for lattice qcd. *arXiv preprint hep-lat/0409003*, 2004.

[30] R. Fletcher. Conjugate gradient methods for indefinite systems. In *Numerical analysis*, pages 73–89. Springer, 1976.

[31] E. Forest and R. D. Ruth. Fourth-order symplectic integration. *Physica D: Nonlinear Phenomena*, 43(1):105 – 117, 1990.

[32] R. Frezzotti, P. A. Grassi, S. Sint, and P. Weisz. A Local formulation of lattice QCD without unphysical fermion zero modes. *Nucl. Phys. Proc. Suppl.*, 83:941–946, 2000.

[33] R. Frezzotti, P. A. Grassi, S. Sint, and P. Weisz. Lattice QCD with a chirally twisted mass term. *JHEP*, 08:058, 2001.

[34] R. Frezzotti, S. Sint, and P. Weisz. O(a) improved twisted mass lattice QCD. *JHEP*, 07:048, 2001.

[35] C. Gattringer and C. Lang. *Quantum chromodynamics on the lattice: an introductory presentation*, volume 788. Springer Science & Business Media, 2009.

[36] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, et al. The ibm blue gene/q compute chip. *Ieee Micro*, 32(2):48–60, 2012.

[37] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. 49, 1952.

[38] S. Heybrock, B. Joó, D. D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig, and P. Dubey. Lattice QCD with Domain Decomposition on Intel Xeon Phi Co-Processors. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis: SC14: HPC matters (SC) New Orleans, LA, USA, November 16-21, 2014*, 2014.

[39] Y.-C. Jang, H.-J. Kim, and W. Lee. Multi GPU Performance of Conjugate Gradient Solver with Staggered Fermions in Mixed Precision. *PoS*, LATTICE2011:309, 2011.

[40] J. Jeffers and J. Reinders. *High Performance Parallelism Pearls Two Volumes: Multicore and Many-core Programming Approaches.* Morgan Kaufmann, 2015.

[41] B. Jegerlehner. Improvements of Luscher's local bosonic fermion algorithm. *Nucl. Phys.*, B465:487–506, 1996.

[42] B. Joó. Scidac-2 software infrastructure for lattice qcd. In *Journal of Physics: Conference Series*, volume 78, page 012034. IOP Publishing, 2007.

[43] B. Joó, D. D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. W. Lee, P. Dubey, and W. A. Watson III. Lattice qcd on intel® xeon phitm coprocessors. In *ISC*, pages 40–54, 2013.

[44] B. Joo, M. Smelyanskiy, D. D. Kalamkar, and K. Vaidyanathan. *Wilson Dslash Kernel From Lattice QCD Optimization.* Jul 2015.

[45] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[46] M. Luscher and P. Weisz. On-Shell Improved Lattice Gauge Theories. *Commun. Math. Phys.*, 97:59, 1985. [Erratum: Commun. Math. Phys.98,433(1985)].

[47] P. B. Mackenze. An improved hybrid monte carlo method. *Physics Letters B*, 226(3):369 – 371, 1989.

[48] L. Markus and K. R. Meyer. *Generic Hamiltonian dynamical systems are neither integrable nor ergodic*, volume 144. American Mathematical Soc., 1974.

[49] R. Martin, G. Peters, and J. Wilkinson. Iterative refinement of the solution of a positive definite system of equations. *Numerische Mathematik*, 8(3):203–216, 1966.

[50] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6), 1953.

[51] R. M. Neal et al. Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2:113–162, 2011.

[52] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE Computer Society, 2010.

[53] M. Peskin and D. Schroeder. *An introduction to quantum field theory.* 1995.

[54] Pochinksy, A. V. Qa0 code generator.

[55] J. Reinders, J. Jeffers, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition.* Boston, MA, USA: Morgan Kaufmann Publishers Inc, 2016.

[56] L. F. Richardson. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 210:307–357, 1911.

[57] H. J. Rothe. *Lattice gauge theories: an introduction*, volume 74. World Scientific, 2005.

[58] R. D. Ruth. A Canonical Integration Technique. *IEEE Transactions on Nuclear Science*, 30:2669, Aug. 1983.

[59] Y. Saad. *Iterative methods for sparse linear systems.* Siam, 2003.

[60] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[61] S. Schaefer. Status and challenges of simulations with dynamical fermions. *PoS*, LATTICE2012:001, 2012.

[62] A. Schäfer and D. Fey. High performance stencil code algorithms for gpgpus. *Procedia Computer Science*, 4:2027–2036, 2011.

[63] M. Schröck, S. Simula, and A. Strelchenko. Accelerating Twisted Mass LQCD with QPhiX. *PoS*, LATTICE2015:030, 2016.

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

[65] A. Shindler. Twisted mass lattice QCD. *Phys. Rept.*, 461:37–110, 2008.

[66] G. L. Sleijpen and H. A. van der Vorst. Reliable updated residuals in hybrid bi-cg methods. *Computing*, 56(2):141–163, 1996.

[67] G. L. G. Sleijpen and H. A. van der Vorst. Reliable updated residuals in hybrid bi-cg methods. *Computing*, 56(2):141–163, 1996.

[68] M. Smelyanskiy, K. Vaidyanathan, J. Choi, B. Joó, J. Chhugani, M. A. Clark, and P. Dubey. High-performance lattice qcd for multi-core based parallel systems using a cache-friendly hybrid threaded-mpi approach. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. ACM, 2011.

[69] P. Sonneveld. Cgs, a fast lanczos-type solver for nonsymmetric linear systems. *SIAM journal on scientific and statistical computing*, 10(1):36–52, 1989.

[70] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[71] Strohmaier, Dongarra, Simon, Meuer. https://www.top500.org/, 2016.

[72] R. Strzodka. Pipelined mixed precision algorithms on fpgas for fast and accurate pde solvers from low precision components. In *In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 259–268, 2006.

[73] K. Symanzik. Continuum Limit and Improved Action in Lattice Theories. 1. Principles and phi**4 Theory. *Nucl. Phys.*, B226:187–204, 1983.

[74] K. Symanzik. Continuum Limit and Improved Action in Lattice Theories. 2. O(N) Nonlinear Sigma Model in Perturbation Theory. *Nucl. Phys.*, B226:205–227, 1983.

[75] L. N. Trefethen and D. Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[76] A. Ukawa. Computational cost of full QCD simulations experienced by CP-PACS and JLQCD Collaborations. *Nucl. Phys. Proc. Suppl.*, 106:195–196, 2002. [,195(2002)].

[77] H. A. Van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

[78] D. Vandevoorde and N. M. Josuttis. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[79] S. Weinberg. *The quantum theory of fields.* Cambridge university press, 1996.

[80] K. G. Wilson. Confinement of quarks. *Physical Review D*, 10(8):2445, 1974.