Master Thesis

Performance Modelling and Analysis of the openQ*D Lattice QCD Application

Roman Gruber

ETH Zürich, June 11, 2021

Supervisor:
Prof. Thomas C. Schulthess

Co-Supervisors:
Prof. Marina Krstić Marinković
Dr. Raffaele Solcà
Dr. Anton Kozhevnikov

Abstract

The open-source software openQ*D is used to perform lattice QCD+QED calculations using $O(a)$-improved Wilson-fermions and a variety of boundary conditions (including C*). The focus of this thesis lies on the optimization of solver kernels provided with the software package intending to give guidance for porting fractions of the code to GPU-accelerated supercomputing platforms. We present an analysis of the current sparse solver implementations, propose suitable extensions to the existing algorithms and possible CPU/GPU-hybrid implementations of them. Various application-specific proposals for optimizations are suggested as well. The most important kernel in every lattice QFT application – the Dirac-operator – is investigated and a reference implementation using threads is shown which can serve as a starting point for an implementation on the GPU. Considering memory-boundedness of the problem, we present different numerical representations of the Dirac-operator with higher arithmetic intensities in their application and less memory-traffic suitable for the GPU.

Contents

1 Introduction

Standard Model particle physics is one of the most successful theories in terms of agreement between theoretical predictions and experimental evidence. Being able to accurately predict observables such as decay constants using computational resources opens the possibility to probe and understand the Standard Model, emulate experiments or even discover new physics. In the regime of high energy, non-abelian theories such as QCD can be treated perturbatively, because the coupling constant is small with large momentum transfers. In the low-energy regime on the other hand where perturbation theory becomes unfeasible, lattice QCD provides a non-perturbative tool to analyse the hadronic dynamics from first principles.

Translating the formalism of QFT to a discretised finite system such as a computer system involves many technical challenges and comes at a high cost, mainly due to extremely large demands on compute capability and memory. The main computational cost of lattice QFT calculations originates from solving the discrete Dirac equation, which is a large sparse system of linear equations. The most used kernel in dynamical QCD simulations is the application of the Dirac-operator to a spinor field. Lattice simulations allow to calculate correlation functions of hadronic operators, predict the running coupling constant, quark masses, hadronic spectra or elements of the CKM matrix. Due to the statistical approach in the calculations, results include systematic errors coming from the lattice description as well as statistical errors.

(Super)computing systems tend to include more and more units called domain-specific accelerators [1] designed to be specialised on distinct computational tasks. The general-purpose compute-device such as the CPU gets less attention, whereas the specialised hardware device becomes more important [2]. A very prominent example of such a device is the GPU that was originally designed to rapidly render geometric 3D objects into 2D images, thus the name: graphics processing unit.[1] Inherently such a process is highly parallel, where at the same time precision of the result is less important, because if the output image contains a pixel that has a different color, the overall error is barely noticed. The scientific world started to misuse these devices and executed parallel non-graphics algorithms on it [10]. Specially the rise of machine learning in the last 10-20 years kicked-off a trend [11], because of which GPU vendors started to develop general-purpose GPU accelerators specially designed for the need of these new scientific calculations.[2] The GPU has been used for general-purpose computation (GPGPU) [13]. A consequence was the advent of HPC clusters based on the GPU as dominant architecture [14].

On the other side (or because of the rise of GPUs) modern supercomputers tend towards exascale computing, where the peak operations per second is around 10^{18} [15]. To reach such a peak performance is challenging and highly depends on the problem as well as the involved data types. Since lattice QFT calculations are bound by memory bandwidth and not by compute performance, one must think about how to reduce memory traffic in order to increase performance. The parallelizability of GPUs thus opened the door to exascale computing but made it harder to achieve its peak performance for memory-bound problems, because on GPUs the memory-bound region is even larger than on regular CPUs [16].

The scope of this work is an analysis of the different solver algorithms used in the lattice QFT application openQ*D [17] as well as the implementation of the Dirac-operator therein. The goal was to get an idea and understand how to utilise the GPU – and by this – increase the overall application performance. The following document is by no means complete or exhaustive. Different aspects of the solvers are highlighted to find potential for improvement, where this is mainly guided and limited by my own understanding of said solvers and how they apply on the considered computing architectures.

The analysis in this document is mostly performed using Python-implementations of the examined kernels. This switch of programming language and philosophy enabled to run the kernels with simulated data types that are usually non-accessible within the native application without large implementation effort. The goal was to rethink the current pure-MPI variant and propose a hybrid code that exploits both, the CPU and the GPU.

Section 3 is a short theoretical introduction in non-abelian gauge theories starting with symme-

[1]Other notable examples where dedicated devices are used include deep learning [3, 4], bioinformatics [5], image processing [6], computer vision [7, 8], Bitcoin mining [9] and many more.

[2]The emergence of cryptocurrencies, such as Bitcoin [12], making use of the blockchain are also partly responsible for that trend.

tries coming from special relativity and basic quantum mechanics. The final result will be Euclidean continuum Yang-Mills-theory.

Section 4 directly picks up on the previous results and continues to derive discrete quantities from continuum theory by introducing the lattice discretisation which acts as a regulator for the inherent infinities appearing in the continuum theory.[3] The goal of this section is the introduction of the lattice Dirac-operator and its importance which motivates the main part of this work.

The main subject of this thesis is the software package openQ*D, which is briefly presented with its main features in section 5.

Section 6 consists of a theoretical discussion of discrete real number formats, their pros and cons as well as their current and potential future appearances in the source code of openQ*D.

With these solid theoretical foundations, the first algorithm Conjugate Gradient is investigated in terms of a convergence analysis with respect to different real number formats in section 7.

Section 8 deals with the Generalized Conjugate Residual with Schwarz Alternating Preconditioning algorithm and shows actual reference implementations, first purely based on CPU, then purely based on GPU and finally a hybrid version running on both. These three implementations are compared against each other and discussed, also with different real number formats. Finally, motivated by the results, an adaptive variant of the solver algorithm is proposed and compared to the native ones. A reference implementation of the adaptive version was directly written in openQ*D.

In section 9 deflation of the original system of linear equations is applied resulting in an efficient and powerful solver for ill-conditioned Dirac-operators, the Deflated Generalized Conjugate Residual with Schwarz Alternating Preconditioning algorithm. A formal proof of efficiency compared to the non-deflated variant is provided. This holds if the Dirac-operator features certain properties such as local coherence and a clustered low-eigenvalue spectrum. These properties are precisely determined in the first part of this section. The main result of the numerical experiments is the determination of the deflation subspace dimension.

The last section 10 investigates the current implementation of the Dirac-operator in openQ*D and gives proposals to increase its arithmetic intensity and decrease memory-traffic by introducing compressed representations of the gauge-fields. Finally, a reference implementation of the Dirac-kernel using threads was implemented and compared to the native implementation in terms of run-time.

A summary and possible continuations of this work are given in sections 12 and 13, respectively.

2 Conventions

Units. We shall work with natural units $\hbar = c = 1$.

Minkowski Space. We use four-vectors with Greek indices, $\mu, \nu, \dots = 0, 1, 2, 3$:

- contravariant position vector $x^\mu := (x^0, x^1, x^2, x^3) = (t, \vec{x})$.
- covariant position vector $\eta_{\mu\nu} x^\nu = x_\mu := (x_0, x_1, x_2, x_3) = (t, -\vec{x})$.
- where $\vec{x} := (x^1, x^2, x^3)$. For the spatial part Latin indices are used $i, j, k, l = 1, 2, 3$.

The Minkowski metric η has signature $(+---)$. The scalar product between two four-vectors

$$\langle x, y \rangle = x^\mu y_\mu := \sum_{\mu=0}^{D} x^\mu y_\mu,$$

where D denotes the space-time dimension and repeated indices are summed over (Einstein summation convention).

Euclidean Space. We use four-vectors with Greek indices, $\mu, \nu, \dots = 1, 2, 3, 4$:

- position vector $x^\mu := (x^1, x^2, x^3, x^4) = (\vec{x}, \tau)$.

[3]The continuum theory assumes infinitely extended fields (IR) with infinite spatial resolution (UV); as a result, the theory produces infinities. As direct implication, it is very intuitive that the finite lattice (IR) with finite spacing (UV) cures these two main sources of infinities.

- position vector $x_\mu := (x_1, x_2, x_3, x_4) = (\vec{x}, \tau)$.

The Euclidean metric η has signature $(++++)$, and we have $x_\mu = x^\mu$.

Spinor indices. Spinor indices are denoted with Greek letters starting from $\alpha, \beta, \dots$.

Lorentz indices. Lorentz indices are denoted with Greek letters starting from $\mu, \nu, \rho, \sigma, \dots$.

Color indices. Color indices are denoted with Latin letters starting from $a, b, c, \dots$.

Norm. The norm is defined to be the one induced by the scalar product, $\|x\| := \sqrt{\langle x, x\rangle}$.

Vectors. Vectors may be written as Latin letters with an arrow $\vec{v}, \vec{w}, \dots$ or Greek letters without an arrow $\psi, \phi, \dots$. The meaning should be clear from context.

In general we may distinguish between the symbols $(\cdot)^\dagger$, $(\cdot)^\star$ and $\overline{(\cdot)}$:

Dirac-adjoint. The overbar shall only be used for the Dirac-adjoint $\overline{\psi} := \psi i \gamma^0$.

Operators. Let A be an operator, $A^\dagger := (A^\star)^T$ is the Hermitian adjoint, where T denotes the transpose and the asterisk means conjugation of a complex number.

Vectors. Let $\vec{v} \in \mathbb{C}^n$ be a vector, $\vec{v}^\dagger := (\vec{v}^\star)^T = (v_1^\star \dots v_n^\star)$, where v_i are the components of the vector, such that the scalar product can be written as a matrix-matrix product resulting in a 1×1-matrix (a number), $\vec{v}^\dagger \vec{w}$.

3 Non-abelian gauge theories

The goal of this section is to derive a Lagrangian describing fundamental fermions that is Lorentz-invariant (coming from special relativity) as well as invariant under local phase-transformations (coming from basic quantum mechanics).

Let's consider a set of N complex independent Dirac spinors

$$\psi(x) = \begin{pmatrix} \psi_1(x) \\ \vdots \\ \psi_N(x) \end{pmatrix},$$

where every $\psi_a(x)$, $a \in \{1, \dots N\}$, has 4 components with spinor indices $\alpha \in \{1, \dots 4\}$. Thus, we have a set of complex Grassmann-valued fields $\psi_a^\alpha(x)$, where a is called the ***color index*** and α is the ***spinor index***. It makes sense to demand the theory to be invariant under $SU(N)$-transformations, because basic quantum mechanics tells us that phases are unobservable. Such a field theory that is invariant under the gauge group $SU(N)$ has to introduce (massless) vectorial fields $A_\mu(x)$, called the ***gauge fields***. From special relativity, we also demand the theory to be Lorentz-invariant. The goal is to construct a Lagrangian for fermionic fields that satisfies these symmetries.

Let's introduce the local[4] gauge transformation under which we want the theory to be invariant

$$\psi(x) \longrightarrow \tilde{\psi}(x) = V(x)\psi(x),$$

with $V(x) \in SU(N)$, thus it has indices in color space, explicitly $V(x)_{ab}$. Since we want our fermions to possess a non-zero mass, the Lagrangian will contain a term quadratic in the field ψ. A first approach of an $SU(N)$-invariant expression would be a term proportional to $\psi^\dagger\psi$. Unfortunately, ψ is Grassmann valued[5], thus anti-commutes with itself, meaning the mentioned term is equal to zero.[6] Special relativity also demands the term to be a Lorentz-scalar, which $\psi^\dagger\psi$ is not. Under Lorentz-transformation Λ, ψ and $\psi(x)^\dagger$ transform as

$$\psi(x) \longrightarrow \Lambda\psi(x),$$

[4] The phase depends on the space-time coordinate x.

[5] For a formal introduction to Grassmann variables see appendix B.

[6] To be pedantic, 0 *is* $SU(N)$- and Lorentz-invariant.

$$\psi(x)^\dagger \longrightarrow \psi(x)^\dagger \Lambda^\dagger.$$

We can use a property of the γ-matrices here. The γ-matrices are defined such that they obey the ***Clifford-algebra***[7]

$$\{\gamma^\mu, \gamma^\nu\} = 2\eta^{\mu\nu} \cdot id, \tag{3.1}$$

with $\mu, \nu \in \{0, 1, \ldots, D-1\}$, where D is the space-time dimension and id is the identity operator in spinor space. The needed property of the γ-matrices is

$$\Lambda^\dagger \gamma^\mu = \gamma^\mu \Lambda^{-1}.$$

With this, it makes sense to define the ***Dirac-adjoint*** as $\overline{\psi} := \psi i \gamma^0$ and construct a Lorentz- and $SU(N)$-invariant quadratic expression $c\overline{\psi}\psi$, with $c \in \mathbb{C}$.

The Lagrangian of the theory inevitably contains derivatives of ψ and since the transformation is local (different for every space-time point x), we must redefine a derivative that compensates for this. The regular ***directional derivative*** along direction n is

$$n^\mu \partial_\mu \psi(x) = \lim_{\epsilon \to 0} \frac{\psi(x + n\epsilon) - \psi(x)}{\epsilon}. \tag{3.2}$$

The two fields appearing in this expression are evaluated at different space-time points and thus transform differently under $V(x)$. For the kinetic expression in the Lagrangian $n^\mu \partial_\mu$ to be invariant under $SU(N)$, we introduce a compensator for the shifts in the derivative.

Definition 3.1 (Compensator field)**.** *The* ***compensator field*** *$U(x, y)$ is a non-local matrix quantity that transforms under $V(x) \in SU(N)$ as*

$$U(x, y) \longrightarrow \tilde{U}(x, y) = V(x) U(x, y) V^\dagger(y). \tag{3.3}$$

$U(x, y)$ is an element of $SU(N)$ for all x, y, with $U(x, x) = id$.

We then redefine the derivative as

$$n^\mu D_\mu \psi(x) = \lim_{\epsilon \to 0} \frac{U(x, x + \epsilon n)\psi(x + \epsilon n) - \psi(x)}{\epsilon} \tag{3.4}$$

to compensate for the shift in x. Now, the derivative term transforms as desired,

$$n^\mu D_\mu \psi(x) \longrightarrow n^\mu \tilde{D}_\mu \tilde{\psi}(x) = n^\mu V(x) D_\mu \psi(x).$$

Since the compensator field is non-local and thus not representing a real physical particle, we can obtain a local quantity by Taylor-expanding $U(x + n\epsilon, x)$ for small distances ϵ around $\epsilon = 0$ and find

$$\begin{aligned} U(x, x + \epsilon n) &\approx U(x, x) + \frac{1}{1!} \frac{\partial U}{\partial x^\mu} \frac{\partial (x^\mu + \epsilon n^\mu)}{\partial \epsilon} \bigg|_{\epsilon = 0} \cdot \epsilon + O(\epsilon^2) \\ &= id + \frac{\partial U}{\partial x^\mu} \bigg|_{\epsilon = 0} \epsilon n^\mu + O(\epsilon^2) \\ &= id + ig\epsilon n^\mu A^a_\mu(x) T^a + O(\epsilon^2), \end{aligned} \tag{3.5}$$

[7] We use the particle physics convention of the metric tensor $\eta_{\mu\nu}$ with signature $(+, -, \ldots, -)$. In $D = 4$ dimensions, $\eta_{\mu\nu} = \mathrm{diag}(+1, -1, -1, -1)_{\mu\nu}$

where we introduced a local real-valued bosonic vector-fields $A^a_\mu(x)$[8], an arbitrary constant $g \in \mathbb{R}_{>0}$ and the generators $T^a \in su(N)$, the Lie-algebra of $SU(N)$. The fields $A^a_\mu(x)$ are defined by

$$\left.\frac{\partial U}{\partial x^\mu}\right|_{\epsilon=0} =: igA^a_\mu(x)T^a.$$

Since $C_{x,n}(t) := U(x, x+tn)$ for every given x and n is a curve in $SU(N)$ that satisfies $C_{x,n}(t = 0) = id$, its derivative evaluated at $t = 0$ is therefore an element of the Lie-algebra $su(N)$[9]. Inserting this into (3.4) gives

$$\begin{aligned} n^\mu D_\mu \psi(x) &= \lim_{\epsilon\to 0} \frac{\psi(x+\epsilon n) + ig\epsilon n^\mu A^a_\mu(x)T^a\psi(x+\epsilon n) - \psi(x)}{\epsilon} \\ &= \lim_{\epsilon\to 0} \frac{\psi(x+n\epsilon) - \psi(x)}{\epsilon} + ign^\mu A^a_\mu(x)T^a\psi(x) \\ &= n^\mu\left(\partial_\mu + igA^a_\mu(x)T^a\right)\psi(x), \end{aligned}$$

which leads directly to the definition of the ***gauge covariant derivative***

$$D_\mu := \partial_\mu + igA^a_\mu(x)T^a. \tag{3.6}$$

We introduced a new field – and by this a corresponding particle. In order for the particle to have a propagator, we also have to implement a kinetic term that is quadratic in A^a_μ or its derivatives. Obviously, the kinetic term should be invariant under $SU(N)$ transformations as well. For this we need the plaquette[10].

Definition 3.2 (Plaquette)**.** *Let $n_1 \neq n_2$ be two 4-vectors and $\epsilon > 0$. The* ***plaquette*** *(see figure 1) in the (n_1, n_2)-subspace is defined as*

$$\hat{U}_{n_1,n_2}(\epsilon, x) := U(x, x+\epsilon n_2)U(x+\epsilon n_2, x+\epsilon n_2+\epsilon n_1)U(x+\epsilon n_2+\epsilon n_1, x+\epsilon n_1)U(x+\epsilon n_1, x). \tag{3.7}$$

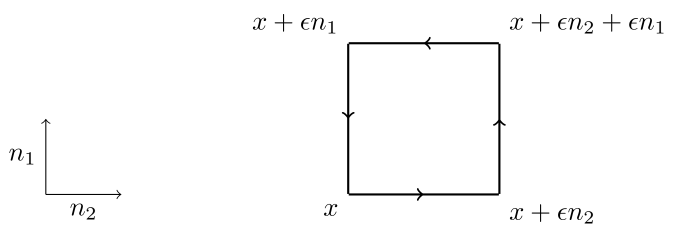

Figure 1: Scheme of the plaquette $\hat{U}_{n_1,n_2}(\epsilon, x)$ in the (n_1, n_2)-subspace.

The plaquette $\hat{U}_{n_1,n_2}(\epsilon, x)$ is not invariant under $SU(N)$, but its trace over color space[11] is, $tr_C(\hat{U}_{n_1,n_2}(\epsilon, x))$, because of the cyclicity of the trace. In the continuum theory the compensator field $U(x, y)$ is understood as the parallel transport of a curve connecting x and y.

[8] $A^a_\mu(x)$ is an auxiliary field that is the infinitesimal limit of a compensator field, also called a ***connection*** on a fibre-bundle.

[9] The Lie-algebra element is $igA^a_\mu(x)T^a$, where $igA^a_\mu(x)$ are the coefficients of the generators T^a. The sum over a is implicit.

[10] There are other (probably simpler and more elegant) methods to derive a kinetic term for the gauge fields, but the plaquette will arise again later in context of lattice gauge theories. And as a physicist sometimes one has to go though some pain and suffer a little bit here and there – this is part of the game.

[11] With trace over color space, we mean the partial trace over the color part in a tensor product of spinor- (S), color- (C) and Lorentz- (L) or other spaces. The trace function may then be decorated with the space if it's not clear from context, for example $tr_C(U)$.

Theorem 3.1 (The compensator field $U(x,y)$ as parallel transport). *The unique $SU(N)$-valued object satisfying the defining transformation law* (3.3) *is*

$$U(x,y) = P\exp\left(ig\int_y^x dz^\mu A_\mu^a(z)T^a\right), \tag{3.8}$$

where P is the ***path-ordering*** *operator.*

Equation (3.8) can be interpreted that a fermion moving from x to y picks up a (path-ordered) phase given by A_μ^a integrated over that path, thus $U(x,y)$ "links"[12] space-time points x and y.

Proof. We have to show that (3.8) transforms as (3.3). First, we write the curve starting from y as $\gamma(s)$, with $\gamma(0)=y$, then the role of the variable endpoint x is taken by $\gamma(s)$,

$$U(\gamma(s),y) = P\exp\left(ig\int_y^{\gamma(s)} dz^\mu A_\mu^a(z)T^a\right).$$

$U(\gamma(s),y)$ satisfies the differential equation

$$\begin{aligned}\frac{d}{ds}U(\gamma(s),y) &= U(\gamma(s),y)\left\{ig\frac{d}{ds}\int_y^{\gamma(s)} dz^\mu A_\mu^a(z)T^a\right\}\\ &= U(\gamma(s),y)\left\{ig\frac{d\gamma^\mu}{ds}A_\mu^a(\gamma(s))T^a\right\},\end{aligned}$$

or equivalently

$$\frac{dx^\mu}{ds}D_\mu U(x,y) = 0. \tag{3.9}$$

Since the solution to a first order differential equation with a fixed boundary condition must be unique, equation (3.8) is therefore the unique solution to the above differential equation. A defining property of the covariant derivative was, that when applied to a fermion field, it transforms as

$$\begin{aligned}D_\mu\psi(x) &\longrightarrow \tilde{D}_\mu\tilde{\psi}(x) = V(x)D_\mu\psi(x),\\ D_\mu\psi(x) &\longrightarrow \tilde{D}_\mu\tilde{\psi}(x) = \tilde{D}_\mu V(x)\psi(x),\end{aligned}$$

meaning that the *yellow* parts must be equal (as operators acting to the right applied to *any* field) and $\tilde{D}_\mu$ is written in terms of the transformed fields $\tilde{A}_\mu^a$. We have not derived a transformation rule for $A_\mu^a(x)$. We also don't have to because, if $U(x,y)$ is the solution to (3.9), then

$$\begin{aligned}\frac{dx^\mu}{ds}\left[\tilde{D}_\mu V(x)U(x,y)V^\dagger(y)\right] &= \frac{dx^\mu}{ds}\left[\tilde{D}_\mu V(x)U(x,y)\right]V^\dagger(y)\\ &= V(x)\left[\frac{dx^\mu}{ds}D_\mu U(x,y)\right]V^\dagger(y)\\ &= 0,\end{aligned}$$

using $V(x)D_\mu = \tilde{D}_\mu V(x)$ in the second line. Therefore $V(x)U(x,y)V^\dagger(y)$ is the (unique) solution to the transformed differential equation and thus must be the transformed version of $U(x,y)$ under $SU(N)$.

□

[12] In context of lattice gauge theory the discrete version of the parallel transport is thus called ***link variable***, since it "links" lattice points.

If x and y are not far apart – say $y = x + \epsilon n$, for an $\epsilon > 0$ small – the integrals within the path-ordered exponential can be approximated as the difference between the two space-time points in every direction μ multiplied by the value of $A^a_\mu(x)$ evaluated between the two points[13],

$$U(x + \epsilon n_1, x + \epsilon n_2) = \exp\Big(-ig\epsilon(n_1 - n_2)^\mu A^a_\mu(x + \epsilon\frac{n_1 + n_2}{2})T^a + O(\epsilon^3)\Big). \tag{3.10}$$

If ϵ is small, we can expand

$$A^a_\mu(x + \epsilon n) = A^a_\mu(x) + \epsilon n^\nu \partial_\nu A^a_\mu(x) + O(\epsilon^2)$$

and insert it in equation (3.10),

$$\begin{aligned}&U(x + \epsilon n_1, x + \epsilon n_2) = \\ &\qquad \exp\Big(-ig\epsilon(n_1 - n_2)^\mu A^a_\mu(x)T^a - ig\frac{\epsilon^2}{2}(n_1 + n_2)^\nu \partial_\nu (n_1 - n_2)^\mu A^a_\mu(x)T^a + O(\epsilon^3)\Big).\end{aligned}$$

Using this formula as well as the Baker–Campbell–Hausdorff formula[14] we obtain

$$\begin{aligned}\hat{U}_{n_1,n_2}(\epsilon, x) = \exp\Bigg(&+ ig\epsilon n_2^\mu A^a_\mu(x)T^a + ig\frac{\epsilon^2}{2} n_2^\nu \partial_\nu n_2^\mu A^a_\mu(x)T^a \\ &+ ig\epsilon n_1^\mu A^a_\mu(x)T^a + ig\frac{\epsilon^2}{2}(n_1 + 2n_2)^\nu \partial_\nu n_1^\mu A^a_\mu(x)T^a \\ &- ig\epsilon n_2^\mu A^a_\mu(x)T^a - ig\frac{\epsilon^2}{2}(2n_1 + n_2)^\nu \partial_\nu n_2^\mu A^a_\mu(x)T^a \\ &- ig\epsilon n_1^\mu A^a_\mu(x)T^a - ig\frac{\epsilon^2}{2} n_1^\nu \partial_\nu n_1^\mu A^a_\mu(x)T^a \\ &- \frac{1}{2}g^2\epsilon^2 n_2^\mu A^a_\mu(x) n_1^\nu A^b_\nu(x)[T^a, T^b] + \frac{1}{2}g^2\epsilon^2 n_2^\mu A^a_\mu(x) n_1^\nu A^b_\nu(x)[T^a, T^b] \\ &+\frac{1}{2}g^2\epsilon^2 n_1^\mu A^a_\mu(x) n_2^\nu A^b_\nu(x)[T^a, T^b] - \frac{1}{2}g^2\epsilon^2 n_2^\mu A^a_\mu(x) n_1^\nu A^b_\nu(x)[T^a, T^b] + O(\epsilon^3)\Bigg)\end{aligned}$$

By staring long enough at this expression, we see that there are a lot of terms cancelling each other; the first terms in the first 4 lines cancel exactly, the second terms in the first 4 lines all cancel except the terms involving $2n_1$ and $2n_2$, the two terms in the fifth line cancel, and the two terms in the sixth line are the same. Thus, we have 3 terms surviving which are highlighted in yellow color. Using the directed fields and derivatives $n_i^\mu A^a_\mu(x) =: A^a_i(x)$ and $n_j^\nu \partial_\nu n_i^\mu A^a_\mu(x) =: \partial_j A^a_i(x)$, we end up with

$$\hat{U}_{n_1,n_2}(\epsilon, x) = \exp\Bigg(-ig\epsilon^2 \underbrace{[\partial_1 A^a_2(x) - \partial_2 A^a_1(x) + gf^{abc}A^b_1(x)A^c_2(x)]}_{=:F^a_{12}} T^a + O(\epsilon^3)\Bigg), \tag{3.11}$$

[13]This is in principle the midpoint rule within a Riemann sum that has only one summand. The approximation is good up to $O(\epsilon^3)$.

[14]In this case we used the BCH-formula on steroids,

$$e^{\epsilon A}e^{\epsilon B}e^{\epsilon C}e^{\epsilon D} = e^{\epsilon A+\epsilon B+\epsilon C+\epsilon D+\frac{\epsilon^2}{2}[A,B]+\frac{\epsilon^2}{2}[A,C]+\frac{\epsilon^2}{2}[A,D]+\frac{\epsilon^2}{2}[B,C]+\frac{\epsilon^2}{2}[B,D]+\frac{\epsilon^2}{2}[C,D]+O(\epsilon^3)}.$$

where we used the totally anti-symmetric ***structure constants*** of the Lie-algebra defined over the commutation relations $[T^a, T^b] = if^{abc}T^c$.

The expression in the square brackets of the last line is what we define as the ***Yang-Mills field strength tensor***[15]

$$F^a_{\mu\nu} = \partial_\mu A^a_\nu(x) - \partial_\nu A^a_\mu + gf^{abc}A^b_\mu(x)A^c_\nu(x). \tag{3.12}$$

The field strength tensor transforms in the same way as the plaquette $\hat{U}_{n_1,n_2}(\epsilon, x)$, thus the trace over $F^a_{\mu\nu}T^a$ in color space is invariant under $SU(N)$ transformations. To be Lorentz-invariant as well, we need a Lorentz-scalar. Thus we need to contract the indices and obtain[16]

$$tr(F^a_{\mu\nu}T^aF^{\mu\nu,b}T^b) = F^a_{\mu\nu}F^{\mu\nu,b}tr(T^aT^b) \tag{3.13}$$
$$= \frac{1}{2}F^a_{\mu\nu}F^{\mu\nu,a}. \tag{3.14}$$

The quadratic terms in the Lagrangian are always of the form $-\frac{1}{2}(\mathit{field})^2$. To honour this convention, we finally end up in a Lorentz- and $SU(N)$-invariant Lagrangian including the fermionic part, the ***Yang-Mills Lagrangian*** [18], of the form

$$\mathcal{L} = -\frac{1}{4}F^a_{\mu\nu}F^{\mu\nu,a} + \bar{\psi}\left(i\not{D} - m\right)\psi, \tag{3.15}$$

where $\not{D} = \gamma^\mu D_\mu$ is the ***Feynman slash notation***. Just as in the abelian theory, there is no term quadratic in A, $m_A A^a_\mu A^{\mu,a}$, a mass-term for the gauge fields, because this would violate the $SU(N)$-invariance. The fundamental vector particles which are represented by these fields must therefore be massless $m_A = 0$. The constant g can be interpreted as ***coupling constant***. The terms in the above Lagrangian are the only ones that can appear if we demand the space-time to have $D = 4$ dimensions, parity- (P) and time-reversal- (T) invariance. The gauge part of the Lagrangian employs interactions among the gauge fields, namely the theory is equipped with pure 3- and 4-vertices of gauge bosons.

3.1 Euclidean theory

Starting from the continuum Yang-Mills Lagrangian in Minkowski D-dimensional space-time (the superscript M stands for Minkowski, see equation (3.15))

$$\mathcal{L}^M_{YM} = \mathcal{L}^M_G + \mathcal{L}^M_F, \tag{3.16}$$

with fermion- (F) and gauge-part (G)

$$\begin{aligned}\mathcal{L}^M_F &= \bar{\psi}\left(i\not{D} - m\right)\psi,\\ \mathcal{L}^M_G &= -\frac{1}{4}F^a_{\mu\nu}F^{\mu\nu}_a,\end{aligned}$$

where – for simplicity – there is only one fundamental Dirac spinor field with mass m, $D_\mu = \partial_\mu + igA^a_\mu(x)T^a$ is the ***gauge covariant derivative*** (see equation (3.6)), T^a are the generators of the Lie-algebra of the gauge group and $A^a_\mu(x)$ are the (massless) gauge fields introduced in the previous section 3. The color index a runs from 1 to $N^2 - 1$, where N is degree of the special unitary symmetry group $SU(N)$. The ***field strength tensor*** is defined as (see equation (3.12))

$$F^a_{\mu\nu} = \partial_\mu A^a_\nu - \partial_\nu A^a_\mu - gf_{abc}A^b_\mu A^c_\nu.$$

[15]When replacing n_1 and n_2 with unit vectors in arbitrary direction μ and ν respectively, we can set $F^a_{12} = F^a_{\mu\nu}$.
[16]The normalisation convention is $tr(T^aT^b) = \frac{1}{2}\delta^{ab}$.

The (Minkowski) action is defined as usual, the integral over space-time of the Lagrangian

$$\mathcal{S}^M_{YM} = \int d^4x \mathcal{L}^M_{YM}.$$

We perform a ***Wick-rotation*** to obtain the Euclidean Lagrangian and action. This is done, because the Wick rotation in path integral formulation translates as $e^{i\mathcal{S}^M} \to e^{-\mathcal{S}^E}$, where $\mathcal{S}^E$ is a positive real number. The Euclidean path integral is then in the form of a classical statistical mechanics model, enabling us to interpret e^{-S^E} as probability density.

The Minkowski metric $\eta^{\mu\nu}$ becomes Euclidean if – through analytic continuation – we restrict the time coordinate to take imaginary values. The substitution is (for covariant and contravariant vectors)

$$\begin{aligned} t &\longrightarrow -i\tau, \\ x^0 &\longrightarrow -ix^4, \\ x_0 &\longrightarrow +ix_4, \end{aligned} \tag{3.17}$$

where $t \in \mathbb{R}$ is the Minkowski time coordinate and the real number τ is the Euclidean time coordinate. Equation (3.17) only holds in signature $(+, -, \ldots, -)$, else the signs in front of all the i would be opposite. The fields transform as well, and the transformed Euclidean fields take τ instead of t as time coordinate. We have to take care when transforming the fields and derivatives to Euclidean space-time. The spinor fields transform as

$$\begin{aligned} \psi(\vec{x}, t) &\longrightarrow S\psi_E(\vec{x}, \tau) \\ \psi(\vec{x}, t)^\dagger &\longrightarrow \psi_E(\vec{x}, \tau)^\dagger S, \end{aligned}$$

where S is a (invertible) matrix in spinor space that has to be determined. Since the gauge fields are vector quantities, they transform under Wick-rotation just as the coordinates $x^0 \to -ix^4$ and $x^k \to x^k$ with $k \in \{1, 2, 3\}$, but the fields appear with lower indices and are therefore covariant. In analogy to the space-time components $x_0 \to ix_4$, we have

$$\begin{aligned} A^{0,a}(\vec{x}, t) &\longrightarrow -i(A_E)^{4,a}(\vec{x}, \tau) \\ A^a_0(\vec{x}, t) &\longrightarrow +i(A_E)^a_4(\vec{x}, \tau) \\ A^a_k(\vec{x}, t) &\longrightarrow (A_E)^a_k(\vec{x}, \tau). \end{aligned}$$

Notice that when in Minkowski space μ takes values $0, 1, 2, 3$, $\mu = 0$ being the time component, but when in Euclidean space μ takes values $1, 2, 3, 4$, where $\mu = 4$ is the time component. Directly from equation (3.17), we obtain the rules for derivatives and integral measures

$$\begin{aligned} dt = dx^0 &\longrightarrow -idx^4 = -id\tau, \\ dx^k &\longrightarrow dx^k, \\ \partial_t = \partial_0 &\longrightarrow i\partial_4 = i\partial_\tau, \\ \partial^0 &\longrightarrow -i\partial^4, \\ \partial_k &\longrightarrow \partial_k. \end{aligned}$$

3.1.1 Fermionic part

Let's first transform the fermion and interaction part of $\mathcal{L}^M_{YM}$ (writing space and time components explicitly)

$$\mathcal{L}^M_F = \overline{\psi}\left(i\gamma^\mu\left(\partial_\mu + igA^a_\mu T^a\right) - m\right)\psi$$

$$
\begin{aligned}
&= \overline{\psi}(\vec{x},t)\left(i\gamma^\mu\partial_\mu - g\gamma^\mu A^a_\mu(\vec{x},t)T^a - m\right)\psi(\vec{x},t)\\
&\xrightarrow{\text{WR}} \psi^\dagger_E(\vec{x},\tau)Si\gamma^0\left(i\gamma^0 i\partial_4 + i\gamma^k\partial_k - g\gamma^0 i(A_E)^a_4(\vec{x},\tau)T^a - g\gamma^k(A_E)^a_k(\vec{x},\tau)T^a - m\right)S\psi_E(\vec{x},\tau).
\end{aligned}
$$

We want in the Euclidean Lagrangian the term $\gamma^\mu_E\partial_\mu$ to appear with Euclidean versions of the γ-matrices. To fulfill these requirements, we need the following theorem.

Theorem 3.2 (Constructing the Euclidean Clifford algebra)**.** *Let γ^μ obey the Clifford algebra, equation (3.1). Let S be an invertible operator in spinor space. Then the Euclidean γ-matrices defined as*

$$
\begin{aligned}
\gamma^4_E &\coloneqq S^{-1}\gamma^0 S,\\
\gamma^k_E &\coloneqq iS^{-1}\gamma^k S,
\end{aligned}
$$

satisfy the Euclidean Clifford algebra

$$
\{\gamma^\mu_E, \gamma^\nu_E\} = 2\delta^{\mu\nu}\cdot id,
$$

where $\delta^{\mu\nu}$ has signature $(+,+,\dots,+)$.

Proof. It is straight forward to check the properties

$$
\begin{aligned}
\{\gamma^4_E,\gamma^4_E\} &= \{S^{-1}\gamma^0S, S^{-1}\gamma^0S\}\\
&= 2S^{-1}\gamma^0SS^{-1}\gamma^0S\\
&= 2S^{-1}\gamma^0\gamma^0S\\
&= S^{-1}\{\gamma^0,\gamma^0\}S\\
&= 2\eta^{00}\cdot id\\
&= 2\delta^{44}\cdot id.
\end{aligned}
$$

Let $k,l\in\{1,2,3\}$

$$
\begin{aligned}
\{\gamma^k_E,\gamma^l_E\} &= \{iS^{-1}\gamma^kS, iS^{-1}\gamma^lS\}\\
&= -\{S^{-1}\gamma^kS, S^{-1}\gamma^lS\}\\
&= -S^{-1}\{\gamma^k,\gamma^l\}S\\
&= -S^{-1}2\eta^{kl}S\\
&= 2\delta^{kl}\cdot id.
\end{aligned}
$$

And finally, let $k\in\{1,2,3\}$

$$
\begin{aligned}
\{\gamma^4_E,\gamma^k_E\} &= \{S^{-1}\gamma^0S, iS^{-1}\gamma^kS\}\\
&= iS^{-1}\{\gamma^0,\gamma^l\}S\\
&= 0.
\end{aligned}
$$

□

It remains to determine the operator S. Since the Wick rotation does not affect space coordinates x^k, it should also not rotate the space components of the γ^k [19]. We therefore demand that the spatial γ-matrices commute with S and the temporal one satisfies

$$
\begin{aligned}
[S,\gamma^k] &= 0,\\
S\gamma^0 &= \gamma^0S^{-1}.
\end{aligned}
$$

The second equation can be motivated, because then the Dirac-adjoint transforms as $\overline{\psi}(\vec{x},t) \overset{\text{WR}}{\longrightarrow} \overline{\psi_E}(\vec{x},\tau)S^{-1}$, cancelling the S when multiplied with ψ. Using the above restrictions, we end up with

$$\begin{aligned} S &= e^{\frac{\theta}{2}\gamma^4\gamma^5}, \\ \gamma^4 &:= i\gamma^0, \\ \gamma^5 &:= \gamma^1\gamma^2\gamma^3\gamma^4, \end{aligned}$$

where θ is an arbitrary angle. Now we can replace the Minkowski γ-matrices with the Euclidean ones,

$$\begin{aligned} \mathcal{L}_F^M \overset{\text{WR}}{\longrightarrow} & \psi_E^\dagger(\vec{x},\tau)i\gamma^0\Big[-S^{-1}\gamma^0 S\partial_4 + iS^{-1}\gamma^k S\partial_k - m \\ & - igS^{-1}\gamma^0 S(A_E)_4^a(\vec{x},\tau)T^a - gS^{-1}\gamma^k S(A_E)_k^a(\vec{x},\tau)T^a\Big]\psi_E(\vec{x},\tau) \\ = & - \psi_E^\dagger(\vec{x},\tau)i\gamma^0\Big[(S^{-1}\gamma^0 S)\partial_4 - (iS^{-1}\gamma^k S)\partial_k + m \\ & + ig(S^{-1}\gamma^0 S)(A_E)_4^a(\vec{x},\tau)T^a - ig(iS^{-1}\gamma^k S)(A_E)_k^a(\vec{x},\tau)T^a\Big]\psi_E(\vec{x},\tau) \\ = & - \overline{\psi}_E(\vec{x},\tau)\Big[\gamma_E^4\partial_4 + \gamma_E^k\partial_k + m \\ & + ig\gamma_E^4(A_E)_4^a(\vec{x},\tau)T^a + ig\gamma_E^k(A_E)_k^a(\vec{x},\tau)T^a\Big]\psi_E(\vec{x},\tau) \\ = & - \overline{\psi}_E(\vec{x},\tau)\Big[\gamma_E^\mu\partial_\mu + m + ig\gamma_E^\mu(A_E)_\mu^a(\vec{x},\tau)T^a\Big]\psi_E(\vec{x},\tau). \\ = & - \overline{\psi}_E\Big[\gamma_E^\mu\partial_\mu + m + ig\gamma_E^\mu(A_E)_\mu^a T^a\Big]\psi_E = -\mathcal{L}_E. \end{aligned}$$

And we obtain the fermion and interaction part of the Euclidean Lagrangian $\mathcal{L}_E$. The action transforms as

$$\begin{aligned} i\mathcal{S}_F^M = i\int d^4x\mathcal{L}_F^M &= i\int d^3\vec{x}dt\mathcal{L}_F^M \\ &\overset{\text{WR}}{\longrightarrow} -i\int d^3\vec{x}(-id\tau)\mathcal{L}_F^E = -\int d^4x\mathcal{L}_F^E = -\mathcal{S}_F^E, \end{aligned}$$

as desired to model a probability density in the path integral.

3.1.2 Gauge part

Next, we look at the pure gauge Lagrangian $\mathcal{L}_G^M$. The Euclidean field strength tensor consists of derivatives and gauge fields, both transform equally under Wick-rotation. Thus, the trace over the field strength tensor transforms just as it is

$$\begin{aligned} \mathcal{L}_G^M &= -\frac{1}{4}F_{\mu\nu}^a F_a^{\mu\nu} \\ &\overset{\text{WR}}{\longrightarrow} -\frac{1}{4}(F_E)_{\mu\nu}^a(F_E)_a^{\mu\nu} = -\mathcal{L}_G^E. \end{aligned}$$

Using the same rules for derivatives and gauge fields, it derives as

$$(F_E)_{\mu\nu}^a = \partial_\mu(A_E)_\nu^a - \partial_\nu(A_E)_\mu^a - gf_{abc}(A_E)_\mu^b(A_E)_\nu^c.$$

3.2 Observables

In path integral formalism the ***expectation value*** of an arbitrary observable $\mathcal{O}$ written in terms of field degrees of freedom is given by

$$\langle \mathcal{O}[\psi, \bar{\psi}, A] \rangle := \lim_{t \to \infty(1-i\epsilon)} \frac{\int \mathcal{D}\psi \mathcal{D}\bar{\psi} \mathcal{D}A \mathcal{O}[\psi, \bar{\psi}, A] e^{iS[\psi,\bar{\psi},A]}}{\int \mathcal{D}\psi \mathcal{D}\bar{\psi} \mathcal{D}A e^{iS[\psi,\bar{\psi},A]}}. \tag{3.18}$$

The observable can be any combination of time-ordered products of gauge- and fermion-fields, so called ***time-ordered correlations functions*** of fields. The denominator is called ***partition function*** of the system and usually denoted by Z,

$$Z := \int \mathcal{D}\psi \mathcal{D}\bar{\psi} \mathcal{D}A e^{iS[\psi,\bar{\psi},A]}.$$

The expectation value (3.18) has an unconventional limit that has to be taken from the expression[17]. The limit has a slight tilt in the complex plane in order to suppress terms of the form $e^{-i\Delta E_n t}$, where $\Delta E_n := E_n - E$ denotes the energy difference of the state with quantum numbers n (n might be a multi-index) to the state considered in the observable. The term with the least suppressed weight and the lowest ΔE_m for some $E_m \approx E$ dominates the limit. Using that particular limit, exactly the state considered in the observable is projected out.

Since the fermion fields take Grassmannian values, they can be integrated analytically in the partition function using proposition B.3 resulting in

$$Z = \int \mathcal{D}A \det(D) e^{iS_G[A]},$$

where D is the ***Dirac-operator*** in continuum theory, $D = i\gamma^\mu D_\mu - m$. Unfortunately, the determinant of D depends on the gauge fields A^a_μ, thus we cannot pull the term out of the integral[18].

4 Lattice Gauge Theories

Non-abelian gauge theories with a certain number of fundamental fermionic particles can have the property of asymptotic freedom, meaning that the strength of the interaction becomes asymptotically weak as the distance between elementary particles decreases and the energy scale increases. Perturbatively such theories can be treated only in the high energy regime, where the running coupling constant is small to allow perturbative expansion of the problem. Consequently, at low energies the interaction is strong leading to confinement. Lattice gauge theory is a non-perturbative approach to deal with aforementioned theories in the low energy regime by discretising the problem on a finite space-time lattice. The finiteness of the lattice volume results in a momentum cut-off $\sim 1/a$ curing IR-divergences, where a is the lattice spacing. The finiteness of the lattice spacing on the other hand results in a cut-off $\sim a$ curing UV-divergences. Lattice discretisation of a field theory therefore acts as regularisation scheme.

4.1 Lattice Discretisation

To discretise the theory, from now on we work in $D = 4$ dimensional Euclidean space-time. The space-time is discretised in terms of a lattice.

Definition 4.1 (4D lattice). *Let $L_0, L_1, L_2, L_3 \in \mathbb{N}$ be natural numbers, define the **full lattice volume** as $V := L_0 \cdot L_1 \cdot L_2 \cdot L_3 = |\Lambda|$ and the **full 4D lattice** of volume V as*

$$\Lambda := \{n = (n_0, n_1, n_2, n_3) \mid n_i \in \{0, 1, \dots, L_i - 1\}, i \in \{0, 1, 2, 3\}\}.$$

[17]Only physicists come up with the idea of taking a limit that contains an infinitesimal ϵ multiplied by an infinitly growing quantity. Calling it unconventional barely does justice. Surprisingly, this expression is mathematically legal, because ϵ – although infinitesimal – is held constant while taking the limit. Nonetheless it hurts the eye.

[18]In the abelian theory it *is* independent and *can* be pulled out. In the expectation value (3.18) this determinant then cancels the one in the numerator very conveniently.

The points on the lattice are evenly spaced by the ***lattice spacing*** $a > 0$ (or ***lattice constant***), which is a real number of mass dimension -1[19]. This spans the (finite) 4D space-time lattice, where each point n[20] on the lattice corresponds to a physical point $x = an$ in space-time. The real continuum physics are put in a finite box of volume V. To extract a result valid in the continuum from such calculation one has to repeat the lattice calculation for multiple lattice spacings a. The final result is then extrapolated to $a \to 0$.

We start by discretising the gauge-part of the theory. Here, our effort in deriving the gauge-part using the plaquette will bear fruit. Note the gauge-part of the Lagrangian (the subscript E for "Euclidean" is dropped from now on)

$$\begin{aligned}\mathcal{L}_G &= \frac{1}{4} F^a_{\mu\nu} F^{\mu\nu,a} \\ &= \frac{1}{2} \sum_{\mu,\nu} tr(F^a_{\mu\nu} F^b_{\mu\nu} T^a T^b).\end{aligned}$$

In the second step, the sum over μ and ν was explicitly written. Notice that in Euclidean space-time upper and lower Lorentz-indices do not differ, $F^a_{\mu\nu} = F^{\mu\nu,a}$. The Lagrangian was constructed as the trace over the sum of plaquettes in all possible (μ,ν)-subspaces. To see this, we use the result from equation (3.11) with $n_1 = \hat{\mu}$ and $n_2 = \hat{\nu}$, where $\hat{\mu}$ is the unit vector in direction μ,

$$\begin{aligned}\hat{U}_{\mu\nu}(\epsilon, x) &= \exp\bigl(-ig\epsilon^2 F^a_{\mu\nu} T^a + O(\epsilon^3)\bigr) \\ &= \delta_{\mu\nu} \cdot id - ig\epsilon^2 F^a_{\mu\nu} T^a + iO(\epsilon^3) + \frac{1}{2!}(-ig\epsilon^2)^2 F^a_{\mu\nu} F^b_{\mu\nu} T^a T^b + \delta_{\mu\nu} O(\epsilon^6).\end{aligned}$$

This is the plaquette in subspace $(\hat{\mu}, \hat{\nu})$. In the last line, the exponential was written in terms of its series representation, where it has to be noted that there is *no* sum over the Lorentz-indices μ, ν in the expression above, although indices are repeated. When tracking down the $O(\epsilon^3)$-term, it turns out to be imaginary. The yellow term is the same as the one that is traced and summed over in the Lagrangian. Solving the equation for the yellow part and taking the trace of the real part on both sides (the trace over the yellow part itself is real),

$$tr(F^a_{\mu\nu} F^b_{\mu\nu} T^a T^b) = \frac{2}{g^2\epsilon^4} Re\ tr\left[\delta_{\mu\nu} \cdot id - \hat{U}_{\mu\nu}(\epsilon, x)\right] + \delta_{\mu\nu} O(\epsilon^2).$$

By taking the real part, the $O(\epsilon^3)$-term vanishes. Inserting this expression into the Lagrangian gives

$$\begin{aligned}\mathcal{L}_G &= \frac{1}{2} \sum_{\mu,\nu} tr(F^a_{\mu\nu} F^b_{\mu\nu} T^a T^b) \\ &= \frac{1}{g^2\epsilon^4} \sum_{\mu,\nu} Re\ tr\left[\delta_{\mu\nu} \cdot id - \hat{U}_{\mu\nu}(\epsilon, x)\right] + O(\epsilon^2).\end{aligned}$$

This is the continuum Lagrangian, now written in terms of plaquettes of size $\epsilon > 0$ instead of the field strength tensor and omitting terms of order ϵ^2. The term above is purely real by construction, thus e^{-S_G} can be interpreted as probability density.

Now we discretise the involved $SU(N)$-valued compensator fields $U(x, x + \epsilon\hat{\mu})$. At this point we can replace ϵ with the lattice spacing a[21] and only keep the fields in discrete space-time points an for

[19]We use particle physics units $c = \hbar = 1$ in which length and time have the same unit. It is therefore legal to use the same a as lattice spacing in all 4 space-time dimensions.

[20]n stand for a point in the lattice and is a four-vector containing only non-negative integers. Its components are dimensionless.

[21]The discretized theory can somehow be interpreted as the theory where the smallest distance resolvable is given by the (constant and finite) lattice spacing a and we ignore terms proportional to a^2, while in the continuum theory the resolution is given by the (variable, continuous and even infinitesimally small) ϵ. The replacement of ϵ with a comes naturally, since the theory before replacing only considered space-time points separated by a distance ϵ, when neglecting $O(\epsilon^2)$-terms.

$n \in \Lambda$. All fundamental fields and objects are therefore replaced by their discretised counterparts (the right-hand side defines the discretised quantity),

$$\begin{aligned}
U(x, x+\epsilon\hat{\mu}) = U(an, an+a\hat{\mu}) &\longrightarrow U_\mu(n),\\
\hat{U}_{\mu\nu}(\epsilon, x) = \hat{U}_{\mu\nu}(a, an) &\longrightarrow \hat{U}_{\mu\nu}(n),\\
A^b_\mu(x) = A^b_\mu(an) &\longrightarrow A^b_\mu(n),\\
\psi(x) = \psi(an) &\longrightarrow \psi(n).
\end{aligned}$$

Integrals and derivatives are replaced as well,

$$\begin{aligned}
\int dx^D f(x) &\longrightarrow a^D \sum_{n\in\Lambda} f(n),\\
\partial_\mu f(x) &\longrightarrow \frac{1}{2a}\left(f(n+\hat{\mu}) - f(n-\hat{\mu})\right).
\end{aligned} \tag{4.1}$$

Notice that the integral measure becomes a^D, which seems intuitive and naturally. The derivative on the other hand was discretised by using the fact that if the function $f(x)$ is smooth, then the right-hand and the left-hand derivatives are equal and thus $\partial_\mu = \frac{1}{2}(\partial_{+\mu} + \partial_{-\mu})$. Recalling the definition of the directional derivative along n, see equation (3.2), we define the ***left-hand*** or ***forward derivative*** in direction $n = +\hat{\mu}$ and the ***right-hand*** or ***backward derivative*** in direction $n = -\hat{\mu}$, respectively

$$\begin{aligned}
n^\mu\partial_\mu\psi(x) &\overset{n=+\hat{\mu}}{=} \lim_{\epsilon\to0^+}\frac{\psi(x+\epsilon\hat{\mu})-\psi(x)}{\epsilon} && =: \partial_{+\mu}\psi(x),\\
n^\mu\partial_\mu\psi(x) &\overset{n=-\hat{\mu}}{=} -\lim_{\epsilon\to0^+}\frac{\psi(x-\epsilon\hat{\mu})-\psi(x)}{\epsilon} = \lim_{\epsilon\to0^+}\frac{\psi(x)-\psi(x-\epsilon\hat{\mu})}{\epsilon} && =: \partial_{-\mu}\psi(x).
\end{aligned}$$

The replacement (4.1) very much looks like central differences with lattice spacing $a/2$, but we only have nearest neighbour interactions, and the error is smaller than with forward or backward differences[22]. The same holds for the covariant derivative, which is then replaced by

$$D_\mu f(x) = \frac{1}{2}\left[D_{+\mu} + D_{-\mu}\right] f(x) \longrightarrow \frac{1}{2a}\left[U_\mu(n) f(n+\hat{\mu}) - U_{-\mu}(n) f(n-\hat{\mu})\right],$$

with

$$\begin{aligned}
D_{+\mu} f(x) &= \lim_{\epsilon\to0^+}\frac{U(x,x+\epsilon\hat{\mu})f(x+\epsilon\hat{\mu}) - f(x)}{\epsilon} \longrightarrow \frac{1}{a}\left[U_\mu(n)f(n+\hat{\mu}) - f(n)\right],\\
D_{-\mu} f(x) &= \lim_{\epsilon\to0^+}\frac{f(x) - U(x,x-\epsilon\hat{\mu})f(x-\epsilon\hat{\mu})}{\epsilon} \longrightarrow \frac{1}{a}\left[f(n) - U_{-\mu}(n)f(n-\hat{\mu})\right].
\end{aligned}$$

The $\pm$ in $D_{\pm\mu}$, analogue to the directional derivatives, denotes left-hand or right-hand gauge covariant derivatives in direction μ. One usually uses the same lattice spacing a in all spatial and time directions ($\hbar = 1$). Of importance is only that the value of a is small, because it decides to what length and time scale the system can be resolved. The limit $a \to 0$ is called the ***naive continuum limit***.

The lattice is structured as in figure 2.

[22] Another – maybe even more important – reason for taking symmetric differences in the derivative is that then the resulting Dirac-operator obeys γ^5-***hermiticity***, $\gamma^5 D = (\gamma^5 D)^\dagger$. This property implies that the eigenvalues are either real or appear in complex conjugate pairs, making the determinant a real quantity.

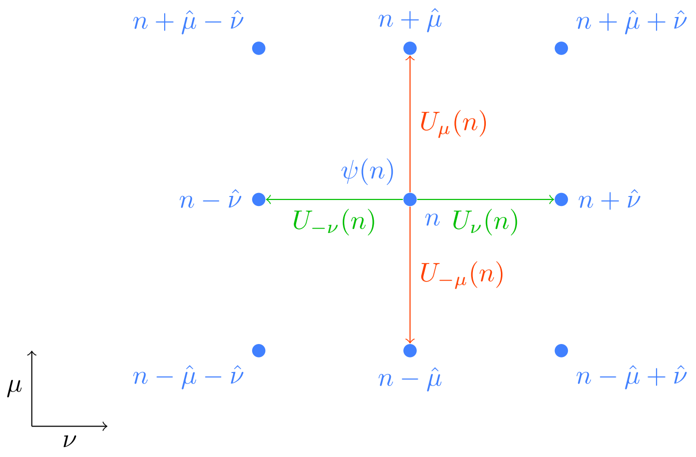

Figure 2: Cut-out of a lattice. The point in the middle is denoted by the integer four-vector n, the value of the fermion-field at that point is $\psi(x)$, with $x = na$. In this 2D slice, the field at that point is connected to its neighbouring lattice points $n \pm \hat{\mu}$ and $n \pm \hat{\nu}$ via the link variables $U_{\pm\mu}(n)$ and $U_{\pm\nu}(n)$ (coming from the discretised derivative). The link variables can be thought of as lying in-between the lattice points.

Since the compensator fields $U_\mu(n)$ (their discretised variants are usually called ***link variables***) connect neighbouring spinors $\psi(n)$ and $\psi(n+\hat{\mu})$ they reside visually "in-between" the lattice points. The index μ decorating the link variable is a directed one; it can take 8 values in a 4D lattice, the possible (Euclidean) values are $\mu \in \{\pm 1, \pm 2, \pm 3, \pm 4\}$. Using this notation, we have

$$U_\mu(n) = U_{-\mu}(n+\hat{\mu})^\dagger, \tag{4.2}$$

because the connection between lattice site n and $n+\mu$ is equal in both directions[23]. The spinor fields $\psi(n)_{\alpha,a}$ residing on the lattice points carry color- (a) and spinor indices (α). Therefore, for each lattice site, we have 12 complex numbers from the fermion fields and 4 $SU(3)$ matrices coming from the link variables.

The full discretised Yang-Mills action is

$$\mathcal{S} = \mathcal{S}_G + \mathcal{S}_F,$$

with

$$\mathcal{S}_G = \frac{1}{g^2}\sum_{n\in\Lambda}\sum_{\mu,\nu} Re\ tr\left[\delta_{\mu\nu}\cdot id - \hat{U}_{\mu\nu}(n)\right],$$

$$\mathcal{S}_F = a^4 \sum_{n\in\Lambda}\bar{\psi}(n)\underbrace{\left[\sum_\mu \frac{\gamma^\mu}{2}(D_{+\mu}+D_{-\mu}) + m\right]}_{:=D}\psi(n).$$

The expression in the square brackets between the fermion fields is the (naive) ***lattice Dirac-operator***. One of the problems with this term is that it brings so called ***doublers*** into the theory, artifacts from the lattice discretisation. Doublers are unphysical poles in the fermion-propagator D^{-1} and thus correspond to particles that do not exist in the continuum theory which has only one pole. In 4-dimensional space-time, the action above incorporates 15 unphysical doublers[24]. Wilson found a solution to this problem [20], namely he added a term proportional to a to the action which vanishes in the continuum limit. With $0 \neq r \in \mathbb{R}$, the ***Wilson-Dirac-operator*** then is

[23]This follows directly from the continuum version, equation (3.8), and holds in the lattice description as well.

[24]For general D-dimensional space-time the theory would contain $2^D - 1$ doublers. Of the total 2^D poles only one is physical. Here there might be a slight overflow in notation, D denotes the space-time dimension, while the Dirac-operator is also denoted as D. We assume that the meaning is clear from context.

$$D_W = \sum_\mu \frac{\gamma^\mu}{2}(D_{+\mu} + D_{-\mu}) + m - \frac{ar}{2}\sum_\mu D_{-\mu}D_{+\mu}. \tag{4.3}$$

This improvement eliminates the doubler states by making their mass proportional to r/a, such that they decouple in the continuum limit, but it breaks chiral symmetry. There are several other extensions of the action to deal with doublers and chiral symmetry breaking, for example the Sheikholeslami-Wohlert-term [21], the Lüscher-Weisz action [22], staggered fermions [23] or domain wall fermions [24]. Unfortunately, no improvement led to a lattice description with correct continuum limit maintaining chiral symmetry, while simultaneously introducing no doublers. It has even been proven that such a lattice discretisation does not exist [25]. However, equation (4.3) shows that there is a lot of flexibility in defining a lattice discretisation that has the correct continuum limit. Any term proportional to a power of a can be added to the action at the expense of increasing computational cost. Evidently continuous rotational, translational and Lorentz-symmetry is broken as well by any lattice description.

4.2 Hopping expansion

The Wilson-Dirac-operator (4.3) can be written as

$$D_W = C(id - \kappa H)$$

with

$$\begin{aligned} C &= m + \frac{4r}{a}, \\ \kappa &= \frac{1}{2(am+4r)}, \\ H &= \sum_\mu \Big[(rid + \gamma^\mu)U_\mu(n)\delta_{n+\hat{\mu},m} + (rid - \gamma^\mu)U_{-\mu}(n)\delta_{n-\hat{\mu},m}\Big]. \end{aligned}$$

The parameter κ is called the ***hopping parameter*** and H the ***hopping matrix***. Using a geometric series, the inverse of D_W can then be expanded in powers of κH, if $\|\kappa H\| < 1$,

$$D_W^{-1} = \frac{1}{C}\sum_{n=0}^{\infty} \kappa^n H^n.$$

The hopping expansion becomes useful when the mass m is large. Conversely, the mass is given by the hopping parameter

$$ma = \frac{1}{2\kappa} - 4r = \frac{1}{2\kappa} - \frac{1}{2\kappa_c},$$

where the critical value of $\kappa = \kappa_c := \frac{1}{8r}$ is defined to be where the mass approaches zero, usually $r = 1$.

4.3 Expectation values

The next step is to translate expectation values (3.18) to the lattice description of the theory. First, the ***Euclidean expectation value***

$$\langle \mathcal{O}[\psi, \bar{\psi}, A]\rangle := \lim_{\tau\to\infty} \frac{\int \mathcal{D}\psi\mathcal{D}\bar{\psi}\mathcal{D}A\mathcal{O}[\psi, \bar{\psi}, A]e^{-S[\psi,\bar{\psi},A]}}{\int \mathcal{D}A \det(D)e^{-S_G[A]}}.$$

The strange limit from the Minkowskian expectation value (3.18) has vanished, because of the Wick-rotation of the time-variable. If we put the determinant inside the exponential, we can redefine the (effective) Euclidean action as[25]

$$S[A] = S_G[A] + tr(\log D), \tag{4.4}$$

and write for observables including only gauge-fields

$$\langle \mathcal{O}[A] \rangle = \lim_{\tau\to\infty} \frac{\int \mathcal{D}A\mathcal{O}[A]e^{-S[A]}}{\int \mathcal{D}Ae^{-S[A]}}.$$

For fermionic observables we obtain inverses of the Dirac-operator,

$$\langle T[\psi_a^\alpha(x)\bar{\psi}_b^\beta(y)] \rangle = \lim_{\tau\to\infty} \frac{\int \mathcal{D}AD^{-1}(x,y)_{ab}^{\alpha\beta}e^{-S[A]}}{\int \mathcal{D}Ae^{-S[A]}}, \tag{4.5}$$

where $D^{-1}(x,y)_{ab}^{\alpha\beta}$ is the ***fermion-propagator*** from (x,α,a) to (y,β,b). The Euclidean action (4.4) is real and bounded from below for QCD at zero chemical potential [26]. While evaluating the functional integral exactly is clearly not feasible, but since the term e^{-S} in the Euclidean space-time can be seen as probability density, we can interpret the functional integral as expectation value in the statistical sense. Using the compensator fields U instead of the connections A_μ^a, a discretisation of this expression would look like

$$\langle \mathcal{O}[U] \rangle = \lim_{\tau\to\infty} \frac{1}{Z} \int \mathcal{D}Ue^{-S[U]}\mathcal{O}[U] \longrightarrow \sum_U P(U)\mathcal{O}[U].$$

The probability density in the last line is set to $P(U) := Z^{-1}e^{-S[U]}$. Although this expression seems better, it is still not realistic to evaluate explicitly, because if the lattice is large, we still have too many fields U to loop over within the sum[26]. The sum is weighted according to the probability density $P(U)$. Instead of randomly choosing N gauge-field configurations to approximate the sum[27], we could try to avoid field configurations with small weights $P(U)$, by focusing on configurations that minimize the action. By this, we can approximate the sum using much fewer terms – terms that dominate the sum. The expectation value then takes the form of an arithmetic mean of these considered field configurations – a sample mean

$$\langle \mathcal{O}[U] \rangle \approx \frac{1}{N} \sum_{i=1}^N \mathcal{O}[U_i], \tag{4.6}$$

where U_i denotes the i-th field configuration and N is the total number of configurations considered. Clearly, the U_i in the expression above are sampled according to the probability density $P(U)$. This is called ***importance sampling***. The statistical error is $O(N^{-\frac{1}{2}})$. A large part of active and past research in lattice gauge theories consists of finding clever trickery to calculate the sum and restricting it to only the best field configurations U_i. An approximation called quenched QCD (QQCD) just sets $\det(D) = const.$, making $tr(\log D) = 0$, leading to a theory similar to QED, because the determinant cancels just as in the abelian case. Another approach is to use the hopping expansion to calculate the determinant,

$$\det(D) = C\det(id - \kappa H) = C\exp\big(tr\log(id - \kappa H)\big) = C\exp\left(-\sum_{n=1}^\infty \frac{\kappa^n}{n} tr(H^n)\right).$$

The sum in the exponential will then be truncated after some powers.

[25]Using the well-known identity $\det(e^A) = e^{tr(A)}$.

[26]On an L^4 lattice, the quatitiy U consists of $2 \cdot 8 \cdot L^4$ real numbers (if a representation with 8 real number was used for the $SU(3)$ matrices). Even if we restrict every continuous component of U to only 100 discrete possibilities, we end up in 100^{16L^4} possible configurations.

[27]Which would be a perfectly fine technique, but introduces a large spread of values, thus a large variance. If $P(U)$ is peaked in a certain small region, a plain sampling could fail to have even one point inside the relevant region.

4.4 Pseudofermions

One way to introduce physical fermions into the lattice description of our theory is to rewrite the determinant in the expectation value (3.18) in terms of fermion fields (having spinor indices) that behave statistically as bosonic fields (they commute), so called ***pseudofermions***[28]. Assume we have two fundamental fermions in the theory ψ_u and ψ_d (for example the up- and down-quark) of approximately the same mass. The same mass implies that their Dirac-operators are equal $D_u = D_d = D$ and the path integral looks like

$$\int \mathcal{D}\psi_u \mathcal{D}\bar{\psi}_u \mathcal{D}\psi_d \mathcal{D}\bar{\psi}_d e^{-\bar{\psi}_u D \psi_u - \bar{\psi}_d D \psi_d}.$$

If the the determinant of D is real, using propositions B.4 and B.3 we can write

$$\int \mathcal{D}\psi_u \mathcal{D}\bar{\psi}_u \mathcal{D}\psi_d \mathcal{D}\bar{\psi}_d e^{-\bar{\psi}_u D \psi_u - \bar{\psi}_d D \psi_d} = \pi^{-V} \int \mathcal{D}\phi \mathcal{D}\phi^\star e^{-\phi^\star (D^\dagger D)^{-1} \phi}.$$

In this expression $\psi_u, \bar{\psi}_u, \psi_d, \bar{\psi}_d$ are complex Grassmann-valued fields and $\phi, \phi^\star$ are regular complex-valued fields, $V = |\Lambda|$ is the number of lattice points. The integral over Grassmann variables has been reduced to an integral over regular numbers. Thus, the Dirac-operator has to be inverted for a given vector ϕ to construct the (effective) ***pseudofermion action*** for the probability weight (4.6)

$$S = S_G[A] + \phi^\star (D^\dagger D)^{-1} \phi. \tag{4.7}$$

The irrelevant factor π^{-V} can be absorbed in a redefinition of the fields ϕ and $\phi^\star$. An advantage of this description over equation (4.4) is that the determinant is a highly non-local term in the sense that is depends on the state of the entire system. By introducing pseudofermions, the non-locality of equation (4.4) can be reduced [27], if one has an even number of mass-degenerate fundamental fermion flavors.

4.5 Hybrid Monte Carlo

In openQ*D, an algorithm called Hybrid Monte-Carlo (HMC) is used to generate the gauge- and fermion-field configurations that are then evaluated in the expectation value (4.6). A Monte Carlo simulation consists of creating a Markov chain of field configurations according to a probability distribution. A ***Markov chain*** is a sequence of random values where each value depends only on its predecessor. When a new field configuration is accepted[29], it is called a ***Monte Carlo step*** or ***update step***. The crucial ingredient is that field configurations with a large weight e^{-S} are visited more often than configurations with a small weight. Finding these random quantities with a large weight is a non-trivial task for multiple reasons: boundary condition have to be obeyed, $SU(3)$ valued quantities are not trivial to generate, strong ergodicity in the transition probability from one configuration to another has to be concerned [28, 29], a high acceptance rate is desirable since every step in the Markov chain includes inversions of the Dirac-operator [30], the random number generator has to be able to produce large amounts of random numbers in a short time [31] and many more.

The HMC algorithm [32] features reduced correlations among Markov steps and thus fewer required samples in order to produce enough field configurations to evaluate the sample mean. Once high, the acceptance probability can be kept high, because HMC conducts Markov steps among field configurations that approximately conserve energy, thus the action is conserved as well. We will not discuss this vast and fascinating topic further since it would go far beyond the scope of this document.

[28]There is some similarity to Faddeev–Popov ghosts in the sense that both fields are unphysical. Ghosts are anti-commuting bosonic (scalar) fields that have to be introduced when fixing the gauge in non-abelian gauge theories, while pseudofermions are commuting fermionic fields.

[29]Via an acceptance probability.

5 OpenQ*D

The open-source software openQ*D code has been used by the RC* collaboration for the generation of fully dynamical QCD+QED gauge configurations with C* boundary conditions and $O(a)$-improved Wilson-fermions. Its codebase is written completely parallel using MPI and thus designed for portable and scalable large-scale parallel applications. Its scalability is only limited by the CPU- and memory-resources of classical supercomputing platforms[30]. With its parallel design approach, enormous lattice sizes of $96^4 \times n$ are achievable, where n is the number of ranks/processes used by the MPI library.

Unique features of openQ*D include the possibility to use C* boundary conditions and its solid solver implementations are specially designed for ill-conditioned systems. The deflated solver algorithm is extended by the ability to use more than one deflation subspace in a single simulation [17]. Additionally, the molecular dynamics evolution of the $U(1)$-field is equipped with the possibility to employ Fourier acceleration. The optimized implementation of the Dirac-operator is inherited from the openQCD package [33]. The application is highly configurable given as a human-readable INI-style input file holding all adjustable parameters, except the lattice size and the number of ranks[31].

Notable as well is the archaic C-89 standard and the style of code that reflects the age of the software[32]. On the other hand, the clean and concise memory layout must be stressed and the coherency in the code that always shows the same coding and naming principles throughout the entire codebase.

6 Real number formats

6.1 IEEE Standard for Floating-Point Arithmetic

Floating-point numbers are omnipresent in scientific applications. In the Conjugate Gradient (CG) kernel of openQ*D [17], large scalar products appear over vectors of high dimensionality over multiple ranks. The components of these vectors are single precision floating-point numbers (called binary32 from here on). The precision was reduced from binary64 to binary32 already and a speedup of roughly a factor of 2 was achieved. This motivates to explore even smaller floating-point formats with encoding lengths of 16 bits. Since scalar products as well as matrix-vector products are memory-bound operations, going to a smaller bit-length will increase the throughput of the calculation. Therefore, a 16 bits floating-point format with a smaller exponent could lead to a double of performance if the new operation is still memory-bound.

Definition 6.1 (IEEE 754 Floating-point format)**.** *The* ***IEEE 754 floating-point format*** *[34] is defined using the* ***number of exponent bits*** *e and the* ***number of mantissa bits*** *m respectively. A binary floating-point number is illustrated in figure 3.*

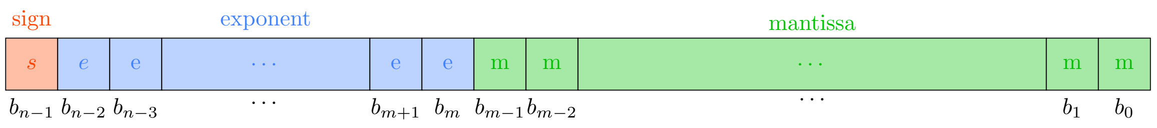

Figure 3: Binary representation of an IEEE 754 n-bit precision floating-point number. The orange bit represents the sign bit, the blue bits represent the fixed-length e exponent bits and the green bits represent the fixed-length m mantissa bits. Notice that $n = 1 + e + m$.

[30]The term "classical" in this context refers to supercomputing facilities without GPU accelerators.

[31]Which are given in a global header file containing application constants, meaning that the application must be recompiled when changing the lattice size or the process layout.

[32]It has the typical caveats that come with older software that grew over time, for example: no comments within the code, excessive use of pointer arithmetic or code that is broken by compiler optimization flags. This comment should by no means be misunderstood as being disrespectful, it's an opinion of a modern developer. I just wanted to constructively mention difficulties that made it (unnecessarily) hard for me as a newcomer to engage with the code, naively assuming that other beginners might have a similar experience.

The resulting floating-point number is then calculated as

$$f = (-1)^{s} \cdot M \cdot 2^{E},$$

where $E = E' - B$ denotes the ***biased exponent***, B is the ***exponent bias***, M the ***mantissa*** and s the ***sign*** bit. The (unbiased) exponent E' is calculated as follows

$$E' = \sum_{i=0}^{(e-1)} b_{m+i} 2^{i}, \tag{6.1}$$

where B is the exponent bias.

Definition 6.2 (Exponent bias)**.** *The **exponent bias** is given by*

$$B = 2^{(e-1)} - 1.$$

The calculation of the mantissa M is a bit more involved, since it depends on the number being normal or subnormal.

Definition 6.3 (Subnormal numbers)**.** *The IEEE 754 standard introduces so called **subnormal numbers**. If all the exponent bits are* 0*, meaning the unbiased exponent* $E' = 0$*, but the mantissa bits are not all* 0*, then the number is called subnormal. The exponent being zero causes the implicit bit to flip to* 0*, instead of* 1*.*

Remark. Subnormal numbers have a variable-length mantissa and exponent, because some of the mantissa bits are used as additional exponent bits, making the numbers less precise the smaller they are (see the smooth cut-off in figure 5).

Therefore, the mantissa of a regular (non-subnormal) number is (when the exponent $0 < E < B$, this implies that the implicit bit is 1)

$$M = \underset{\text{implicit bit}}{1} + \sum_{i=1}^{m} b_{m-i} 2^{-i},$$

while the mantissa of a subnormal number (when the exponent $E = 0$) is

$$M = \underset{\text{implicit bit}}{0} + \sum_{i=1}^{m} b_{m-i} 2^{-i},$$

The 1 or 0 in the front of the summand is the leading ***implicit bit***, sometimes also called the $(m+1)$***-th mantissa bit*** that tells us whether the number is subnormal or not.

Remark. The mantissa range of a regular floating-point number is $M \in [1, 2)$, while the mantissa range of a subnormal floating-point number is $M \in (0, 1)$. The number zero is not considered subnormal.

The most common floating-point formats are summarised in table 1.

The highest representable number is when the exponent is highest. This is not the case when all e exponent bits are 1, because then – according to the specification [34] – the number is either $\pm\infty$ or not a number (NaN), depending on the mantissa. The maximal unbiased exponent is therefore the next smaller number,

$$E'_{max} = \underbrace{1...1}_{e-1 \text{ times}} 0.$$

[33] Allocates 32 bits, but only 19 bits are used.

Floating-point formats				
name	s	e	m	comment
binary64	1	11	52	double precision, IEEE 754 [35]
binary32	1	8	23	single precision, IEEE 754 [35]
binary16	1	5	10	half precision, IEEE 754 [35]
bfloat16	1	8	7	Googles Brain Float [36]
tensorfloat32	1	8	10	NVIDIAs TensorFloat-32 [37] [33]
binary24	1	7	16	AMDs fp24 [38]
binary128	1	15	112	IEEE 754 [35]
binary256	1	19	236	IEEE 754 [35]

Table 1: Commonly used floating-point formats, where s is the number of sign bits, e the number of exponent bits and m the number of mantissa bits.

Using equation (6.1), we find

$$E'_{max} = \sum_{i=1}^{(e-1)} 2^i = 2^e - 2.$$

The mantissa on the other hand is maximal when all mantissa bits are 1 (including the implicit bit),

$$M_{max} = 1 + \sum_{i=1}^{m} 2^{-i} = 2 - 2^{-m}.$$

Using these two formulas we can define the

Definition 6.4 (highest representable number)**.** *The* ***Highest representable number*** *in any floating-point format is*

$$\begin{aligned} f_{max} &= (-1)^0 \cdot M_{max} \cdot 2^{(E'_{max}-B)} \\ &= (2-2^{-m}) \cdot 2^{(2^e-2^{e-1}-1)} \\ &= (2-2^{-m}) \cdot 2^{(2^{e-1}-1)}. \end{aligned}$$

The minimal number above 0 can be found similarly, using minimal unbiased exponent (when all exponent bits are 0, except the last one, therefore $E'_{min} = 1$) and the minimal mantissa ($M_{min} = 1$).

Definition 6.5 (Minimal (non-subnormal) representable number above 0)**.** *The* ***Minimal (non-subnormal) representable number above*** 0 *in any floating-point format is*

$$\begin{aligned} f_{min} &= (-1)^0 \cdot M_{min} \cdot 2^{(E'_{min}-B)} \\ &= 2^{(2-2^{e-1})}. \end{aligned}$$

The minimal subnormal number can the found, when the unbiased exponent consists of only zeros ($E'_{smin} = 0$) and for the mantissa only the rightmost bit is one ($M_{smin} = 2^{1-m}$).

Definition 6.6 (Minimal subnormal representable number above 0)**.** *The* ***minimal subnormal representable number above*** 0 *in any floating-point format is*

$$\begin{aligned} f_{min} &= (-1)^0 \cdot M_{smin} \cdot 2^{(E'_{smin}-B)} \\ &= 2^{1-m} \cdot 2^{(1-2^{e-1})} \\ &= 2^{(2-m-2^{e-1})}. \end{aligned}$$

Floating-point format limits				
name	f_{max}	f_{min}	f_{smin}	sign. digits [34]
binary64	1.8×10^{308}	2.2×10^{-308}	4.9×10^{-324}	≤ 15.9
binary32	3.4×10^{38}	1.2×10^{-38}	1.4×10^{-45}	≤ 7.2
binary16	6.6×10^{4}	6.1×10^{-5}	6.0×10^{-8}	≤ 3.3
bfloat16	3.4×10^{38}	1.2×10^{-38}	9.2×10^{-41}	≤ 2.4
tensorfloat32	3.4×10^{38}	1.2×10^{-38}	1.1×10^{-41}	≤ 7.2
binary24	1.8×10^{19}	2.2×10^{-19}	3.3×10^{-24}	≤ 5.1
binary128	1.2×10^{4932}	3.4×10^{-4932}	6.5×10^{-4966}	≤ 34
binary256	$1.6 \times 10^{78,913}$	$1 \times 10^{-78,912}$	$1 \times 10^{-78,983}$	≤ 71.3

Table 2: Summary of highest representable numbers, minimal subnormal and non-subnormal representable numbers above 0 in any IEEE 754 floating-point format together with their approximated precision in decimal.

See table 2 for these limiting numbers in the different floating-point formats.

6.2 Posits

The posit data type is designed to be a replacement for the IEEE floating-point format, fixing its various quirks. Some of them are:

- The appearance of NaNs. They are considered unnatural, because a specific bit pattern describing a number that is not a number is a contradiction.
- The NaNs and the fact that floats have two different bit-representations for the number zero (0 and -0) lead to overly complicated and slow comparison units as well as (funny) theoretical contradictions[35].
- Floats may under- or overflow because the standard employs the round to nearest even rounding rule ($\pm\infty$ and 0 are considered even).
- Floats are non-associative and non-distributive[36] leading to rounding errors that must be considered, especially in scientific computing.
- The standard gives no guarantee of bit-identical results across systems.

The goal is to utilise the number of bits more efficiently and remove these inconsistencies. The key idea is to place half of all numbers between 0 and 1 and the other half are the reciprocals (the reciprocal of 0 being $\pm\infty$). The number can then be drawn on a projective real number circle [40]. The structure of a binary posit number is illustrated in figure 4.

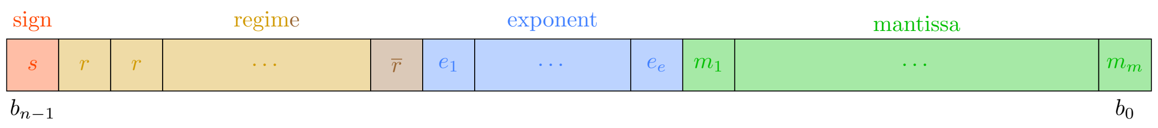

Figure 4: Binary representation of an n-bit posit number. As with regular floats the orange bit represents the sign bit, but the yellow bit(s) represent the variable length regime bit(s) terminated by the brown bit that is the opposite regime bit, the blue bit(s) represent the variable-length exponent bit(s) and the green bit(s) represent the variable-length mantissa bit(s).

[34]Number of significant digits in decimal; $-\log_{10}(\texttt{MACHINE\_EPSILON}) = \log_{10}(2^{m+1})$.

[35]According to IEEE 754 floating-point arithmetic, it holds $\frac{1}{\infty} = 0$ and $\frac{1}{-\infty} = -0$, but we have $0 = -0$. This implies $\infty = -\infty$, but (also according to IEEE 754) $\infty > -\infty$.

[36]There was even a system using IEEE 754 that had non-commutative floating-point operations [39].

The actual value of the number is calculated as follows: the yellow and brown bits determine the regime of the number. They either start with a row of all 0 or all 1 terminated by the opposite bit indicating the end of the row. The number of bits in the row are counted as m and if they are all 0 they get a minus sign, the regime is then $k = -m$. If they are all 1 the regime is calculated as $k = m - 1$. After the regime is decoded, the remaining bits contain the exponent with at most es bits depending on how many bits remain. If no bits remain the exponent is 0. The exponent and the mantissa are both of variable length. Both can have 0 bits, in this case the number consists of only regime bits. This is the reason posits have a larger number range than floats. The exponent is encoded as unsigned integer, so there is no bias and no bit pattern denoting special numbers such as subnormals or NaNs. Therefore n-bit posits have more numbers than n-bit floats because they have no NaNs. However, after the maximal es exponent bits – if there are still bits remaining – the fraction follows, else the fraction is defined as 1.0. Since there are no subnormals the implicit bit is always 1. There are two special numbers that do not follow the above encoding scheme: zero which has the bit pattern of all 0 and $\pm\infty$ with a 1 followed by all 0. These two numbers are reciprocals of each other. A general posit number can therefore be written as

$$p = (-1)^{s} \cdot useed^{k} \cdot M \cdot 2^{E},$$

where s is the sign bit, $useed$ is defined as $useed = 2^{2^{es}}$, with es the number of predefined exponent bits, M is the mantissa and E the exponent.

The mantissa is calculated as

$$M = 1 + \sum_{i=1}^{m} m_i 2^{m-i},$$

where m is the variable number of mantissa bits and the implicit bit in front of the sum is always 1. The exponent is

$$E = \sum_{i=1}^{e_v} e_i 2^{e_v - i},$$

where e_v is the variable number of exponent bits satisfying $e_v \leq es$.

Using these two equations, we are now able to calculate the highest representable number and the minimal representable number above 0 in posit format.

Definition 6.7 (Highest representable number)**.** *The* ***highest representable number*** *in any posit format is*

$$\begin{aligned} p_{max} &= (-1)^0 \cdot useed^{n-2} \\ &= 2^{2^{es}(n-2)}. \end{aligned}$$

Definition 6.8 (Minimal representable number above 0)**.** *The* ***minimal representable number above*** 0 *in any posit format is the reciprocal of the highest representable number* p_{max}*,*

$$\begin{aligned} p_{min} &= \frac{1}{p_{max}} \\ &= 2^{2^{es}(2-n)}. \end{aligned}$$

See table 3 for these limiting numbers in the different posit formats.

Posits employ a feature called the ***quire***. The quire is the generalised answer to the fused multiply–add operation that recently found its way into [35] in 2008, where the rounding is deferred to the very end of the operation.

[37]Number of significant digits in decimal; $-\log_{10}($`MACHINE_EPSILON`$)$. Notice that posits have ***tapered accuracy***; numbers near 1 have much more precision than numbers at the borders of the regime. The precision of floats decreases as well with very large and small numbers, but posit precision decreases faster, see figure 5.

Posit format limits				
name	es	p_{max}	p_{min}	sign. digits [37]
posit64	3	2.0×10^{149}	4.9×10^{-150}	≤ 17.7
posit32	2	1.3×10^{36}	7.5×10^{-37}	≤ 8.1
posit16	1	2.7×10^{8}	3.7×10^{-9}	≤ 3.6
posit8	0	64	1.6×10^{-2}	≤ 1.5

Table 3: Summary of highest representable numbers, minimal representable numbers above 0 in any posit format together with their approximated precision.

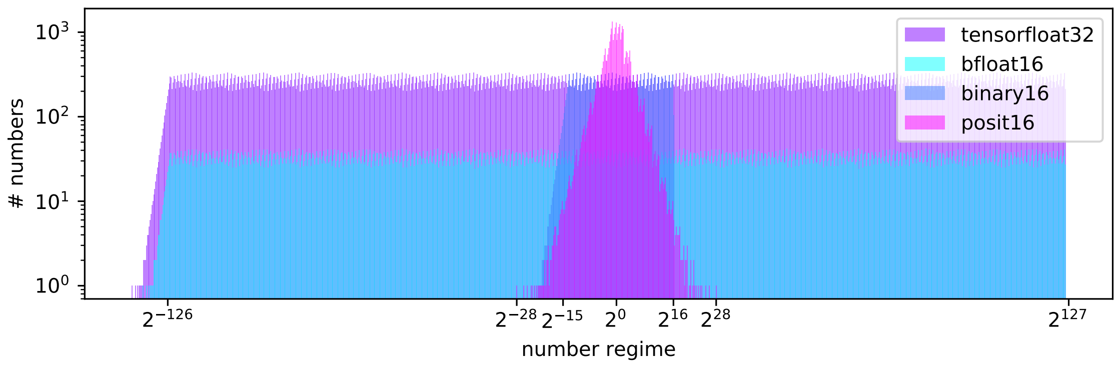

Figure 5: Density and distribution of numbers for tensorfloat32, binary16, posit16 and bfloat16. The number of bins was chosen to be 1024 of logarithmic width. The y-axis gives the amount of numbers in the number regime given by the x-axis. The IEEE conformant floats tensorfloat32, binary16 and bfloat16 show a similar shape, namely the distribution of numbers is exponentially decreasing for higher and smaller numbers. The high numbers undergo a rough cut-off at the highest representable number. Numbers above that value will be cast to infinity. Compared to this, the small numbers show a smooth cut-off, because of the existence subnormal numbers. The range of posit16 is larger than the range of binary16, but especially in their smallest numbers this difference in range is negligible. Some features of posits can be observed. First, their distribution is symmetric around 1, because posits have no subnormals and the reciprocal of every number is in the set as well. Second, more numbers are closer to 1 than in case of floats; the closer to 1, the better the number resolution. Closest to 1, the number resolution becomes better than binary16 resolution. Third, posits have neither a fixed-length mantissa nor exponent. That is the reason the height of the posit shape depends on the number regime, which happens for floats only in the subnormal regime, where the exponent and mantissa are indeed of variable length. For all formats, the amount of numbers decreases exponentially when going away from 1, but posits decrease faster. This suggests that when calculating in the number regime close to 1 posits might be the better choice, but when numbers span the entire number range equally, floats might be superior. In the latter case one has to take care about over- and underflows. Notice that the height of the shape is determined by the number of mantissa bits, therefore giving the precision, while the width is determined by the number of exponent bits, therefore giving the number range. For example, tensorfloat32 and binary16 have a vastly different number range but show the same precision for numbers in their intersection, meaning that binary16 is a subset of tensorfloat32. On the other hand, tensorfloat32 and bfloat16 have approximately the same number range, but different precisions in them, meaning that bfloat16 is a subset of tensorfloat32, which itself is a subset of binary32. Notice that when plotting binary32 and posit32 in such a plot, they would look similar to binary16 vs. posit16.

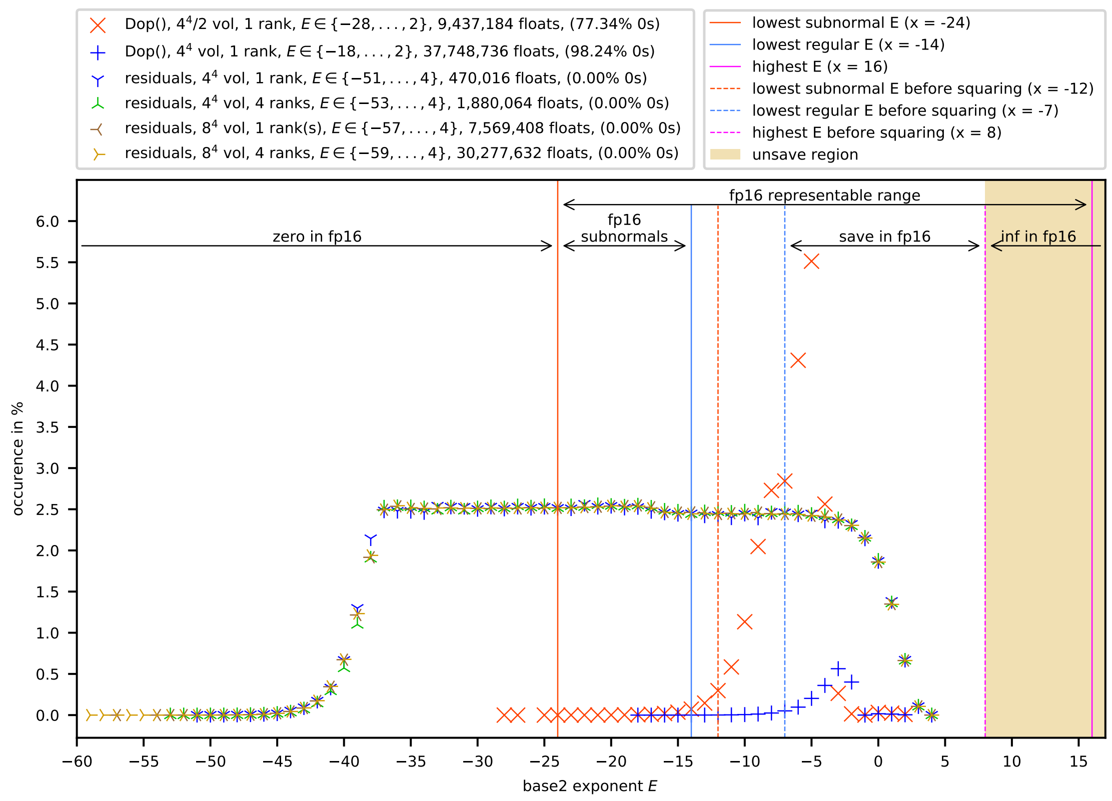

Figure 6: Exponent distribution of binary32 single precision floats in the residual vectors of all steps in a conjugate gradient run in openQ*D as well as entries of the Dirac-operator. 4 runs were made, with local lattice size of 4^4 and 8^4 on one single rank and 4 ranks, respectively. The number is normalised to $(-1)^s \cdot M \cdot 2^E$, where $M \in [1, 2)$. For the Dirac-operator with volume $4^4/2$ only the even lattice points were considered.

6.3 Floating-point numbers in openQ*D

To explore how the conjugate gradient kernel in openQ*D would perform when using smaller bit lengths, one can look at the exponential of the numbers in the matrix and vectors, see figure 6. The plot shows all exponents appearing together with their overall occurrence in percent. The number zero was taken out of the plot, because it has biased exponent $E = -127$. The occurrences for zero are given in the legend.

The highest exponent in all 4 runs was $E = 4$, while the lowest exponent decreased when the number of lattice points increased. The range of exponents that is representable in binary16 spans from -24 to $+16$ and is indicated by the solid orange line and the solid pink line. Between -24 and -14 is the regime of subnormal numbers in binary16, with the lowest regular (non-subnormal) exponent indicated by the solid blue line. When using half precision instead of single precision, all numbers with exponents below -24, will be converted to zero, while exponents above $+16$ will be cast to $\pm\infty$ depending on the sign of the number. It can be seen, that when calculating the norm of these numbers, only numbers between the dashed blue line and the dashed pink line will participate. If there is a number above the dashed pink line in the unsafe region this number will – after squaring – be cast to ∞ and therefore the norm will be ∞ as well[38]. In this case the variable holding the norm $x = \|\vec{v}\|$ should be of higher precision than binary16. The plot shows that the Dirac matrix `Dop()` is confined in a narrow exponent regime and a representation in 16-bit floats would suffice. Notice the sparsity of the Dirac matrix.

[38]A method to circumvent this is to scale the vector entries during the calculation and scale the result back, exploiting homogeneity of the norm, $\|\vec{v}\| = \frac{1}{s}\|s\vec{v}\|$ for $s \in \mathbb{R}_{>0}$.

7 Conjugate Gradient

Systems of linear equations need to be solved in many scientific computations. Usually, these systems are large, and the matrices and vectors are distributed among many ranks. The method to solve such systems should therefore be iterative. The problem can be formulated mathematically in the following way.

7.1 Derivation

Let $n \in \mathbb{N}$ and A be an $n \times n$-matrix with components in $\mathbb{C}$, Hermitian, positive definite and sparse,

$$\begin{aligned} & A^\dagger = A, && (\textbf{\textit{Hermitian}}) \\ \forall \vec{x} \in \mathbb{C}^n \setminus \{0\} \quad : \quad & \vec{x}^\dagger A \vec{x} > 0, && (\textbf{\textit{positive definite}}) \end{aligned}$$

as well as $\vec{b} \in \mathbb{C}^n$ be given, then the ***system of linear equations*** can be described as

$$A\vec{x} = \vec{b}. \tag{7.1}$$

We are interested in the ***solution*** vector $\vec{x}$, that is the one that satisfies the above equation, n is called the ***problem size***. First let us define a function that will be helpful in the next sections.

Definition 7.1 (Quadratic form)**.** *The* ***quadratic form*** *depends on the problem matrix A as well as on the* ***source*** *vector $\vec{b}$ and is defined as*

$$f(\vec{x}) = \frac{1}{2}\vec{x}^\dagger A \vec{x} - \vec{b}^\dagger \vec{x} + c,$$

where $c \in \mathbb{C}$ (see figure 7).

When taking the derivative of this function with respect to $\vec{x}$, we find that

$$f'(\vec{x}) = A\vec{x} - \vec{b}.$$

Therefore, finding the extrema of $f(\vec{x})$ is equivalent to solving the linear system of equations (7.1). The question whether the solution $\vec{x}$ is unique remains.

Lemma 7.1 (Uniqueness of the solution)**.** *The solution $\vec{x}$ in equation (7.1) is unique and the global minimum of $f(\vec{x})$ if A is Hermitian and positive definite* [39].

Proof. Let us rewrite $f(\vec{p})$ at an arbitrary point $\vec{p} \in \mathbb{C}$ in terms of the solution vector $\vec{x}$:

$$f(\vec{p}) = f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger A (\vec{p} - \vec{x}). \tag{7.2}$$

This is indeed the same as $f(\vec{p})$ (inserting $A\vec{x} = \vec{b}$ and using $A^\dagger = A$ and $\vec{a}^\dagger \vec{b} = \vec{b}^\dagger \vec{a}$),

$$\begin{aligned} f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger A (\vec{p} - \vec{x}) &= \frac{1}{2}\vec{x}^\dagger A \vec{x} - \vec{b}^\dagger \vec{x} + c + \frac{1}{2}\vec{p}^\dagger A \vec{p} - \frac{1}{2}\vec{p}^\dagger A \vec{x} - \frac{1}{2}\vec{x}^\dagger A \vec{p} + \frac{1}{2}\vec{x}^\dagger A \vec{x} \\ &= \frac{1}{2}\vec{p}^\dagger A \vec{p} + c + \vec{x}^\dagger \vec{b} - \vec{b}^\dagger \vec{x} - \vec{b}^\dagger \vec{p} \\ &= \frac{1}{2}\vec{p}^\dagger A \vec{p} - \vec{b}^\dagger \vec{p} + c \end{aligned}$$

[39] Negative definiteness is sufficient as well and $\vec{x}$ would be the global maximum instead – define $A' = -A$ which is positive definite and all the argumentation that follows will hold as well. Indefinite matrices on the other hand might have local minima and maxima.

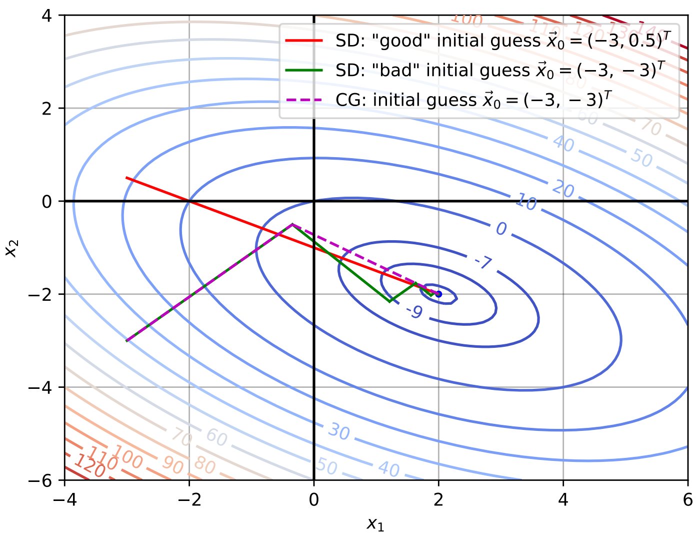

Figure 7: Example of a quadratic form $f(\vec{x})$ with $A = \begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix}$, $\vec{b} = (2, -8)^T$ and $c = 0$. The analytic minimum is $\vec{x}^* = (2, -2)^T$. The red and green lines show two runs of steepest descent, with different starting vectors $\vec{x}_0$. The dashed magenta line shows a conjugate gradient run (notice that it only needed 2 steps).

$$= f(\vec{p}).$$

In the new form of $f(\vec{p})$, one can directly see that if A is positive definite, $\vec{x}$ must minimise the function:

$$f(\vec{p}) = f(\vec{x}) + \frac{1}{2} \underbrace{(\vec{p} - \vec{x})^\dagger A (\vec{p} - \vec{x})}_{> 0 \text{ if } A \text{ pos. def.}}.$$

Therefore $\vec{x}$ is the global unique minimum.

□

Before deriving the conjugate gradient method, we look at a related method called the ***method of steepest descent***. We are interested in a method that iteratively solves equation (7.1) starting at a ***initial guess*** $\vec{x}_0$ until the series is interrupted, because the approximate solution $\vec{x}_i$ might be close to the real solution by a certain tolerance or the solution was found exactly,

$$\vec{x}_0 \longrightarrow \vec{x}_1 \longrightarrow \cdots \longrightarrow \vec{x}_i \longrightarrow \vec{x}_{i+1} \longrightarrow \cdots$$

For each step, we can define the ***error*** and ***residual*** of the current step i.

Definition 7.2 (Error and residual)**.** *Define the* ***error*** $\vec{e}_i$ *and the* ***residual*** $\vec{r}_i$ *as*

$$\vec{e}_i = \vec{x}_i - \vec{x}, \tag{7.3a}$$

$$\vec{r}_i = \vec{b} - A\vec{x}_i. \tag{7.3b}$$

The residual is the vector of discrepancies and the same as $\vec{r}_i = -f'(\vec{x}_i) = -A\vec{e}_i$, the negative derivative of the quadratic form. The derivative points in direction of the maximal increase, thus the residual points in direction of the steepest descent seen from the position of point $\vec{x}_i$.

Definition 7.3 (Method of Steepest Descent (SD)). *The iteration step equation of the* ***method of steepest descent*** *in defined as*

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{r}_i, \tag{7.4}$$

where the $\alpha_i \in \mathbb{C}$ are the amounts to go in direction of the current residual $\vec{r}_i$. The α_i are determined by minimising the parabola with respect to α_i, $\frac{d}{d\alpha_i} f(\vec{x}_{i+1}) \overset{!}{=} 0$.

Remark (Convergence). The method of steepest descent converges very slowly to the actual solution, when starting at an unfavourable starting point $\vec{x}_0$ (see figure 7). The speed of convergence heavily depends on the condition number of the matrix A. The process happens to go in the same direction multiple times. If we only go one step in each direction i by the perfect amount α_i, we would be done after at most n steps.

This gives motivation for an enhanced method. Let's define a new ***step equation*** as

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{p}_i, \tag{7.5}$$

with ***directions*** $\vec{p}_i$ and ***amounts*** α_i which we have to determine. But this time, we will impose the condition to go in every direction only once at most. This will lead us to the ***method of conjugate gradient***.

Using the step equation (7.5), we can update the error and residuals,

$$\begin{aligned} \vec{e}_{i+1} &= \vec{x}_{i+1} - \vec{x} && \text{(7.6a)} \\ &= \vec{e}_i + \alpha_i \vec{p}_i && \text{(7.6b)} \\ &= \vec{e}_0 + \sum_{j=0}^{i} \alpha_j \vec{p}_j, && \text{(7.6c)} \end{aligned}$$

$$\begin{aligned} \vec{r}_{i+1} &= \vec{b} - A\vec{x}_{i+1} && \text{(7.7a)} \\ &= \vec{r}_i - \alpha_i A \vec{p}_i && \text{(7.7b)} \\ &= -A\vec{e}_{i+1}. && \text{(7.7c)} \end{aligned}$$

The $\{\vec{p}_i\}$ need to form a basis of $\mathbb{C}^n$, because the method should succeed with any arbitrary initial guess $\vec{x}_0$. Since we move in the vector space $\mathbb{C}^n$ from an arbitrary point $\vec{x}_0$ to the solution $\vec{x}$, the n direction vectors need to cover all possible directions in the space, therefore need to be linear independent.

To be done after at most n steps, we require the n-th error to be zero, $\vec{e}_n = 0$. Since the directions form a basis, we can write $\vec{e}_0$ as a linear combination of the $\{\vec{p}_i\}$,

$$\vec{e}_0 = \sum_{j=0}^{n-1} \delta_j \vec{p}_j.$$

Using this we can rewrite $\vec{e}_n$,

$$\begin{aligned} \vec{e}_n &= \vec{e}_o + \sum_{j=0}^{n-1} \alpha_j \vec{p}_j \\ &= \sum_{j=0}^{n-1} \delta_j \vec{p}_j + \sum_{j=0}^{n-1} \alpha_j \vec{p}_j \\ &= \sum_{j=0}^{n-1} (\delta_j + \alpha_j) \vec{p}_j. \end{aligned}$$

For this to be zero, all coefficients need to be zero, thus $\delta_j = -\alpha_j$. Then the i-th error can be rewritten as

$$\begin{aligned}
\vec{e}_i &= \vec{e}_0 + \sum_{j=0}^{i-1} \alpha_j \vec{p}_j \\
&= \sum_{j=0}^{n-1} \delta_j \vec{p}_j - \sum_{j=0}^{i-1} \delta_j \vec{p}_j \\
&= \sum_{j=i}^{n-1} \delta_j \vec{p}_j .
\end{aligned} \tag{7.8}$$

In the last row, we can see that after every step in the iteration, we shave off the contribution of one direction $\vec{p}_i$ to the initial error $\vec{e}_0$ (or equivalent: $\vec{e}_{i+1}$ has no contribution from previous directions $\vec{p}_i, \vec{p}_{i-1}, \dots$). We still need to find these directions. We could for example impose that the $(i+1)$-th error should be orthogonal to the i-th direction, because we never want to go in that direction again,

$$\begin{aligned}
0 &\overset{!}{=} \vec{p}_i^\dagger \vec{e}_{i+1} \\
&= \vec{p}_i^\dagger (\vec{e}_i + \alpha_i \vec{p}_i).
\end{aligned}$$

This gives us an expression for the amount α_i,

$$\alpha_i = -\frac{\vec{p}_i^\dagger \vec{e}_i}{\vec{p}_i^\dagger \vec{p}_i}.$$

The problem with this expression is that we don't know the value of $\vec{e}_i$ – if we would, we could subtract it from the current $\vec{x}_i$ and obtain $\vec{x}$ exactly. We do not know $\vec{e}_i$, but what we know is something similar, namely $-A\vec{e}_i$, with is the residual. If we manage to insert an A in the expression above, we reached the goal. It turns out that imposing A-orthogonality instead of regular orthogonality between $\vec{e}_{i+1}$ and $\vec{p}_i$ accomplishes the desired result[40],

$$\begin{aligned}
0 &\overset{!}{=} \vec{p}_i^\dagger A \vec{e}_{i+1} \\
&= \vec{p}_i^\dagger A (\vec{e}_i + \alpha_i \vec{p}_i).
\end{aligned}$$

Solving for α_i gives the (almost) final expression for the amounts,

$$\implies \quad \alpha_i = -\frac{\vec{p}_i^\dagger A \vec{e}_i}{\vec{p}_i^\dagger A \vec{p}_i} = \frac{\vec{p}_i^\dagger \vec{r}_i}{\vec{p}_i^\dagger A \vec{p}_i}. \tag{7.9}$$

Notice that the denominator is never zero, because A is positive definite. Let us continue with the expression for A-orthogonality, but insert the derived expression (7.8) for $\vec{e}_{i+1}$ this time,

$$\begin{aligned}
0 &\overset{!}{=} \vec{p}_i^\dagger A \vec{e}_{i+1} \\
&= \vec{p}_i^\dagger A \left[\sum_{j=i+1}^{n-1} \delta_j \vec{p}_j \right]
\end{aligned}$$

[40]This is equivalent to imposing $0 \overset{!}{=} \vec{r}_{i+1}^\dagger \vec{p}_i$ which is done in most literature, but in the opinion of the author this is less intuitive.

$$= \sum_{j=i+1}^{n-1} \underbrace{\delta_j}_{\neq 0} \vec{p}_i^\dagger A \vec{p}_j.$$

This implies that for $j > i$ and $i \in \{0, \dots, n-1\}$, we have

$$\vec{p}_i^\dagger A \vec{p}_j = 0.$$

But since A is Hermitian, we can Hermitian conjugate the entire expression above and obtain

$$0 = \left(\vec{p}_i^\dagger A \vec{p}_j\right)^\dagger = \vec{p}_j^\dagger A \vec{p}_i.$$

So the expression holds for $i > j$ as well, which implies that the $\{\vec{p}_i\}$ are ***A-orthogonal***,

$$\vec{p}_i^\dagger A \vec{p}_j = 0 \quad \forall i \neq j.$$

The problem has reduced to finding a set of A-orthogonal vectors in an iterative way. Luckily, there is a well know method to find orthogonal vectors from a set of linear independent vectors: ***Gram-Schmidt orthogonalization***. The procedure can be altered to find A-orthogonal vectors instead.

Definition 7.4 (Gram-Schmidt Orthogonalization)**.** *Let $\{\vec{u}_0, \dots, \vec{u}_{n-1}\} \subset \mathbb{C}^n$ be a set of n linear independent vectors. The iterative Gram-Schmidt procedure is*

$$\begin{aligned} \vec{p}_0 &= \vec{u}_0 \\ \vec{p}_i &= \vec{u}_i + \sum_{k=0}^{i-1} \beta_{ik} \vec{p}_k, \end{aligned} \tag{7.10}$$

where the $\beta_{ik} \in \mathbb{C}$ are coefficients (to be determined).

In the regular procedure, the β_{ik} are normalised projections of $\vec{u}_i$ to $\vec{p}_k$ that are subtracted from $\vec{u}_i$, leading to a vector $\vec{p}_i$ that is orthogonal to all previously calculated $\vec{p}_k$. In our problem, we need a set of vectors that are A-orthogonal. By imposing this condition, we find a different expression for the β_{ik},

$$\begin{aligned} 0 &\overset{!}{=} \vec{p}_i^\dagger A \vec{p}_j \\ &= \vec{u}_i^\dagger A \vec{p}_j + \sum_{k=0}^{i-1} \beta_{ik} \vec{p}_k^\dagger A \vec{p}_j \\ &= \vec{u}_i^\dagger A \vec{p}_j + \beta_{ij} \vec{p}_j^\dagger A \vec{p}_j, \end{aligned}$$

where in the last step, we assumed $i > j$ (else we would not find an expression for β_{ij}) and therefore only the j-th term in the sum remains, because of the A-orthonormality of the directions. Solving this for β_{ij} gives

$$\beta ij = -\frac{\vec{u}_i^\dagger A \vec{p}_j}{\vec{p}_j^\dagger A \vec{p}_j}. \tag{7.11}$$

In principle we are done here, we only need a set of linearly independent vectors $\{\vec{u}_i\}$. Since the conjugate gradient method is iterative and often dealing with huge problem sizes n, we need to store all previous directions $\vec{p}_k$ in order to calculate the current direction (see equation (7.10)). This becomes a problem in limited memory situations. We want that the current step only depends on the previous one. By imposing this condition, we need the sum in equation (7.10) to collapse;

the β_{ik} should only be non-zero for $k = i-1$. If we manage to satisfy this, the orthogonalization procedure would simplify to

$$\begin{aligned}\beta_i &\coloneqq \beta_{i,i-1},\\ \vec{p}_i &= \vec{u}_i + \beta_i \vec{p}_{i-1},\end{aligned}$$

where in the second equation, the current $\vec{p}_i$ only depends on the previous $\vec{p}_{i-1}$. For this to hold, all other β_{ij} need to be zero. For such a β_{ij} the numerator needs to be zero. Let therefore $j < i-1$

$$\vec{u}_i^\dagger A \vec{p}_j \overset{!}{=} 0.$$

To find a different expression for the left-hand side, consider

$$\begin{aligned}\vec{u}_i^\dagger \vec{r}_{j+1} &= \vec{u}_i^\dagger \left(\vec{r}_j - \alpha_j A\vec{p}_j\right)\\ &= \vec{u}_i^\dagger \vec{r}_j - \alpha_j \vec{u}_i^\dagger A \vec{p}_j,\\ \implies \vec{u}_i^\dagger A\vec{p}_j &= \frac{1}{\alpha_j}\left[\vec{u}_i^\dagger \vec{r}_j - \vec{u}_i^\dagger \vec{r}_{j+1}\right],\end{aligned} \tag{7.12}$$

where we inserted the recursive relation of the residuals (7.7b) and the yellow part is the expression we want to be zero for $j < i-1$. We therefore find a condition for the linear independent set $\{\vec{u}_i\}$, namely that the scalar product of $\vec{u}_i$ with $\vec{r}_{j+1}$ and $\vec{r}_j$ must be the same. We can apply the same equation multiple times and obtain for $j < i-1$

$$\vec{u}_i^\dagger \vec{r}_{j+1} = \vec{u}_i^\dagger \vec{r}_j = \cdots = \vec{u}_i^\dagger \vec{r}_0.$$

We have to find $\{\vec{u}_i\}$ that satisfy the above equation. It is sufficient to find a set of $\{\vec{u}_i\}$ that are orthogonal to all the residuals in order for the equation to be obeyed.

Lemma 7.2. *The residuals are orthogonal, thus for all $i \neq j$, it holds*

$$\vec{r}_i^\dagger \vec{r}_j = 0.$$

Proof. The proof consists of 2 steps.

1) Let $i < j$,

$$\begin{aligned}\vec{p}_i^\dagger \vec{r}_j &= -\vec{p}_i^\dagger A \vec{e}_j\\ &= -\sum_{k=j}^{n-1} \delta_j \vec{p}_i A \vec{p}_k\\ &= 0,\end{aligned}$$

where the yellow expression is zero, because $i < j \leq k$.

2) Let $i < j$. By step 1), we have

$$\begin{aligned}0 &= \vec{p}_i^\dagger \vec{r}_j\\ &= \vec{r}_i^\dagger \vec{r}_j + \sum_{k=0}^{i-1} \beta_{ik} \vec{p}_k^\dagger \vec{r}_j\\ &= \vec{r}_i^\dagger \vec{r}_j.\end{aligned}$$

The yellow expression is again zero by step 1). Using the symmetry of the scalar product, the above equation holds for i and j interchanged ($i > j$) too, therefore holds for all $i \neq j$.

□

In the following, we set $\vec{u}_i = \vec{r}_i$. What remains to find is the final expression for the β_i.

$$\begin{aligned}\beta_i := \beta_{i,i-1} &= -\frac{\vec{u}_i^\dagger A\vec{p}_{i-1}}{\vec{p}_{i-1}^\dagger A\vec{p}_{i-1}} \\ &= -\frac{1}{\vec{p}_{i-1}^\dagger A\vec{p}_{i-1}}\frac{1}{\alpha_{i-1}}\left[\vec{r}_i^\dagger\vec{r}_{i-1} - \vec{r}_i^\dagger\vec{r}_i\right] \\ &= \frac{\vec{r}_i^\dagger\vec{r}_i}{\alpha_{i-1}\vec{p}_{i-1}^\dagger A\vec{p}_{i-1}} \\ &= \frac{\vec{r}_i^\dagger\vec{r}_i}{\vec{p}_{i-1}^\dagger\vec{r}_{i-1}},\end{aligned}$$

where in the first row we used the definition (7.11), in the second row we have used equation (7.12) and the yellow expression is zero by the orthogonality of the residuals, lemma 7.2. In the last line we used the expression for the α_j equation (7.9).

To obtain the final form of the α_i and the β_i, we can use the proof of lemma 7.2, namely

$$\begin{aligned}\vec{p}_i^\dagger\vec{r}_i &= \vec{r}_i^\dagger\vec{r}_i + \beta_i\underbrace{\vec{p}_{i-1}^\dagger\vec{r}_i}_{=0 \text{ by lemma 7.2 step 1)}} \\ &= \vec{r}_i^\dagger\vec{r}_i.\end{aligned}$$

Using this we find the final form of the α_i and the β_i as well as the ***method of conjugate gradient***.

Definition 7.5 (Method of conjugate gradient (CG)). *The iteration step equation of the* ***method of conjugate gradient*** *in defined as*

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i\vec{p}_i,$$

with

$$\begin{aligned}\vec{r}_{i+1} &= \vec{r}_i - \alpha_i A\vec{p}_i, & \alpha_i &= \frac{\vec{r}_i^\dagger\vec{r}_i}{\vec{p}_i^\dagger A\vec{p}_i}, \quad (7.13)\\ \vec{p}_{i+1} &= \vec{r}_{i+1} + \beta_{i+1}\vec{p}_i, & \beta_{i+1} &= \frac{\vec{r}_{i+1}^\dagger\vec{r}_{i+1}}{\vec{r}_i^\dagger\vec{r}_i}, \quad (7.14)\end{aligned}$$

and initial starting vectors

$$\begin{aligned}\vec{x}_0 &= \textit{arbitrary starting point}, \quad (7.15)\\ \vec{p}_0 &= \vec{r}_0 = \vec{b} - A\vec{x}_0.\end{aligned}$$

There are some remarks to note about the method of conjugate gradient.

Remark. The β_{i+1} of the current iteration depends on the norm of the current residual as well as the last one. This means that we can store the result of the last iteration and reuse it in the current, the norm may not be calculated twice.

Remark. In the source code of openQ*D (see [17]) the matrix A is the Dirac matrix applied twice $A = D^\dagger D$. This means that the denominator of α_i is a regular inner product as well; $\vec{p}_i^\dagger A\vec{p}_i = \vec{p}_i^\dagger D^\dagger D\vec{p}_i = (D\vec{p}_i)^\dagger (D\vec{p}_i) = \|D\vec{p}_i\|^2$.

Remark. Therefore in each iteration, we have:

- 2 times the norm of a vector,

- 2 matrix-vector multiplications,
- 3 times axpy.[41]

Remark (Floating-point errors). Since the method introduces recursive steps, floating-point round-off accumulation has to be dealt with. This causes the residuals to lose their A-orthogonality. It can be resolved by calculating the residual from time to time using its (computationally more expensive) definition $\vec{r}_i = \vec{b} - A\vec{x}_i$, which involves one matrix vector multiplication (or two is $A = D^\dagger D$). One can for example do this every m-th step. The same problem applies to the directions $\vec{p}_i$ that lose their A-orthogonality.

Remark (Problem size). The method of conjugate gradient is suitable for problems of enormous sizes n. The algorithm is done after n steps (see figure 7), but there might be problems such that even n steps are out of reach for an exact solution.

Remark (Complexity). The time complexity of the conjugate gradient method is $O(m\sqrt{\kappa})$, where m is the number of non-zero entries in A and κ is its ***condition number***. The space complexity is $O(m)$.

Remark (Starting). The ***starting vector*** $\vec{x}_0$ can be chosen arbitrarily. If there is already a rough estimate of the solution one can take that vector. Usually, $\vec{x}_0 = 0$ is chosen. Since the minimum is global, there is no issue in choosing a starting point. The method will always converge towards the real solution.

Remark (Stopping). If the problem size does not allow to run n steps, one can stop when the norm of the residual falls below a certain ***threshold*** ϵ. Usually, this threshold is a fraction of the initial residual $\|\vec{r}_i\| < \epsilon\|\vec{r}_0\|$, [41].

Remark (Initialisation). The very first step of the method is equivalent to a step in the method of steepest descent, see equation (7.4) as well as figure 7.

Remark (Speed of convergence). Conjugate gradient is faster if there are duplicated eigenvalues. The number of iterations for an exact solution is at most the number of distinct eigenvalues.

Remark (Preconditioning). The linear system of equations can be transformed using a matrix M to

$$M^{-1}A\vec{x} = M^{-1}\vec{b}.$$

It is assumed M is such that it is easy to invert, and it approximates A in some way, resulting in $M^{-1}A$ to be better conditioned than A[42]. An example of a particular preconditioner M is a diagonal matrix, with diagonal entries of D. It is indeed easy to invert, and it approximates A quite well if A has non-zero diagonal entries and most off-diagonal entries are zero.

Remark (Conjugate Gradient on the normal equations (CGNE)). The algorithm can be used even if A is neither symmetric nor Hermitian nor positive definite. The linear system of equations to be solved is then

$$A^\dagger A\vec{x} = A^\dagger\vec{b}.$$

If A is square and invertible, solving the above equation is equivalent to solving $A\vec{x} = \vec{b}$. Conjugate gradient can be applied, because $A^\dagger A$ is Hermitian and positive definite ($\vec{x}^\dagger A^\dagger A\vec{x} = \|A\vec{x}\| \geq 0$). Notice that $A^\dagger A$ is less sparse than A, and often $A^\dagger A$ is badly conditioned.

7.2 CG kernel in openQ*D

The conjugate gradient kernel `cgne()` in `modules/linsolv/cgne.c` in [17] implements the algorithm, see listing 1. The algorithm is already implemented in mixed precision using binary32 in most of the computations and binary64 in correction steps[43].

[41]This stands for $a\vec{x} + \vec{y}$, scalar times vector plus vector, "a x plus y" (to resemble the BLAS level 1 routine call of the same name).

[42]Or equivalent, $M^{-1}A$ has a more clustered spectrum than A.

[43]The method is also referred to as ***mixed precision defect-correction***, see [42]

```
double cgne(int vol,int icom,void (*Dop)(spinor *s,spinor *r),
            void (*Dop_dble)(spinor_dble *s,spinor_dble *r),
            spinor **ws,spinor_dble **wsd,int nmx,double res,
            spinor_dble *eta,spinor_dble *psi,int *status)
```

Listing 1: The conjugate gradient kernel in `modules/linsolv/cgne.c` line 429ff. The volume of the lattice is given by `vol`, `icom` is a communication control parameter, `Dop()` and `Dop_dble()` are the Dirac-operators in single and double-precision, `ws` and `wsd` are workspace allocations. The remaining parameters are the maximum number of iterations `nmx`, the relative residue `res`, the source spinor `eta`, the starting spinor, which also holds the result after the run `psi`. `status` holds the number of CG-steps performed. The return value is the norm of the residual of the current solution.

```
if ((rn<=tol)||(rn<=(PRECISION_LIMIT*xn))||(ncg>=100)||
    ((*status)>=nmx))
   break;
```

Listing 2: Break condition in `modules/linsolv/cgne.c` line 490ff, `rn` is the norm of the current residual, `xn` is the norm of the current solution vector, both in binary32. The other parameters are the tolerance `tol`, the current iteration number `status`, the current iteration number since the last restart step `ncg` and the maximal number of iterations `nmx`.

The function requires the Dirac-operator `Dop()` in binary32, `Dop_dble()` in binary64 format and the source vector `eta` ($\vec{b}$) in binary64 only. In the initialisation, the starting vector `psi` ($\vec{x}_0$) is set to zero. The algorithm stops when the desired maximal relative residue `res` $= \frac{\|\texttt{eta}-D^\dagger D\texttt{psi}\|}{\|\texttt{eta}\|}$ is reached, where `psi` is the calculated approximate solution of the Dirac equation $D^\dagger D\texttt{psi} = \texttt{eta}$ in binary64. For this, the tolerance `tol` is calculated using `tol` $= \|\texttt{eta}\| *$ `res`. The parameter `nmx` is the maximal number of iterations that may be applied and `status` reports the total number of iterations that were needed, or a negative value if the algorithm failed. `icom` is a control parameter and `ws` and `wsd` are workspace allocations. The volume of the local lattice should be provided in `vol` when calling the function.

Since the Dirac-operator is given in two precisions, the algorithm in the code bails out of the main conjugate gradient loop, when some particular conditions are met, see listing 2.

This may happen in 4 cases:

1. if the recursively calculated residual is below the tolerance,
2. if the precision of binary32 is reached[44],
3. after a hard coded number of 100 steps since the last restart step,
4. if the maximal number of steps `nmx` is reached.

We discuss in the following the second condition in more detail: assume the precision of binary32 is reached, but the algorithm does not break out of the main loop. Therefore, the norm of the current residual compared to the norm of the current solution vector differ in their orders of magnitude by the precision limit of their data type (binary32 in this case). This means that the solution vector $\vec{x_i}$ contains large numbers compared to the residual vector $\vec{r_i}$. Therefore, the changing in residual in successive iterations is negligible compared to numbers in $\vec{x_i}$ as well. Since $\vec{r_i}$ contains small numbers, the amounts α_i are small too. This causes $\vec{x_{i+1}} = \vec{x_i} + \alpha_i \vec{d_i}$ to not change anymore, because adding very large and very small numbers in floating-point arithmetic will return the larger number unchanged if the two numbers differ in magnitude by the precision limit of the data type. The algorithm stalls in that case and breaking out of the main loop is the emergency brake in such a case.

When one of the above conditions is met, the algorithm performs a ***reset step*** (or ***restart step***). In a reset step, the residual is not calculated in the recursive way (as described above in

[44]The constant `PRECISION_LIMIT` is defined to be `100*MACHINE_EPSILON`, where the `MACHINE_EPSILON` is the difference between 1 and the lowest value above 1 depending on the data type. In case of binary32 the `MACHINE_EPSILON` takes a value of $1.192{,}092{,}9 \times 10^{-7}$.

definition (7.5)) but by its definition $\vec{r_i} = \vec{b} - A\vec{x_i}$ in double precision. Such a step involves two invocations of each `Dop_dble()` as well as `Dop()` which is expensive. The algorithm is resetting in the sense that the solution vector is set back to $\vec{x_i} = 0$, but before resetting, the solution vector in binary32 is added to the real solution vector `psi` in binary64 which was initialised to zero at the start of the algorithm. It looks like a restart of the entire calculation, but the direction for the next iteration $\vec{d_i} = \vec{r_i}$ is set to the just calculated, more accurate residual. Therefore, the algorithm now continues in a new direction A-orthogonal to all previous directions and progression is kept. The step is meant to remove the accumulated round-off errors due to the recursive calculation of the residuals and directions. The first step following a reset step is a step in the direction of steepest descent like the very first step of the algorithm. The less precise the data type, the more reset steps are required, because the precision limit (case 2) is reached earlier.

7.3 Simulating CG with different data types

Some operations such as norms, scalar products and applications of the Dirac-operator are memory-bound, which means the memory bandwidth of the controller and main memory determine how much time is spent in performing the operation. Storing input data in a format with lower bit-length reduces the amount of data to be transferred from and to main memory, thus improving the speed of calculation.

7.3.1 Setup

The complete conjugate gradient kernel was simulated in different data types, floats as well as posits. To produce the plot series, the Dirac matrix `Dop_dble()` and the source vector `eta` were extracted in binary64 format from the original code running a simulation of a 4^4 lattice, Schrödinger functional (SF) boundary conditions (`type 1`), no C* boundary conditions (`cstar 0`) and 1 rank. The first 2000 trajectories were considered thermalisation. The matrix was extracted in trajectory 2001. A Python script mimicking the exact behaviour of the `cgne()` kernel from the source code, was implemented to cope with arbitrary data types. The simulated data types were binary64, binary32, tensorfloat32, binary16, bfloat16, posit32, posit16, and posit8. The considered Dirac-operator represented as a CSR-matrix had approximately 2% non-zero values. The results are plotted in figures 8 - 11.

7.3.2 Discussion of figures 8 - 11

Figures 8, 9, 10 and 11 contain all relevant data. It is expected in general that the plots show data types of the same bit-length in clusters and exhibit a hierarchy in precision and exponent range; more precision and larger exponent range should end up in faster convergence. Thus, we expect the following hierarchy (where smaller means convergence in fewer steps)

$$\text{binary64} < \text{posit32} \leq \text{binary32} \leq \text{tensorfloat32} \leq (1) \leq \text{posit16} \leq \text{binary16} \leq (2) < \text{posit8}, \tag{7.16}$$

where bfloat16 could be either at position (1) or (2), depending on what is more important, precision or number range.

Figure 8: In figure 8 where the original data type is naively[45] replaced by the simulated data type, it can be concluded that only data types with large enough number ranges converged. binary64, binary32 and posit32 converged each after `status=27` steps with one reset step. The less precise tensorfloat32 took `status=27` (+3) and the even less precise bfloat16 required `status=32` (+5) steps. Such a hierarchical result was expected since they have the same exponent range and thus approximately the same number range but differ only in precision (see table 1). Notice that the less precise the data type, the more reset steps are needed. This happens because the precision limit of the simulated data type is reached earlier if the data type has less precision.

The round-off accumulation error of posit32 is slightly better than the one of binary32, although defeated by 8 orders of magnitude of binary64 because of its much more precision. It is notable to remark that the round-off accumulation does not increase significantly in successice steps, what

[45]Naive in this context means that *all* appearances of binary32-variables in the cgne-kernel were blindly replaced by variables of simulated data type.

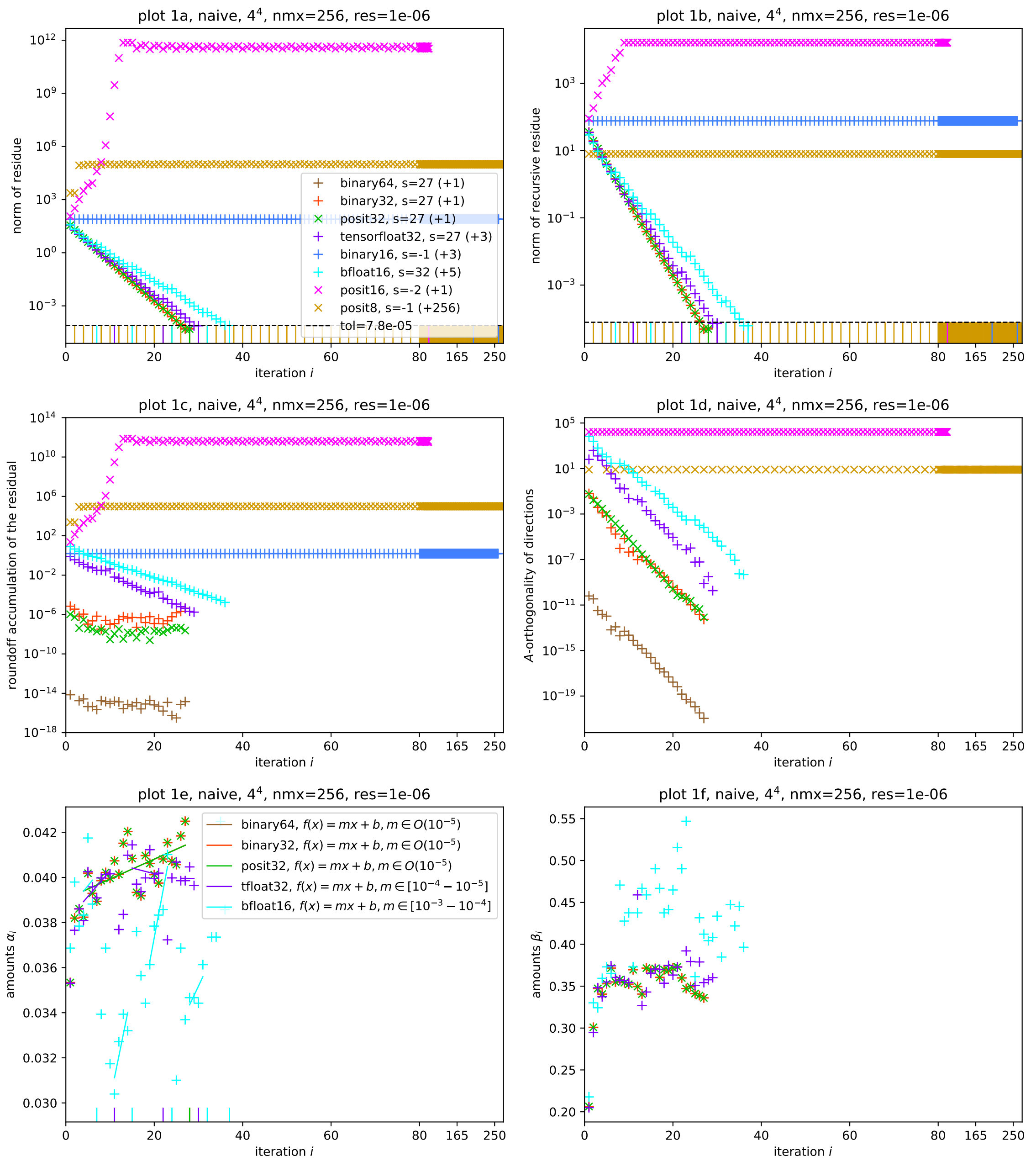

Figure 8: Convergence analysis of a conjugate gradient run, where binary32 was replaced by one of the simulated data types. The number s describes the number of normal steps needed (the value of `status`), while the number in the brackets indicate the number of reset steps. The 6 plots show the naive replacement of the binary32 data type with the simulated one. This means that every single variable containing a binary32 was replaced with a variable of the simulated data type. Plot *1a* shows the exact residue (7.7a) calculated in every iteration using the Dirac matrix and the source vector both in binary64, while plot *1b* shows the norm of the recursively calculated residue (7.7b) (cast from the simulated data type to binary64). The relative residue suffers from round-off accumulation because of the recursive calculation; this is the difference between plots *1a* and *1b*, which is plotted in plot *1c*. Plot *1d* shows the A-orthogonality of the current direction to the last direction, namely the value of $\vec{p}_{i-1}^{\dagger} A \vec{p}_i$. The last 2 plots, *1e* and *1f*, show the values of the amounts α_i and β_i (see equations (7.13) and (7.14)) in every iteration, but only of the data types that converged (`status>0`). The lines in plot *1e* are linearly fitted to the data points ($f(x) = mx + b$). The number range of the slope m is given in the plot legend.

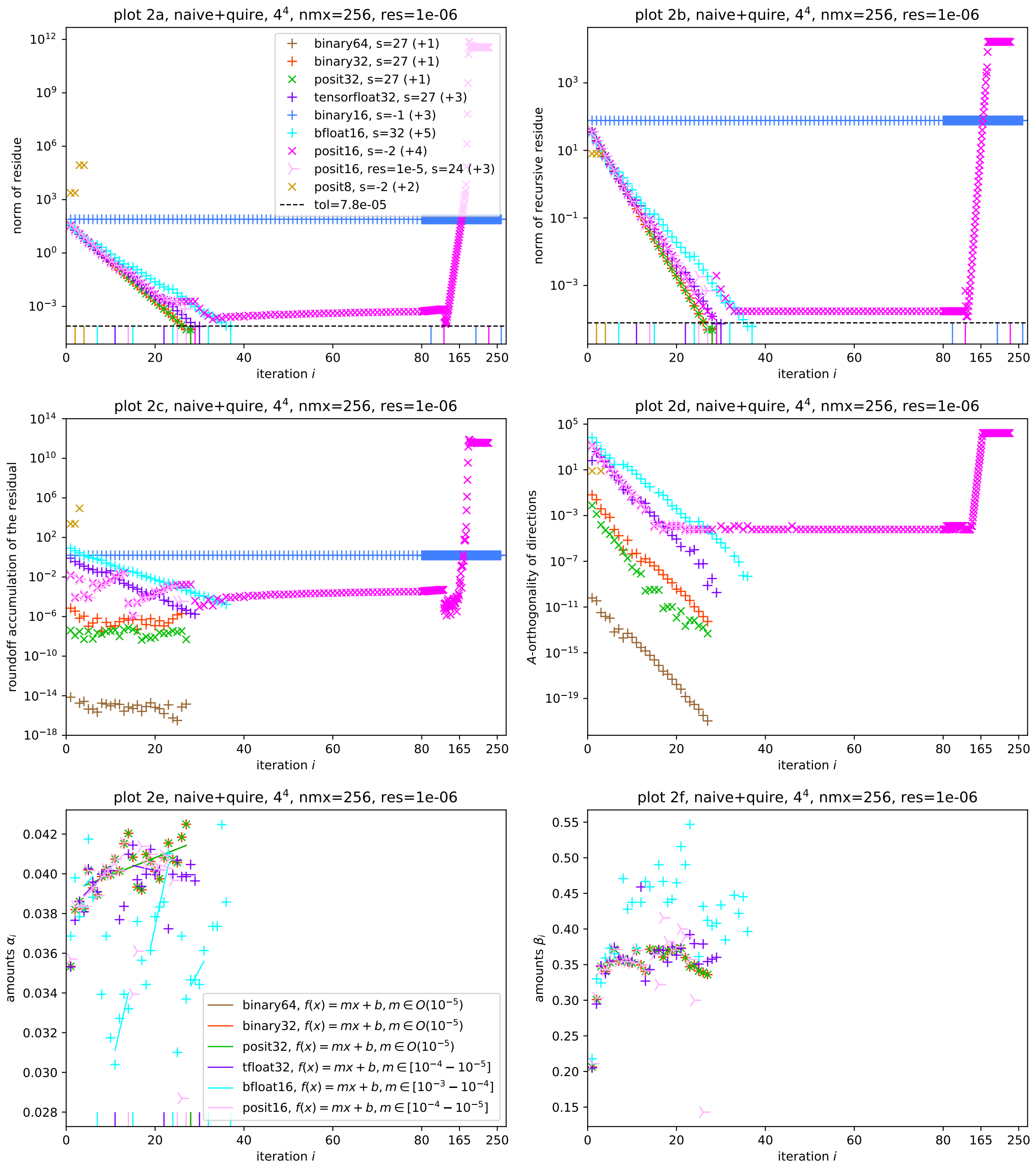

Figure 9: In these plots, the posits made use of quires as their collective variables, the remaining setup was the same as for figure 8, therefore the floating-point data types show exactly the same values, only posits changed their behaviour.

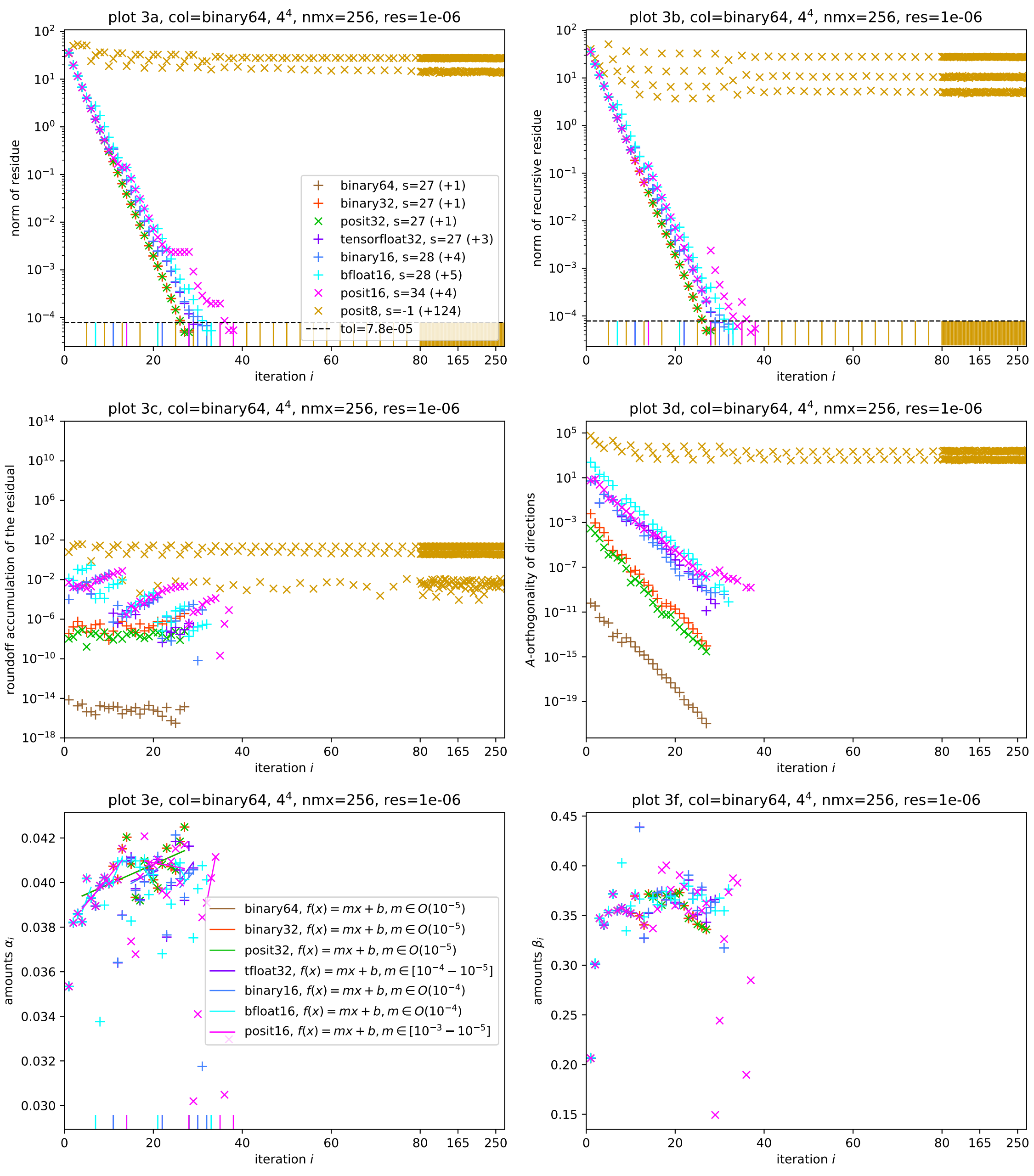

Figure 10: The 6 plots introduce a smarter replacement. All collective variables such as norms were calculated in binary64, such that a data type with a small number range such as binary16 may not over- or underflow when calculating the norm of a vector full of said data type. This replacement resembles the quire for posits. Using this replacement, even heavily reduced data types like binary16 and posit16 converged and threw a result of equal quality as the one simulated with binary64.

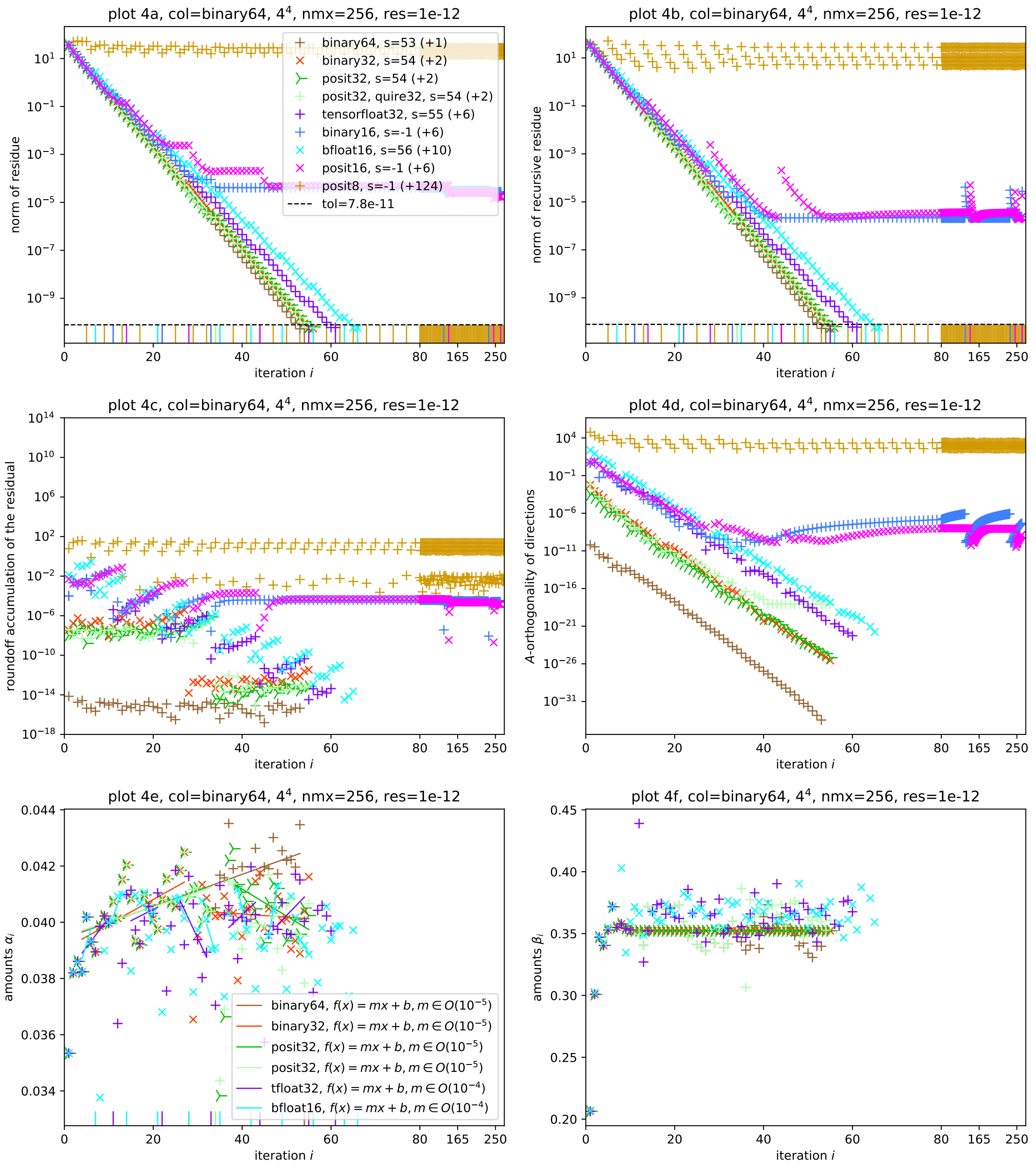

Figure 11: The configuration in this series of plots is equal to figure 10, besides the value of `res` – the desired relative residue of the calculated solution – is set to 10^{-12} instead of 10^{-6}. Notice that 10^{-12} is outside the representable number range of the data types that did not converge; binary16, posit16 and posit8.

would be expected from a recursive calculation. The reason for the small difference between binary32 and posit32 could be that the involved real numbers are closer to representable numbers in posit32 than in binary32. Posits have a larger number density around 1 compared to floats of the same bit-length, and therefore more precision in that regime (see figure 5 for the example of binary16 vs. posit16). Posits also have more numbers because they have no NaNs. Round-off accumulation depends crucially on the precision of the data type; the lower the precision, the higher the round-off accumulation. The difference in A-orthogonality is negligible for posit32 compared to binary32, but clearly surpassed by binary64.

binary16 did not converge (`status=-1`) after the maximal number of `nmx=256` steps. Its footprint is absent in plot *1d*, because it consisted only of NaNs and infinities, causing $\alpha_i = 0$ and $\beta_i = 1$. This implied that $\vec{r}_i = \vec{r}_{i+1}$ and $\vec{p}_{i+1} = \vec{r}_{i+1}$ and therefore $\vec{x}_{i+1} = \vec{x}_i$ and the algorithm stalled. This explains why the residues don't change in plots *1a* and *1b*. The reason for the first infinity was an overflow when calculating the norm of $\vec{b}$ in the very first iteration. This suggests that the limited number range of binary16 might not be enough (at least for a naive replacement), comparing to bfloat16 with the same bit-length, but larger number range that was able to converge, although very slowly.

The behaviour of posit8 is similar to binary16, but without the overflow, because posits do not overflow by definition. Instead, the largest representable number is returned or in case of an underflow the smallest representable number is returned [43]. The algorithm stalled at a value $\|\vec{r}_i\| = 8$ of the recursive residual norm. The largest 8-bit posit number with exponent bits $es = 0$ is $2^6 = 64$, thus the norm squared cannot be greater than 64 and the norm itself cannot be larger than $8 = \sqrt{64}$ (see plot *1b*). This happened in the first step, where the actual residual in binary64 was $\backsim 10^3$. The amounts $|\alpha_i| \ll 1$ in iterative steps are therefore negligibly small causing $\vec{x}_{i+1} \approx \vec{x}_i$. Significant changes in $\vec{x}_i$ will not happen and convergence is unlikely. Notice that posit8 had 256 reset steps, which means that after every step there was a reset step. The steps were caused by the low precision (and thus high machine epsilon) of posit8. The value of `PRECISION_LIMIT` is $100 *$ `MACHINE_EPSILON`, which takes a value of 3.125 for posit8.

Similarly, for posit16, the maximal representable value with $es = 1$ is 268,435,456 whose square root is 16,384 which is reached after 8 steps (see plot *1b*). The actual residual in the 8-th step was $\backsim 10^7$, the algorithm diverged and then stalled. Iterative steps are therefore mostly too small, and convergence is unlikely.

We observe that number range is more important than precision, when naively replacing the data type, but the higher the precision, the faster the convergence and the fewer reset steps are needed.

Figure 9: In figure 9 posits made use of the quire. Therefore, the numbers for the float data types are exactly equal to the ones in figure 8, because floats have no such feature. They are not further discussed.

Comparing plots *1c* and *2c* and looking at posit32, one can see that the round-off accumulation in the residual due to its recursive calculation is slightly better than without using the quire. This makes sense, because quires introduce deferred rounding. The entire run thus involved less rounding. This is exploited in the calculation of norms and matrix-vector products. It results in a somewhat better maintaining of A-orthogonality for the direction vectors.

However, the data points of posit16 bear little resemblance to their previous or later runs. They come much closer to the target residual tolerance than in the last simulation, but the tolerance is still not reached. The tolerance is within the number range of posit16, even so it did not converge. The reason for this is that the smallest representable number in posit16 is 2^{-28}. The quire for posit16 has the same number range, despite the 128 bits in length. Every norm squared of a non-zero vector must be larger or equal to this number, because posit do not underflow. Therefore, the norm is always greater or equal to $\sqrt{2^{-28}} = 2^{-14} \approx 6.1 \cdot 10^{-5}$. The tolerance of $7.8 \cdot 10^{-5}$ – even though larger than that number – is perhaps still too close. Comparing the lightpink values, that are posit16 as well, but the relative residual `res` is set to 10^{-5} instead (the tolerance being one order of magnitude larger), they converged after only `status=24` steps. This suggests that the reason for the strange behaviour lies in the relative residual that was chosen too close to the lowest number above zero of the number regime.

Using the same arguments and analysis, posit8 did not give a meaningful result.

Figure 10: In figure 10, a smarter replacement was done. All reduction variables that have a collective role suffer from overflow. For example, the norm of a vector $\vec{v} \in \mathbb{R}^n$ is

$$\|\vec{v}\| = \sqrt{\sum_{i=1}^{n} v_i^2}.$$

The number below the square root may be much bigger before squaring than after. If we calculate the norm in posit8, the result will be $\|\vec{y}\| \leq 8$. More importantly, when using a data type that overflows such as binary16, the value after squaring might be perfectly fine, but the value under the square root could be outside the range of representable numbers, $\sqrt{\infty} = \infty$ and $\sqrt{0} = 0$. This is avoided if the collective variable is of a data type with larger number range than the underlying data type that is summed over. In figure 10 all reduction variables were of type binary64.

The data of binary64 exhibits no significant alterations. Again comparing binary32 and posit32 with their previous data points, we see that the round-off accumulation of binary32 is a little better, while posit32 looks the same as with the quire, suggesting that when using posits employing the quire is probably sufficient.

Looking at tensorfloat32, it has the same exponent range as binary32, but less precision and it has the same number of mantissa bits as binary16, but at a higher exponent range. Compared to binary16, both data types have the same amount of numbers to be distributed in their respective number range. It is expected to perform worse or equal to binary32, but better or equal to binary16 and bfloat16. Therefore, it's expected to converge in $27 \leq$ `status` ≤ 28 steps, see equation (7.16). This is indeed the case with `status=27` steps. We see that the larger number range compared to binary16 has little effect on the speed of convergence. This is because the number regime is within the binary16 regime, except for reduction variables. This explains as well why tensorfloat32 performed precisely as in the naive replacement, figure 8, but the round-off accumulation is better because of the more precise reduction variables.

The bfloat16 with even less precision but comparable number range of tensorfloat32 converged in `status=28` steps as well, but required two more reset steps, tightening the previous conclusion about speed of convergence.

The most interesting data points are the ones of binary16 and posit16 that both were able to converge in `status=28` and `status=34` steps, respectively. They performed quite similar, even though it would be expected that posit16 would perform slightly better because of the increased number range and larger number density in the relevant number regimes (see figure 5). In plot *3c* the increase of round-off accumulation can be observed for binary16 and posit16 in steps where the real residue changes (where the algorithm makes progress, see for example: steps 1 to 10). Notice that, when the real residue stalls and the recursive residue still (wrongly) decreases, the round-off accumulation will saturate until the orders of magnitude of the two numbers differs too much such that their difference is dominated by the larger number. This can be seen in the data points of posit16 in plot *3a*. It suggests that the precision limit was chosen too low for the data type. Notice that the precision limit is defined to be `100*MACHINE_EPSILON`, where `MACHINE_EPSILON` depends on the data type. The `MACHINE_EPSILON` for the posit data types is quite misleading, because it gives us (by definition) the precision of numbers around 1. This is the regime where posits are most precise, their precision falling off very rapidly when leaving it. Thus for posit16 in the regime 10^{-1} the `MACHINE_EPSILON` is correct (seen at iteration 14), while in the regime 10^{-3} it is chosen to small and we can see a staircase-shape around the reset steps at iterations 28 and 35. Such a stalling of the real residue should be avoided at any cost because the algorithm stalls as well in that case. The `MACHINE_EPSILON` is defined to be the difference between 1 and the lowest number above 1. For floats this definition makes more sense, because their precision does not fall off that rapidly, but for posits which are most precise around 1, this gives a too precise value, not reflecting the real precision of posits in their entire number range correctly. Instead, the machine epsilon should be a function of the number regime, increasing when going far away from 1. This is the reason for the staircase-shaped curve of posit16 in plot *3a*. This phenomenon is even more prominent for posit16 in plot *4a* of figure 11. Posit32 does not have this problem, because its `MACHINE_EPSILON` is sufficient for the number regime used in the algorithm. When demanding lower relative residuals, staircase-shapes should be expected for posit32 as well.

Comparing binary16 with bfloat16 and tensorfloat32, we see again that exponent range is less relevant than precision, but precision determines the amount of reset steps.

Figure 11: Figure 11 shows all the simulated data types using a reduction data type of binary64 as in figure 10, but with a relative residual of 10^{-12} instead. The last row resembles the predicted hierarchy (7.16) particularly well. Notice that 10^{-12} is outside the representable number range of binary16, posit16 and posit8. This means that these data types cannot reach the target tolerance, therefore we didn't expect them to converge. This is indeed the case. Furthermore, we see that binary16 and posit16 both are not able to go below 10^{-5}, meaning the tolerance in the third row was chosen close to the minimum, but still converging tolerance (see discussion of posit16 in figure 9). Both data types make no further considerable progress after step 45. It can be seen by the recursive residue stalling or even increasing – an indicator that the data type has reached its limits.

The comparison between binary32 and posit32 is again of insight. Their difference is subtle. We see that both required the same number of steps. Round-off accumulation and A-orthogonality are again slightly better, making posit32 the overall better 32-bit data type for the problem. The reason for this is due to the higher precision of posits in the relevant number regime. Looking at the lightgreen values, that are posit32 as well, but utilising the quire instead of binary64 as collective reduction variable, we observe the same number of steps to convergence, but round-off accumulation is slightly worse. It might be an unfair comparison, because binary64 as collective variable has more precision, surpassing even the deferred rounding employed by the 512-bit quire for posit32. In plot *4d* the posit32 with quire will not go below some fixed value. The reason for this is the lowest posit32 value with exponent bits `es=2` is 8^{-30} and the norm of a posit32-vector with at least one non-zero component must be greater or equal to the square root, $1.15 \cdot 10^{-18}$. This suggests that when choosing `res` to be smaller than 10^{-18}, we expect posit32 to not converge anymore in analogy to posit16 in the second row.

Since binary16 was able to converge in figure 10, suggesting that the number regime is within binary16, which gives posit32 more precision in that regime over binary32

Finally, we compare the three data types with the same exponent range, but different precisions; binary32, tensorfloat32 and bfloat16. The less precision, the slower the convergence. The price to go from 23 to 10 mantissa bits results in 1 more conjugate gradient step as well as 4 more reset steps. When going further down to 7 mantissa bits again 1 more regular step and 4 more reset steps where needed to finally bring bfloat16 to convergence after `status=56` regular conjugate gradient plus 10 reset steps. Bearing in mind that it uses only 16 bits, this is a remarkable result. It performed way better than its 16-bit competitors.

As seen in plot *4a*, all data types start to converge by the same speed (all slopes are equal). The actual residual of the data type with the lowest precision, namely bfloat16 with 7 mantissa bits, resets first, followed by binary16 and tensorfloat32 which have both 10 mantissa bits. The next one is posit16, because it has more precision than binary16 in the relevant regime, followed by binary32 with 23 mantissa bits and later by posit32, where the same argument as before holds. The curve of binary64 would also reset at some point, but that is outside the scale.

Specially plot *4a* suggests that we can start to calculate in a data type with 16 bits of length until we fall below a constant value (to be determined and depending on the data type), then continuing the calculation in a data type with 32 bit-length until that number regime is exhausted as well, again switching to a 64 bit data type to finish the calculation.

7.3.3 8^4 lattice

To make sure that the previous analysis is consistent, and the physics involved were relevant, the same data was extracted from an 8^4 lattice and some of the plots were remade from the new data, see figure 12. Only the data types binary64, binary32 and binary16 were simulated. The main difference to figures 10 and 11 is that more steps were required to converge, because the Dirac matrix is much larger than before, although only 0.04% of all components were non-zero, compared to 2% in the 4^4 lattice of the previous analysis. In plots *2a* to *2f*, where the relative residue was chosen to be 10^{-12}, we again see the saturation of binary16 marking the lower limit of the data type. After every reset step, a jump in round-off accumulation can be seen, because the residual in the reset step is calculated in higher precision. It is interesting that the round-off accumulation in the final steps of binary16 come remarkably close to those of binary32 (see plot *1c*). A reason for this could be the clustering of reset steps just before convergence, giving accurate results with little round-off, even for less precise data types. Furthermore, we see the speed of convergence does not significantly depend on the precision of the data type, only the amount of reset steps does, thus the less steep slope of binary16. When the lower limit of the data type is reached, the slope becomes zero and the residual shows no striking change anymore. This is where the data type should be switched to one with a larger number range.

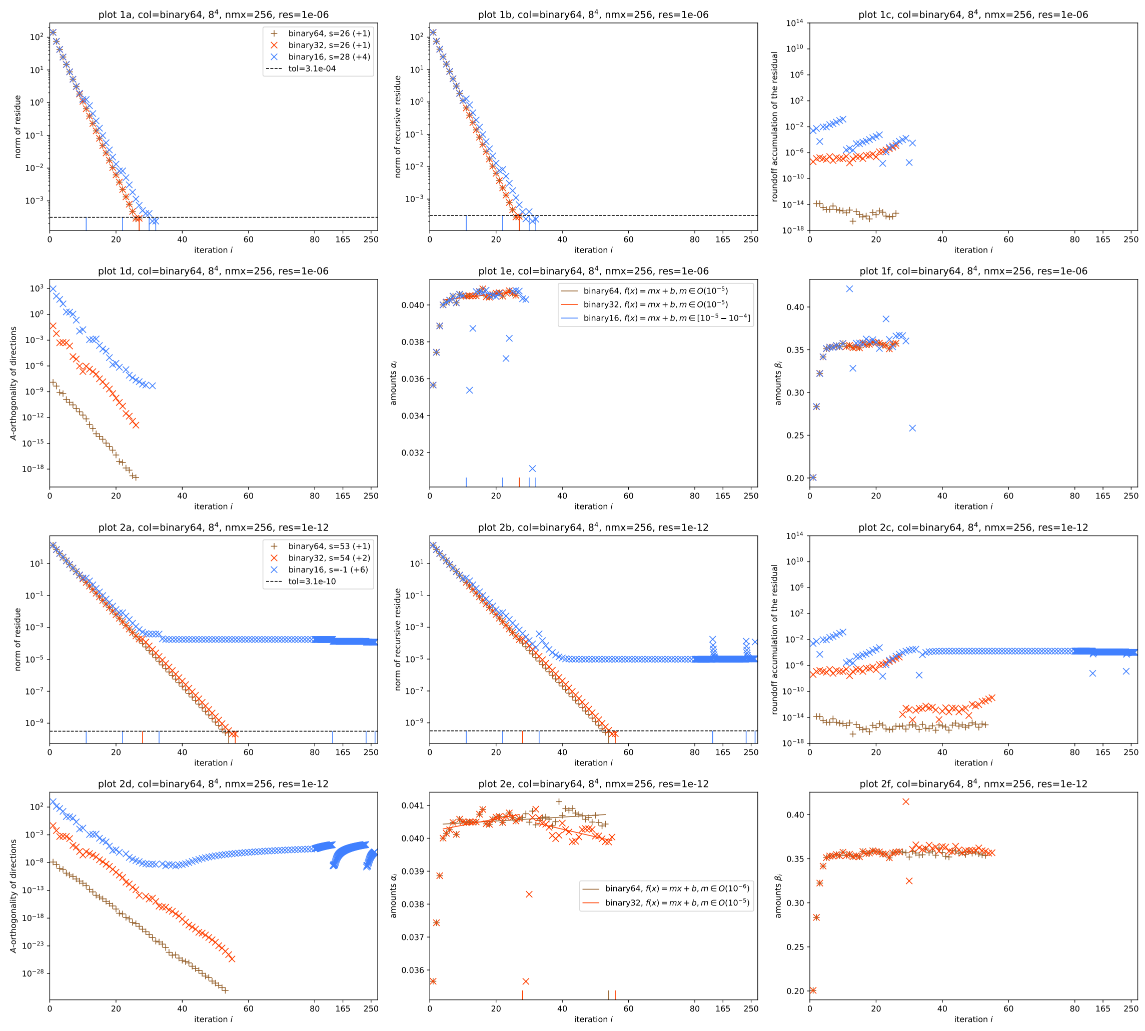

Figure 12: Analogue to figures 10 and 11. This time an 8^4 lattice was used and only the floating-point data types that are available in hardware nowadays were simulated. The *first and second row* use binary64 as collective variable and 10^{-6} was the desired relative residual. The *third and fourth row* have the same setup, but with a relative residual of 10^{-12} instead.

7.3.4 Conclusion

The decision between floats and posits is not trivial. There is currently no hardware available, that has dedicated posit units and posits are not studied as intensive as floats. Furthermore, floats are widespread, well understood and implemented in common hardware.

If one decides to replace binary32 with posits, the most elegant solution would be to naively replace the data type and utilise quires in collective operations. To use binary64 in reduction variables is not recommended, because this would introduce many type conversions between the floating-point and the posit format which is assumed to be expensive. The drawback of this method is that posit16 may only converge if the relative residue is not chosen high enough (see plot *2a* in figure 9).

If the decision goes for floats, which might be the more realistic scenario, then the most elegant solution would be to use reduction variables in binary64. Type conversions between different IEEE floating-point types are not considered to be expensive. The tensorfloat32 compared to binary32 and bfloat16 answers the question how important precision is in the calculation. All of them have the same number of exponent bits and therefore approximately the same number range, but very different precisions. We see that all of them were able to converge in any experiment, but with binary64 as collective variable, the results were closest to each other (see figure 10 plot *3a*). The only relevant difference was in the number of reset steps. If the data type is lower in bit-length, the memory-boundedness suggests that the calculation performs faster, but the trade-off is the amount of (computationally expensive) reset steps that increase with decreased precision. However, the data type for reduction operations should be precise and should have a large number range. Since the number of variables needed in that data type does not scale with the problem size or the number of steps, we can use a data type with large bit-length such as binary64. Comparing the convergence of bfloat16 in the naive case (figure 8 plot *1a*) with the case binary64 collective variables (figure 10 plot *3a*), it can be seen that the algorithm converged 21 steps faster, only because the reduction data type was chosen to be binary64. On the other hand, comparing the performance of binary16 in the two plots, we see that the number range of the reduction data type brought binary16 from no convergence to convergence within `status=35` steps – only marginally slower than binary32.

These arguments make binary64 the best choice for reduction variables. Furthermore, the results show that further reducing the precision still maintains a functional solver.

Proposal 7.1: Mixed Precision Solvers

The above analysis suggests that the calculation of the solution can (at least partially) be conducted in an even less precise data type than binary32. One could for example choose 3 data types with different precision. The algorithm can be started using the least precise one. If the tolerance hits a certain value at the boundaries of the data type, the algorithm switches to the next higher one. The calculation is continued in that data type until the tolerance reaches the limits of the new data type. Again, the data type is switched to the next higher one[a]. Calculating in mixed precision is not dependent on the algorithm itself and can therefore be applied to every iterative solver.
Algorithm 1 shows an example implementation of such a mixed precision calculation. The array d consists of all available data types participating in the calculation in ascending order, meaning the least precise data type comes first. The function $solve()$ performs the underlying algorithm (for example conjugate gradient) in the data type given by its arguments. It expects at least a starting vector $\vec{x_0}$ and a tolerance and returns the status[b], the calculated solution and the residual up to the given tolerance. Reduction variables within $solve()$ are calculated in a datatype with high precision.

[a]One obvious choice could be $d = \{\text{binary16}, \text{binary32}, \text{binary64}\}$. When the algorithm is started in binary16 and a tolerance of $\approx 10^{-4}$ is reached, the algorithm continues in binary32, the limit of which is at a tolerance of $\approx 10^{-30}$. A continuing calculation would then be performed in binary64.

[b]See section 7.2

Proposal 7.2: Approximating the amounts α_i

Looking at plot *4e* of figure 11, where the amounts α_i are plotted for every iteration, we see that after every reset step the amounts need $2-3$ steps to reach a value that is not changing very much for future iterations. This becomes apparent when looking at the fitting lines. The values of the α_i are in the range 10^{-1} and the slopes m of the fitting lines are in the range 10^{-4} - 10^{-5}, suggesting that the value of α_i is not changing from iteration to iteration when only looking at $2-3$ significant decimal digits.
A possibility to reduce computational cost in each iteration could be to approximate the values of future α_i to be constant. The less precise the data type, the larger the change in α_i. The large error in α_i of bfloat16 in all plots suggests that the algorithm is not sensible to errors in α_i. Therefore, it can be expected that the results should not change significantly with an approximated value of α_i.

- Advantage: The residuals can be calculated using $\vec{b} - A\vec{x}$, instead of recursively. This implies less round-off accumulation.
- Advantage: Only one matrix-vector multiplication per iteration.

Algorithm 1: Pseudo-code for an iterative algorithm in mixed precision.

```
    input: desired norm of relative residual $rn$
    input: array of data types in $\{d\}_{k=0}^{N}$
    input: iterative algorithm $solve()$
1   $\vec{x_0}, \vec{r_0}, \ldots \leftarrow$ initial guess, $\ldots$;
2   $\vec{x}, \vec{r} \leftarrow \vec{x_0}, \vec{r_0}$;
3   status $\leftarrow 0$;
4   for $k \leftarrow 0, 1$ to $N$ do
5       convert all variables to data type $d[k]$;
6       $tol \leftarrow \frac{1}{\|\vec{r_0}\|} max(rn,$ MACHINE_EPSILON of $d[k])$;
7       substatus, $\vec{x}, \vec{r}, \ldots \leftarrow solve(tol, \vec{x}, \ldots)$;
8       if $substatus > 0$ then
9           status $\leftarrow$ status + substatus;
10      if $\|\vec{r}\| < rn$ then
11          return status, $\vec{x}$; // success
12  end
13  status $\leftarrow -3$;
14  return status, $\vec{x_0}$; // the algorithm failed
```

- Disadvantage: Since the α_i are just approximated, the number of required iterations may increase.
- Disadvantage: The Dirac-operator D must be given in the form of $A = D^\dagger D$ as *one* operator, else the algorithm still consists of 2 matrix-vector multiplications per iteration and the benefit is non-existent. Also, $D^\dagger D$ is less sparse than D.

The results of simulations with approximated values for the α_i can be observed in plot series 13 and 14. The value was approximated based on previous values. The first 5 steps where skipped (thus the algorithm performed natively). In step number 5, the last 3 values of α_i where averaged. In the following steps the constant value calculated in step 5 was reused. After every reset step, the value of α_i had to be recalculated using the above procedure. Therefore, a data type such as bfloat16 that has reset steps after approximately every 7th regular step, will benefit in only 2 steps per reset step. This is little difference to native runs of data types with high precision.
The calculation became more sensible to the number range of the data type. This can be seen in all plots when looking at binary16 that was not able to converge anymore, although by a very small amount. Tensorfloat32 on the other hand performed similar to the regular rounds, it was expected that it needs slightly more iterations. When going with this strategy, it is therefore advisable to perform more regular CG-steps when coming closer to the boundaries of the data type. One possible solution would be to choose a higher machine epsilon close to the boundaries, forcing the algorithm to perform more reset steps, in turn causing more regular CG-steps and recalculations of α_i.
Notice that with larger lattice sizes, the approximation of the amounts has less error (see plots *1e* and *2e* in figures 13 and 14) and the algorithm is thus more stable.

8 SAP preconditioned GCR

To study spontaneous chiral symmetry breaking we need large lattices and small masses. Conjugate gradient and related algorithms become inefficient on such lattices. The lattice Dirac-operator gets increasingly ill-conditioned in such setups. The solver presented in the following is called SAP+GCR. It makes use of a multiplicative Schwarz Alternating Procedure (SAP) as preconditioner for a flexible Generalized Conjugate Residual (GCR) solver. Parallelizing the algorithm is straight

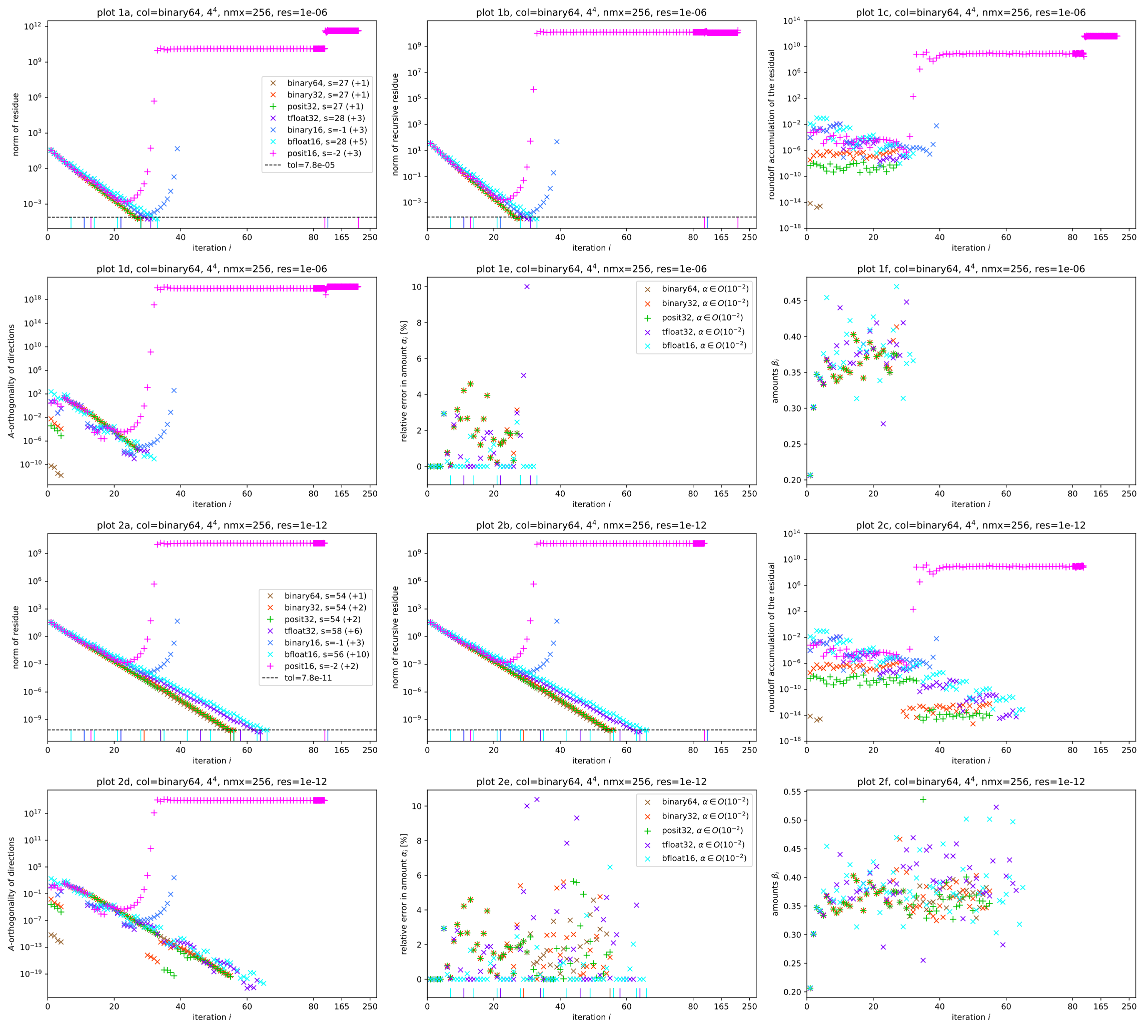

Figure 13: Plots *1a* to *1f* shows the convergence analysis of a conjugate gradient run with a 4^4 lattice, relative residual 10^{-6} and approximated values of α_i. In plots *2a* to *2f* the residual was chosen to be 10^{-12}. Plots *1e* and *2e* show the relative error in the approximated α_i compared to the real α_i.

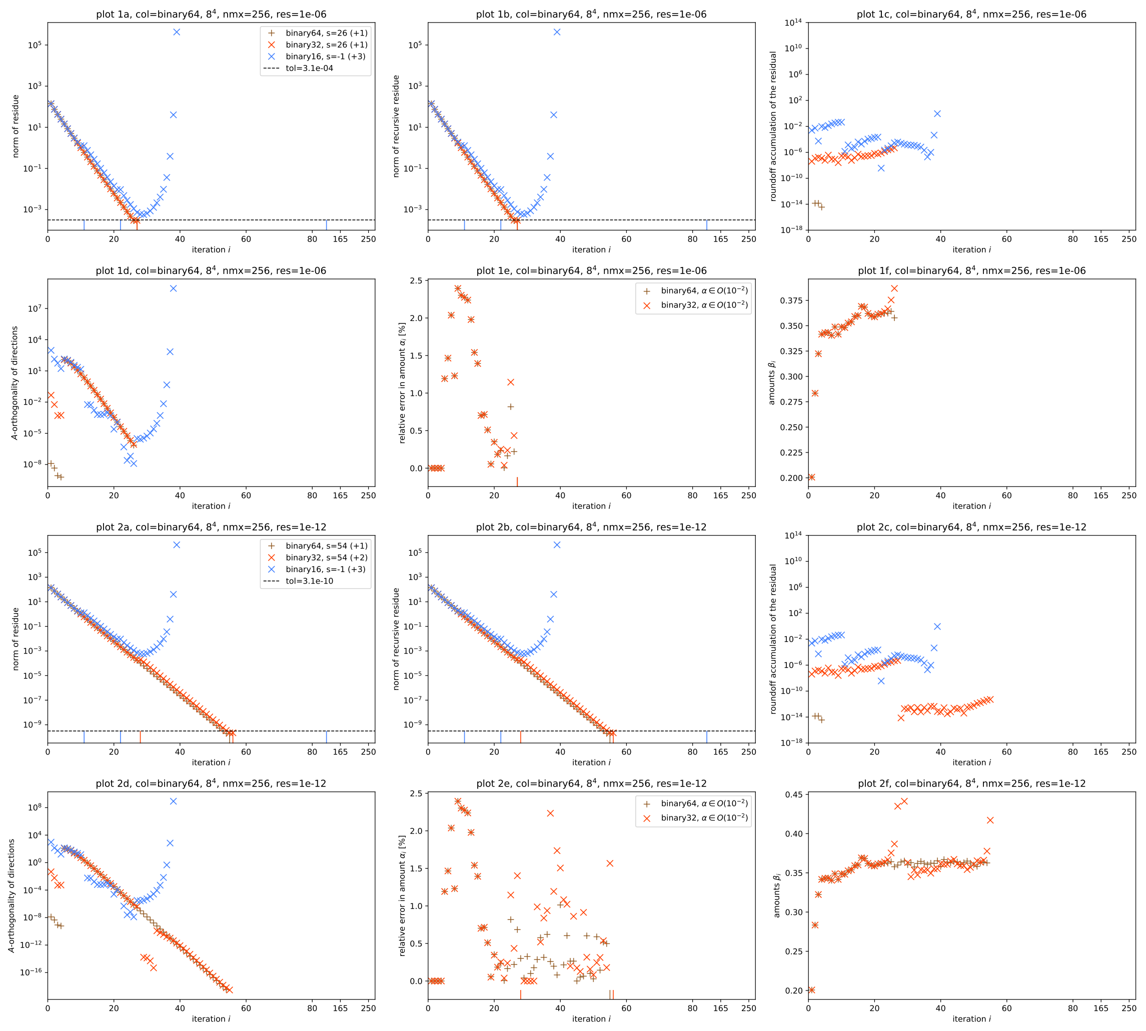

Figure 14: The same setup as figure 13, but with an 8^4 lattice.

forward, because it consists of decomposing the large problem into many smaller independent ones.

8.1 Even-Odd Preconditioning

Preconditioning in general, when employed in lattice QCD, is expected to have significant impact on the number of iterations of a solver. One way of preconditioning $D\psi = \eta$ on a lattice is

$$LDR\psi' = L\eta,$$

with $\psi = R^{-1}\psi'$ and L, R chosen wisely such that LDR is well conditioned. If $L = \mathbb{I}$, it is called ***right preconditioning***, if $R = \mathbb{I}$ it is called ***left preconditioning***. If the Dirac-matrix involves only nearest-neighbour interactions, it is possible to split the lattice into even and odd sites[46] [47]. If the sites are ordered such that the even sites come first[48],

$$D = \begin{pmatrix} D_{ee} & D_{eo} \\ D_{oe} & D_{oo} \end{pmatrix}, \qquad \psi = \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix}$$

[46]It is therefore similar to a domain decomposition method, see section 8.2.

[47]Even lattice points are the ones where the sum of the global Cartesian coordinates $(n_0 + n_1 + n_2 + n_3)$ in units of the lattice spacing a is even, notice $x_\mu = an_\mu$, where $\mu = 0, 1, \ldots, D$.

[48]This is indeed the case in openQ*D (see `main/README.global`) in [17].

D_{ee} (D_{oo}) consists of the interactions of the even (odd) sites among themselves, while D_{eo} and D_{oe} consider the interactions of even with odd sites. ψ_e and ψ_o contain the values for even and odd lattice sites of the spinor.

Using specific forms of L and R, D can be brought into a block-diagonal form, namely

$$L = \begin{pmatrix} 1 & -D_{eo}D_{oo}^{-1}D_{oe} \\ 0 & 1 \end{pmatrix} \qquad \text{and} \qquad R = \begin{pmatrix} 1 & 0 \\ -D_{oo}^{-1}D_{oe} & 1 \end{pmatrix}.$$

After a bit of algebra,

$$LDR = \begin{pmatrix} \hat{D} & 0 \\ 0 & D_{oo} \end{pmatrix}, \qquad \text{with} \qquad \hat{D} = D_{ee} - D_{eo}D_{oo}^{-1}D_{oe}.$$

This specific preconditioning reduces the amount of iterative steps needed by a factor of 2 approximately, because D_{oo} and $\hat{D}$ are matrices of half the dimension of D. The inversion of D_{oo} is simple, because with only nearest-neighbour-interactions the odd sites do not interact among themselves, only with even sites. Thus D_{oo} exhibits block-diagonal form (all blocks are 6×6). Using

$$D\psi = \eta \implies \begin{pmatrix} D_{ee} & D_{eo} \\ D_{oe} & D_{oo} \end{pmatrix} \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix} = \begin{pmatrix} D_{ee}\psi_e + D_{eo}\psi_o \\ D_{oe}\psi_e + D_{oo}\psi_o \end{pmatrix} = \begin{pmatrix} \eta_e \\ \eta_o \end{pmatrix},$$

we can write the preconditioned form, where only the reduced system with even lattice sites has to be solved to determine ψ_e,

$$\begin{aligned} \hat{D}\psi_e &= D_{ee}\psi_e - D_{eo}D_{oo}^{-1}D_{oe}\psi_e \\ &= (\eta_e - D_{eo}\psi_o) - D_{eo}D_{oo}^{-1}(\eta_o - D_{oo}\psi_o) \\ &= \eta_e - D_{eo}D_{oo}^{-1}\eta_o, \end{aligned}$$

because ψ_o follows from the solution ψ_e via

$$\psi_o = D_{oo}^{-1}(\eta_o - D_{oe}\psi_e).$$

Remark. Note that it works only because D_{ee} is diagonal in space for Wilson (and Wilson-twisted) fermions.

Remark. This preconditioning method does not allow multisolvers[49].

8.2 Schwarz Alternating Procedure

Domain decomposition is a way to partition the large system into (possibly many) smaller sub-problems with regularly updated boundary conditions coming from solutions of neighbouring sub-problems. They fit well into the notion of parallel processing because the sub-problem can be chosen to be contained in one single rank. The full lattice is split into sub-lattices called ***local lattice***. Each rank has its own local lattice, the size of which is determined at compilation time. The full lattice consists of the ensemble of all local lattices arranged in a grid. It is therefore advisable to choose the size of the decomposed sub-domains as divisor of the local lattice size such that one or more blocks fit into one rank. These sub-problems can then be solved using an iterative solving method.

The idea behind Schwarz Alternating Procedure is to loop through all blocks Ω_i and solve the smaller sub-problem using boundary conditions given from the most recent global solution (see figure 15). If the original problem only includes nearest-neighbour interactions, the solution of a block Ω_i depends only on that block and its exterior boundary points, which are the adjacent points on the neighbouring blocks with opposite color. For example, the solution of the sub-problem involving Ω_6,

[49] Algorithms which solve for multiple masses in one shot, for example MSCG.

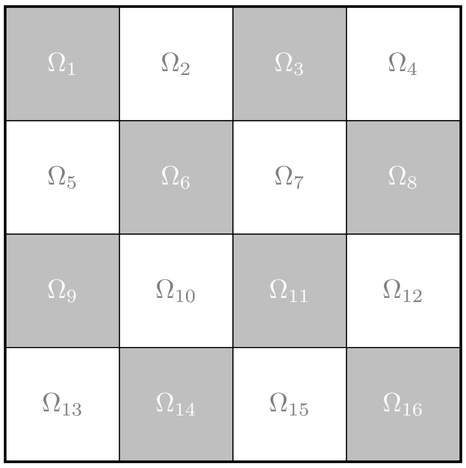

Figure 15: A $d = 2$ dimensional example of a decomposition of a lattice $\Omega = \bigcup_{i=1}^{n} \Omega_i$ into $n = 16$ domains named Ω_i. Notice such a decomposition can always be colored like a chessboard.

depends only on the solutions of Ω_2, Ω_5, Ω_7 and Ω_{10}[50]. Therefore, all grey (white) sub-problems can be solved simultaneously, with the most recent boundary conditions obtained from the white (grey) domains. Solving all grey, followed by all white sub-problems is called a ***Schwarz-cycle*** and is considered one iteration in SAP. Each sub-problem can be solved with any desired solver separately, again applying some preconditioning[51].

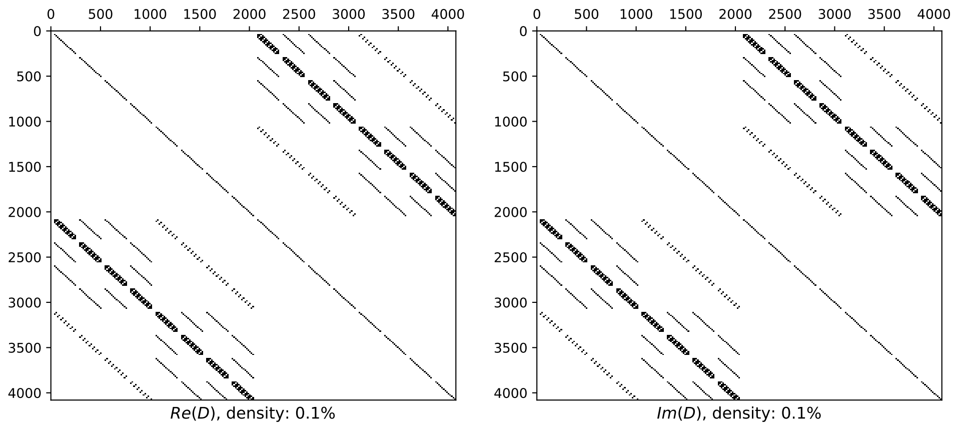

Figure 16: An example plot of a Dirac-matrix of an 8^4-lattice with SF-boundary conditions. The operator is already in a shape, where the even lattice points come first, followed by the odd lattice points. Every pixel consists of 192×192 real numbers. If the average over that numbers is non-zero the pixel is drawn black, else the pixel is drawn white. The density gives the overall percentage of non-zero values.

Whereas the division into domains on the lattice is straightforward, the representation of the Dirac-operator as a sparse matrix and its decomposition is not. Looking at an actual example of a Dirac-operator written as a matrix (see figure 16), one observes a lot of structure: while on the diagonal we find the operators restricted to the black and white blocks, the first and the third quadrant describe the operators restricted to the interior and exterior boundaries. The operator restricted to the exterior boundaries of the union of all black (white) blocks is denoted by $D_{\partial b}$ ($D_{\partial w}$). The decomposition into $2n$ domains (n grey and n white blocks) can be translated as seen

[50]It depends on all other sub-problems as well, but indirectly.

[51]Using even-odd preconditioning is perfectly fine with D replaced by the restricted Dirac-operator D_i acting only on the points in Ω_i.

in figure 17. Notice that the restricted operators D_i are well-conditioned because they have block diagonal form. The blocked problem would then look like, $i = 1, \ldots, 2n$,

$$D_i \psi_i = \eta_i.$$

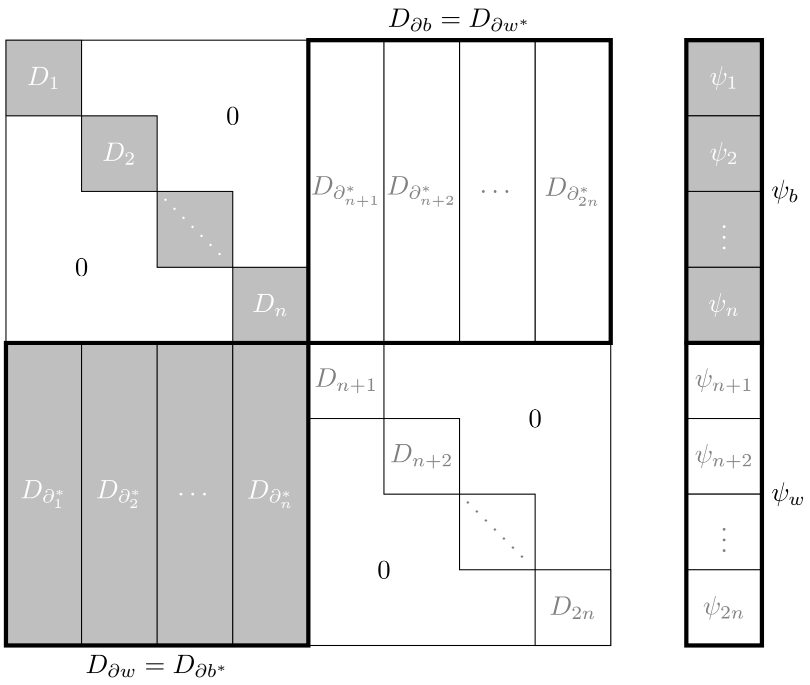

Figure 17: Schematic of the Dirac-operator in terms of a large sparse matrix. If the components of the black blocks are arranged such that they appear first, then the decomposition from figure 15 can be translated into a matrix with blocks as in the picture. D_i describes the Dirac-operator restricted to block i and $D_{\partial b}$ ($D_{\partial w}$) is the Dirac-operator restricted to the external boundaries of the black (white) blocks. The color external boundary operators can be decomposed into external boundary operators of the i-th block, $D_{\partial^*_i}$. The right side describes a vector decomposed into the same $2n$ domains $\psi_1, \ldots, \psi_{2n}$. The upper half corresponds to the black blocks and the lower half to the white blocks.

8.3 SAP as a Preconditioner

The multiplicative Schwarz Alternating Procedure is a domain decomposition method coming from the theory of partial differential equations. It can be applied in the form of a right preconditioner M^{-1} making the preconditioned system

$$M^{-1} A \vec{x} = M^{-1} \vec{b} \tag{8.1}$$

to be solved in very few steps, if M^{-1} is a good approximation for A^{-1}. The preconditioning matrix M^{-1} is never explicitly available during the calculation, such as it is the case in even-odd preconditioning which can also be applied in advance. To solve the preconditioned equation (8.1) using an iterative Krylov subspace method[52], the algorithm must be able to apply M^{-1} and $M^{-1}A$ to an arbitrary vector $\vec{v}$. If it is possible to implement such operations on multiple ranks in an efficient way and if the preconditioner makes $M^{-1}A$ well-conditioned, we reached the goal. Obviously, an application of M^{-1} should be possible without involving A^{-1}. The actions of operators M^{-1} and $M^{-1}A$ on a vector $\vec{v}$ are assembled using a multiplicative Schwarz Alternating Procedure, where

[52]such as Conjugate Gradient, Generalized Minimum Residual, Minimal Residual, Biconjugate Gradient Stabilized or Generalized Conjugate Residual.

the blocks are treated by some fixed number of Minimal Residual (MR) steps[53]. The blocks need not to be solved to a certain precision, because the procedure is only used as a preconditioner approximating the solution.

In openQ*D the `SAP_GCR` solver is implemented as follows: The large problem is solved using a flexible GCR solver, that in each of its `nmx` steps uses a different preconditioner. The preconditioner is given by `ncy` steps of the Schwarz Alternating Procedure applied to the current solution vector. Each SAP cycle involves approximately solving all grey blocks followed by all white blocks on the entire lattice, each with `nmr` steps of the MR method using even-odd preconditioning (`isolv=1`) or not (`isolv=0`).

Proposal 8.1: Performing MR steps on the GPU

The preconditioning procedure involves `mnr` MR steps to be taken on each block in each Schwarz-cycle to approximate a solution to the block problem. Since blocks of the same color are independent of each other and the Dirac-operator acting only on a specific block involves no communication whatsoever, we can conclude that the procedure of solving a sub-problem is *local* to the block and self-contained in the sense that it can be solved independently and without MPI communication among ranks. This could be a very handy starting point when going towards GPU-utilisation.
Once the source vector and the restricted Dirac-operator are transferred to the GPU (both stay constant during the solving process), the problem can be solved on the GPU without involving any communication with other ranks or GPUs. This can be beneficial, because of the following argument: the local lattice of one single rank, can be subdivided into multiple blocks as well (imagine figure 15 being the local lattice). The actual implementation solves the grey (white) blocks in a local lattice sequentially[a]. Since all the grey (white) problems within the local lattice can be solved simultaneously, the code does not exploit the full concurrency potential of the procedure. Solving the sub-problems on the GPU, one could launch MR solvers on all grey blocks simultaneously followed by all white blocks. The MR solver can be called in mixed or even reduced precision.
For a specific implementation of the GPU-solver, one possibility is to encode the restricted Dirac-operator in one of the sparse matrix formats (for example CSR) and use already existing libraries (for example [44] for CUDA) for an application to a spinor. The results in section 8.6 are obtained using such an approach.
Comparing the implementation of the Dirac-operator in QUDA (see ref. [45]), it is advisable to not rely on such generic libraries, because they ignore further symmetries and structure of the operator. The problem lies mostly in the memory-boundedness of the procedure.

[a]By iterating over the blocks, see `sap()` at line 717ff in `modules/sap/sap.c` in [17].

8.4 Generalised Conjugate Residual

The choice for the outer solver goes to GCR, because inexact preconditioning is then possible without affecting the correctness of the solution [46, 47].

We wish to solve (7.1) if A is not Hermitian. Comparing to the conjugate gradient algorithm, we minimise the residual $\vec{r}$ of the solution $\vec{x}$, using the ***quadratic form***

$$\begin{aligned} f(\vec{x}) &= \frac{1}{2}\left(\vec{b} - A\vec{x}\right)^{\dagger}\left(\vec{b} - A\vec{x}\right) + c \\ &= \frac{1}{2}\left\|\vec{b} - A\vec{x}\right\|^2 + c \\ &= \frac{1}{2}\|\vec{r}\|^2 + c, \end{aligned}$$

where $c \in \mathbb{C}$. When taking the derivative of this function with respect to $\vec{x}$, we find that

$$f'(\vec{x}) = A^{\dagger}A\vec{x} - A^{\dagger}\vec{b}.$$

[53]Determined by the value of `mnr` in the solver section of the input file.

Lemma 8.1 (Uniqueness of the solution)**.** *The solution $\vec{x}$ in equation (7.1) is unique and the global minimum of $f(\vec{x})$, if A is non-singular.*

Proof. Let us rewrite $f(\vec{p})$ at an arbitrary point $\vec{p} \in \mathbb{C}$ in terms of the solution vector $\vec{x}$,

$$\begin{aligned}
f(\vec{p}) &= \frac{1}{2}\left(\vec{b} - A\vec{p}\right)^\dagger \left(\vec{b} - A\vec{p}\right) + c + f(\vec{x}) - f(\vec{x}) \\
&= f(\vec{x}) + \frac{1}{2}\vec{p}^\dagger(A^\dagger A)\vec{p} - \frac{1}{2}(A\vec{p})^\dagger\vec{b} - \frac{1}{2}\vec{b}^\dagger(A\vec{p}) + \frac{1}{2}\vec{b}^\dagger\vec{b} \\
&= f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger(A^\dagger A)(\vec{p} - \vec{x}) + \frac{1}{2}(A\vec{p})^\dagger(A\vec{x}) + \frac{1}{2}(A\vec{x})^\dagger(A\vec{p}) - \frac{1}{2}(A\vec{x})^\dagger(A\vec{x}) \\
&\quad - \frac{1}{2}(A\vec{p})^\dagger\vec{b} - \frac{1}{2}\vec{b}^\dagger(A\vec{p}) + \frac{1}{2}\vec{b}^\dagger\vec{b} \\
&= f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger(A^\dagger A)(\vec{p} - \vec{x}),
\end{aligned}$$

where to obtain the last line, $A\vec{x} = \vec{b}$ is used, thus the term simplified.
In the new form of $f(\vec{p})$, one can directly see that, $\vec{x}$ must minimise the function:

$$\begin{aligned}
f(\vec{p}) &= f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger(A^\dagger A)(\vec{p} - \vec{x}) \\
&= f(\vec{x}) + \frac{1}{2}\underbrace{\|A(\vec{p} - \vec{x})\|^2}_{> 0 \text{ for } \vec{p} \neq \vec{x}}.
\end{aligned} \tag{8.2}$$

Therefore $\vec{x}$ is the global unique minimum if A is non-singular.

□

Remark. Notice the similarity of the above equation (8.2) to the analogue of the conjugate gradient algorithm (7.2). The only difference is the substitution of $A \longmapsto A^\dagger A$. It is therefore advisable in the derivation of an algorithm to require the directions $\vec{p}_i$ to be $A^\dagger A$-orthogonal instead of A-orthogonal.

In the same manner as in the derivation of the method of conjugate gradient, we impose an iterative ***step equation***,

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{p}_i,$$

again with ***directions*** $\vec{p}_i$ and ***amounts*** α_i that have to be determined. The recursively calculated ***residual*** has again the same formula

$$\vec{r}_{i+1} = \vec{r}_i - \alpha_i A\vec{p}_i.$$

Imposing $A^\dagger A$-orthogonality instead of regular A-orthogonality between error $\vec{e}_{i+1}$ and direction $\vec{p}_i$,

$$\begin{aligned}
0 &\overset{!}{=} \vec{e}_{i+1}^\dagger(A^\dagger A)\vec{p}_i \\
&= (\vec{e}_i + \alpha_i\vec{p}_i)^\dagger A^\dagger A\vec{p}_i
\end{aligned}$$

gives an expression for the amounts α_i. The above equation is equivalent to imposing A-orthogonality, $0 = \vec{r}_{i+1}^\dagger A\vec{p}_i$. However, we find (compare with equation (7.9))

$$\alpha_i = \frac{\vec{r}_i^\dagger(A\vec{p}_i)}{\vec{p}_i^\dagger(A^\dagger A)\vec{p}_i} = \frac{\vec{r}_i^\dagger(A\vec{p}_i)}{\|A\vec{p}_i\|^2}.$$

The GCR algorithm does store all previous direction $\vec{p}_i$ as well as $A\vec{p}_i$ in contrast to conjugate gradient. Thus, the derivation changes slightly. Let's continue with the determination of the directions using ***Gram-Schmidt orthogonalization*** by imposing $A^\dagger A$-orthogonality instead of A-orthogonality and without imposing all previous β_{ij} to be zero (see definition 7.4). Likewise, we set $\vec{u}_i = \vec{r}_i$ and find

$$\begin{aligned}\vec{p}_0 &= \vec{r}_0,\\ \vec{p}_{i+1} &= \vec{r}_{i+1} + \sum_{j=0}^{i} \beta_{ij}\vec{p}_j,\end{aligned}$$

with

$$\beta ij = -\frac{\vec{r}_{i+1}^\dagger A^\dagger A\vec{p}_j}{\vec{p}_j^\dagger A^\dagger A\vec{p}_j} = -\frac{(A\vec{r}_{i+1})^\dagger (A\vec{p}_j)}{\|A\vec{p}_j\|^2}.$$

Using the above equations, we find the final form of the ***Generalised Conjugate Residuals Method***.

Definition 8.1 (Generalised Conjugate Residuals Method)**.** *The iteration step equation of the* ***Generalised Conjugate Residuals Method*** *in defined as*

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{p}_i, \tag{8.3}$$

with

$$\begin{aligned}\vec{r}_{i+1} &= \vec{r}_i - \alpha_i A\vec{p}_i, & \alpha_i &= \frac{\vec{r}_i^\dagger (A\vec{p}_i)}{\|A\vec{p}_i\|^2}, & (8.4)\\ \vec{p}_{i+1} &= \vec{r}_{i+1} + \sum_{j=0}^{i}\beta_{ij}\vec{p}_j, & \beta ij &= -\frac{(A\vec{r}_{i+1})^\dagger (A\vec{p}_j)}{\|A\vec{p}_j\|^2}, & (8.5)\end{aligned}$$

and initial starting vectors

$$\begin{aligned}\vec{x}_0 &= \textit{arbitrary starting point},\\ \vec{p}_0 &= \vec{r}_0 = \vec{b} - A\vec{x}_0.\end{aligned}$$

There are some remarks to note about the GCR method.

Remark. After calculating $\vec{r}_{i+1}$ and $A\vec{r}_{i+1}$, we can recursively determine $A\vec{p}_{i+1}$ via

$$A\vec{p}_{i+1} = A\vec{r}_{i+1} + \sum_{j=0}^{i}\beta_{ij}A\vec{p}_j. \tag{8.6}$$

This limits the number of matrix-vector products to one per iteration.

Remark. All previous $\vec{p}_i$ and $A\vec{p}_i$ need to be stored in memory in order to construct the next $\vec{p}_{i+1}$ and $A\vec{p}_{i+1}$.

Remark. Compared to the conjugate gradient algorithm, we imposed $A^\dagger A$-orthogonality of the directions $\vec{p}_i$ instead of A-orthogonality as well as A-orthogonality of $\vec{r}_{i+1}$ and $\vec{p}_i$ instead of regular orthogonality. A vanishing of all previous β_{ij} on the other hand was not imposed, leading to the sum in the step equation for $\vec{p}_{i+1}$.

Algorithm 2: Pseudo-code for the GCR recursion.

```
1  $\rho_0 = \eta$ ;
2  for $k \leftarrow 0, 1, 2$ to $n_{kv}$ do
3      $\phi_k = M_{sap}\rho_k$ ;
4      $\chi_k = D\phi_k$ ;
5      for $l \leftarrow 0$ to $k-1$ do
6          $a_{lk} = (\chi_l, \chi_k)$ ;
7          $\chi_k = \chi_k - a_{lk}\chi_l$ ;
8      end
9      $b_k = \|\chi_k\|$ ;
10     $\chi_k = \frac{\chi_k}{b_k}$ ;
11     $c_k = (\chi_k, \rho_k)$ ;
12     $\rho_{k+1} = \rho_k - c_k\chi_k$ ;
13 end
```

8.5 GCR in openQ*D

The actual implementation of the GCR algorithm in openQ*D is quite different[54], but equivalent to definition 8.1 (see lemma 8.2). Ref. [46] explains the implementation of the algorithm in detail. The main GCR-loop looks as in algorithm 2 (see figure 3 in [46])

In algorithm 2, M_{sap} is the SAP preconditioner, that might depend on the iteration number k as well, making the algorithm flexible. D is the Dirac-operator and ρ_k the residual in the k-th step. The algorithm does not include an update of the solution vector ψ_{k+1}, instead this is done after n_{kv} iterations all at once,

$$\psi_{k+1} = \sum_{l=0}^{k} \alpha'_l \phi_k. \tag{8.7}$$

Lemma 8.2. *The iterative algorithm from definition 8.1 is equivalent to algorithm 2 when setting the preconditioning operator $M_{sap} = \mathbb{I}$, the Dirac-matrix $D = A$, the source vector $\eta = \vec{b}$ and the solution vectors $\psi_k = \vec{x}_k$.*

Proof. Noticing that the residual $\rho_k = \vec{r}_k$ from line 12 in algorithm 2 and in definition 8.1 must be identical, we find that χ_k must be proportional to $A\vec{p}_k$. Before the normalisation in line 10, we have $\chi_k = A\vec{p}_k$. The $b_k = \|\chi_k\|$ are set before normalisation of χ_k, therefore $b_k = \|\chi_k\| = \|A\vec{p}_k\|$. Using this we find $a_{lk} = (\chi_l, D\rho_k)$ and since $l < k$ the χ_l are normalised, thus $\chi_l = b_l A\vec{p}_l$ after line 10. Thus $a_{lk} = (A\vec{p}_l, D\rho_k)/b_l = -\beta_{k-1,l}\|A\vec{p}_l\|$. Finally, the c_k are defined after normalisation of the χ_k, therefore they evaluate to $c_k = (\chi_k, \rho_k) = (A\vec{p}_k, \vec{r}_k)/b_k = \alpha_k\|A\vec{p}_k\|$. Using these substitutions, we find the same formulas as in definition 8.1, except for the step equation.

The main difference between the step equations (8.3) and (8.7) is in the former the solution $\vec{x}_{i+1}$ is spanned by the direction vectors $\vec{p}_i$, whereas in the latter it is spanned by the residuals $\rho_i = \vec{r}_i$. This is not a problem since both sets of vectors span the same space, but the amounts α'_l in equation (8.7) differ heavily from the amounts α_i in equation (8.4).

To determine the amounts α'_l in terms of α_i and β_{ij}, we notice equation (8.6),

$$A\vec{p}_i = A\vec{r}_i + \sum_{j=0}^{i-1} \beta_{i-1,j} A\vec{p}_j \iff b_i\chi_i = D\rho_i - \sum_{j=0}^{i-1} a_{ji}\chi_j \tag{8.8}$$

and the fact that

$$\rho_{k+1} = \eta - \sum_{l=0}^{k} c_l\chi_l. \tag{8.9}$$

[54] Called GMRES recursive (GMRESR) algorithm [48], see `fgcr()` in `modules/linsolv/fgcr.c` lines 212ff in [17].

But also

$$\begin{aligned}
\rho_{k+1} &= \eta - D\psi_{k+1} \\
&= \eta - \sum_{l=0}^{k} \alpha'_l D\rho_k \\
&= \eta - \sum_{l=0}^{k} \alpha'_l \left[b_k\chi_k + \sum_{j=0}^{k-1} a_{jk}\chi_j \right],
\end{aligned} \tag{8.10}$$

where in the last step equation (8.8) was inserted. The $\chi_i \propto A\vec{p}_i$ are linearly independent, thus the coefficients from (8.10) can be compared to (8.9), for $m = 0, 1, \dots, k$ resulting in

$$\begin{aligned}
\alpha'_m &= \frac{1}{b_m}\left[c_m + \sum_{l=m+1}^{k} \alpha'_l a_{ml} \right] \\
&= \alpha_m - \sum_{l=m+1}^{k} \alpha'_l \beta_{l-1,m}.
\end{aligned}$$

□

Proposal 8.2: GCR in mixed precision

In the current version of openQ*D [17], the outer GCR solver is performed in pure binary64. A mixed precision variant would require the preconditioning M_{sap} to be done in mixed precision as well. Algorithm 1 would directly apply with *solve*() replaced by `fgcr()` with the difference that `fgcr()` has to accept D, M_{sap}, $\vec{x}_0$ and $\vec{b}$ in the desired precision.

8.6 Simulating SAP+GCR

8.6.1 Setup

The complete SAP+GCR kernel was implemented using Python in the same way as the `fgcr()` function from the source code[55]. The Dirac-operator `Dop_dble()` was extracted in the same way as for the `cgne()` kernel previously (see section 7.3) using the same configuration. The Python implementation contains a floating-point data type for the reduction variables separately (`rdtype`). It also accepts a "large" data type (`ldtype`) by which the restart steps are calculated in, and a "small" data type (`sdtype`) in which the regular and the MR steps are performed in. The result is obtained in terms of the "large" data type. There are various configuration settings to choose from (see table 4).

[55]See line 212ff in `modules/linsolv/fgcr.c` in [17].

setting	meaning	comment
`res`	desired relative residual	
`nmx`	maximal number of GCR steps	
`nkv`	number of generated Krylov vectors until restarting the algorithm	
`ncy`	number of SAP-cycles to perform in each iteration	
`nmr`	number of MR-steps to perform on each block in each SAP-cycle	
`bs`	block size	
`ldtype`	"large" data type	can be binary64 or binary32
`rdtype`	reduction data type	
`sdtype`	"small" data type	

Table 4: Settings for `SAP_GCR` and their meanings.

The possible data types for `ldtype`, `rdtype` and `sdtype` are binary64 and binary32. Unfortunately, there was no possibility to use binary16, bfloat16 or tensorfloat32, even though modern GPUs such as the one tested on do support these data types. The reason for this is the data types were not available in the used CUDA library, CuPy [49]. Furthermore, tensor cores are not able to accelerate sparse matrix-vector products [50].

The following plot series gives an estimate on how much speed improvement can be expected for a GPU-implementation of the solver algorithm. The results give a hint on how to optimally choose the (many) parameters for the solver. It has to be kept in mind that the transfer of the (full, boundary or blocked) Dirac-operator to the GPU is not part of the time measurements; it is assumed that the operators already reside on the correct places (CPU memory or GPU memory), only spinors are transferred back and forth. Figures 18 - 22 contain the measurements. Every datapoint represents the average of at least 20 runs of the `SAP_GCR` kernel in the given configuration and Dirac-operator. The y-axis denotes the duration in seconds and the x-axis shows the configuration (`ncy`, `nmr`) as well as the block size (`bs`) increasing in computational effort per GCR-step from left to right.

Two configurations are non-standard: $(n_{cy}, n_{mr}) = (0, 0)$ and "adap.". The former indicates no preconditioning (thus a pure GCR run) and in the latter configuration the parameters n_{cy}, n_{mr} were chosen automatically in every iteration anew (see proposal 8.3).

The colors denote the data type setup (`ldtype`, `rdtype`, `sdtype`) and the marker symbols indicate whether the calculation was performed purely on the CPU (circles; ○, ○, ○), purely on the GPU (crosses; ×, ×, ×) or a hybrid variant (diamonds; ◇, ◇, ◇), where only the MR-steps are calculated on the GPU and the remainder on the CPU, see proposal 8.1. Runs of openQ*D dealing with the same problem are indicated by black squares (if available, squares; □). All combinations of the above configurations are present in the plots. The different plots show results from different matrices. 3 matrices where extracted directly from a run of openQ*D (figures 18, 19 and 22), while 2 further matrices where taken from a matrix collection [51] (figures 20 and 21). The matrix `conf6_0-8x8-2` taken from [51] has a parameter $0 \leq k \leq k_c$, the hopping parameter (see section 4.2). The closer k is to its critical value k_c the worse the matrix is conditioned. The used values for k and k_c are given in the title of the plots. In all plots, the relative residual was chosen to be 10^{-6} and the number of GCR-steps until restart `nkv=7`.

8.6.2 Discussion of figures 18 - 22

A clear trend visible in the entire plot series is that the pure-GPU variants are most efficient when the block size is large (the number of blocks is small). The pure-CPU variants behave differently – the block size has less influence on the run-time. In general, the pure-GPU variants are faster than the pure-CPU ones. This is because they take advantage of concurrency. The hybrid-variants are as expected in-between them, behaving similarly to the pure-GPU ones.

As a further general observation, the power of the SAP+GCR solver is only fully unfolded if the condition number of the operator is large; figures 18, 19 and 22 show no significant performance

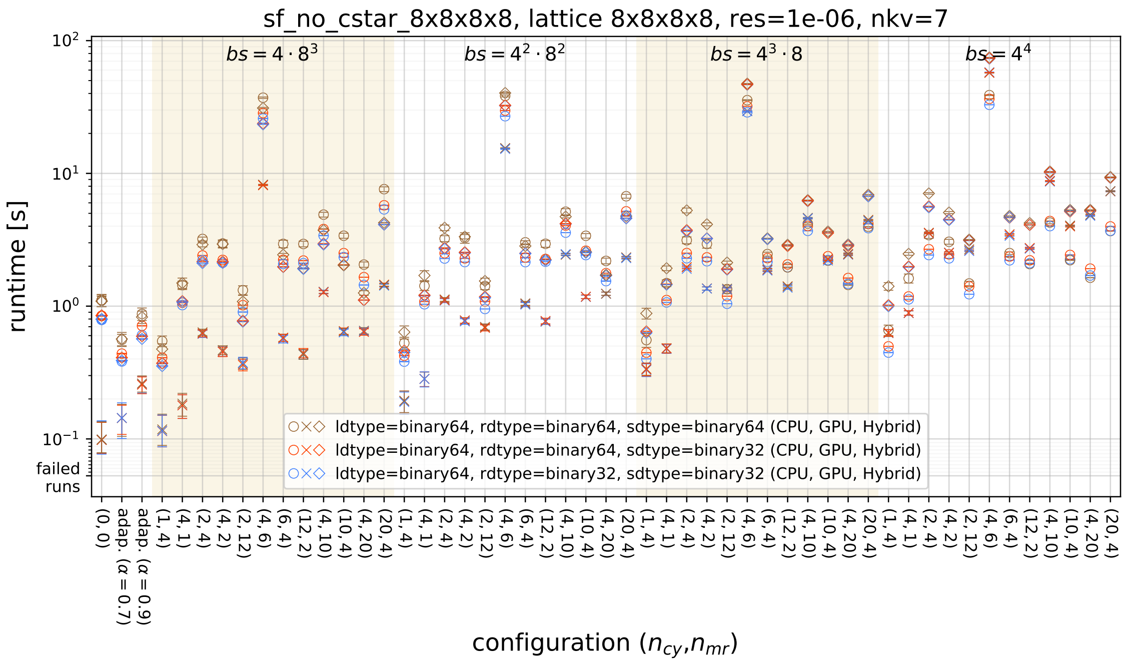

Figure 18: Time measurements for the SAP_GCR kernel on different matrices and configurations. The measurements were conducted on an AMD EPYC 7742 CPU @ 2.25GHz with 512 GB memory and an NVIDIA A100 (via SXM4) GPU with 40 GB memory.

improvement of the SAP-preconditioning compared to a pure GCR solver without preconditioning[56], whereas the runs in figures 20 and 21 do.

An analysis of the different data types shows that the general trend is the lower the involved data types are in bit-length, the faster the solution is obtained, which makes sense in memory-bound problems. The setups with binary32 in reduction variables (`rdtype`) and as "small" data type (`sdtype`) appear to be the most efficient. However, of the given data type setups one should choose the one where the data type of reduction variables (`rdtype`) is set to binary64 to prevent over- or underflows. This was already discussed in section 7.3 for the CGNE-solver.

Figure 18: Looking at the first plot, figure 18, as expected the preconditioning gives no significant improvement; on the CPU only 4 setups were faster than the one without preconditioning, on the GPU even none of the preconditioning setups surpasses the trivial case. All the CPU cases that were faster than the trivial case had the same configuration $(n_{cy}, n_{mr}) = (1, 4)$, but different block sizes. This shows that if the operator is well-conditioned, too much preconditioning worsens the performance. $(n_{cy}, n_{mr}) = (1, 4)$ is the configuration with the least amount of preconditioning. The CPU run-time shows a strong dependence on the configuration; there are even certain configurations (for example $(n_{cy}, n_{mr}) = (4, 6)$) that are more than 40 times slower than the non-preconditioning case. An unsuitable choice of configuration parameters can thus lead to a significant performance degradation. However, the plots show that performance of the algorithm is overly sensitive to the choice of these parameters. The adaptive variant should here come to the rescue.

Figure 19: The operator in figure 19 acting on a 16^4-lattice has a similar behaviour as the previous one, because it is well-conditioned as well. The same configurations give a speedup compared to the case without preconditioning, this time both the CPU and GPU variants improved. The claim that the pure-GPU variant slows down with smaller block sizes is even more visible in this plot, since there are 8 block sizes to work on. Looking at the behaviour within a certain constant block size, the algorithm seems to be unpredictable sensitive to changes in the configuration. As an example, block size $bs = 8^2 \cdot 16^2$, one would expect the run-time to either increase or decrease with respect to the amount of preconditioning. The results show a more complex dependence with an exceptional case at $(n_{cy}, n_{mr}) = (2, 8)$. This exceptional case can be seen in all block sizes and on

[56]The runs with configuration $(n_{cy}, n_{mr}) = (0, 0)$

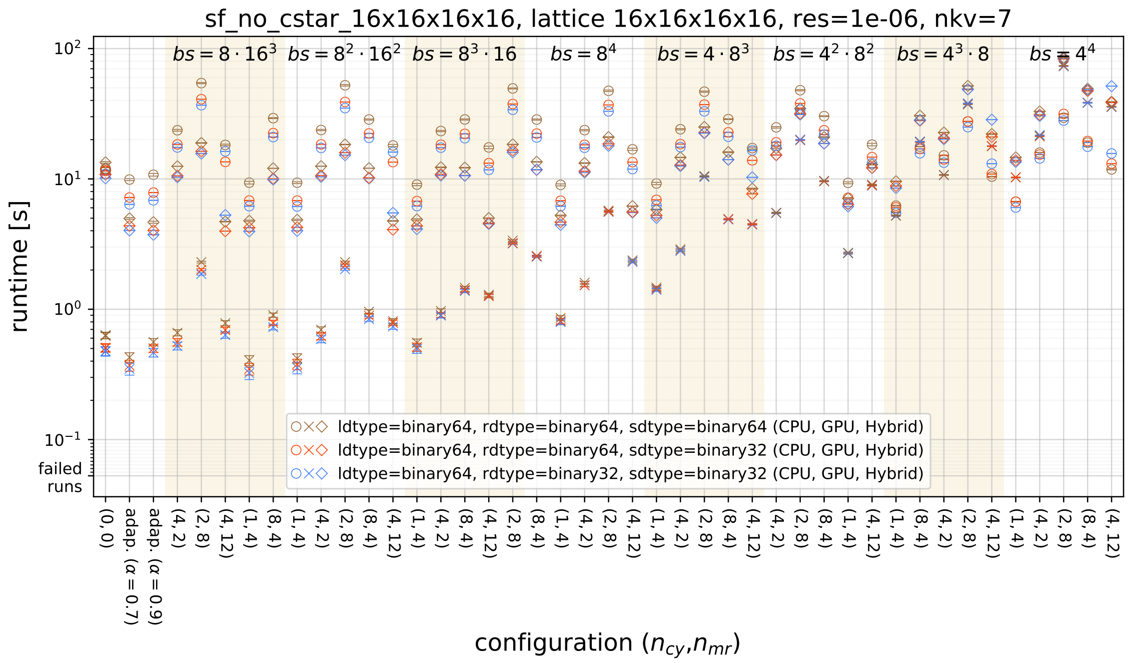

Figure 19: Time measurements for the SAP_GCR kernel on different matrices and configurations. The measurements were conducted on an AMD EPYC 7742 CPU @ 2.25GHz with 512 GB memory and an NVIDIA A100 (via SXM4) GPU with 40 GB memory.

the CPU, GPU as well as in the hybrid case. Figure 18 features such an exceptional case as well at $(n_{cy}, n_{mr}) = (4, 6)$. However, with both matrices any preconditioning makes the run-time worse, they might not be very representative.

Figure 20: We continue the discussion with the matrix `conf6_0-8x8-2` where the condition number depends on how close the $k = 0.15$ parameter is at its critical value $k_c = 0.15717$. This is the regime where the preconditioning shows benefits. For the pure-CPU cases, we see no strong dependence on the amount of preconditioning, but on the block size. Small block sizes seem to be beneficial, while the pure-GPU variant prefers large block sizes. Similar to the above analysis, the good condition number again gives not much speedup gain, when comparing the preconditioned cases with the trivial case.

Figure 21: Using the same matrix as above, but at the critical point $k = k_c$, we are in the regime where the SAP+GCR algorithm shows its true potential; nearly all cases performed better than the trivial case without any preconditioning. The pure-GPU cases behave as usual – large block sizes result in faster convergence. This time even the pure-CPU shows a dependence on the block size. Although, this dependence is weak, the CPU seems to perform slightly better on smaller block sizes (on contrary to the GPU). The hybrid cases – as usual in-between – are closer to the pure-GPU ones, because despite being hybrid most of the work is done on the GPU. The pattern within a certain block size is repeating and the best amount of preconditioning seems to be at $(n_{cy}, n_{mr}) = (4, 6)$.

Figure 22: Finally, the second matrix from openQ*D `sf_no_cstar_8x8x8x8_2` shows similar patterns as the first matrix with the same configuration. The measurement was repeated in order to check whether there is a visible pattern common to both matrices. No such pattern were observed. It is noticeable that there is an exceptional configuration that took disproportionally longer than the others at $(n_{cy}, n_{mr}) = (12, 2)$. More importantly it is not the same configuration as in figure 18, making it hard to determine the best set of parameters beforehand and at the same time maintaining a perfect amount of preconditioning for all invocations of the solver throughout a long simulation.

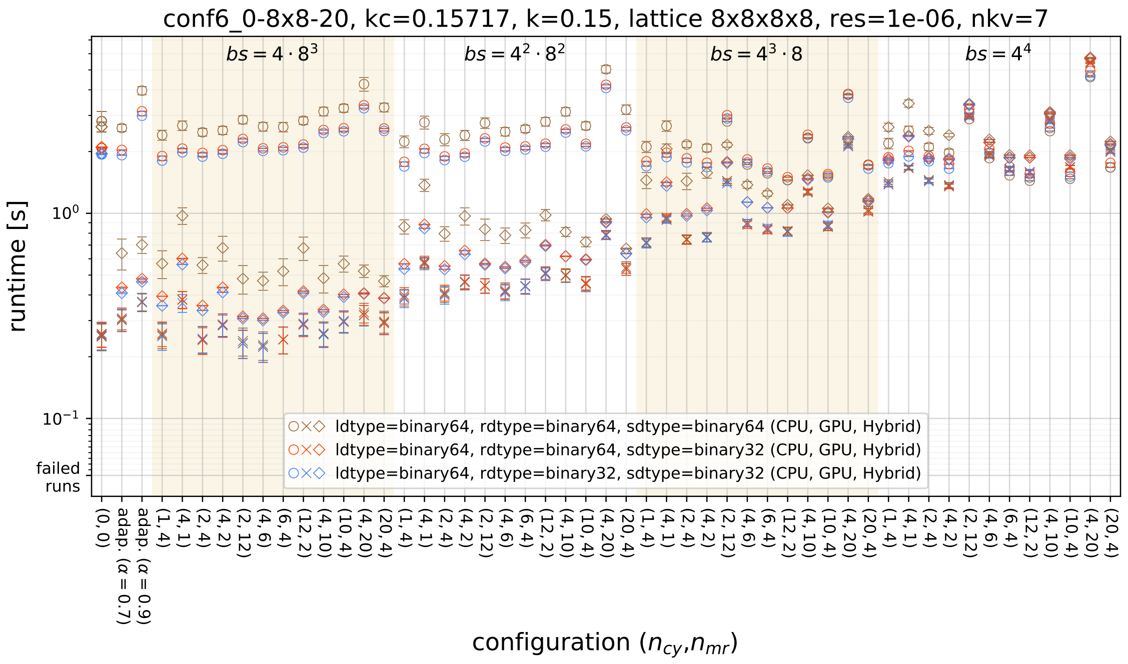

Figure 20: Time measurements for the SAP_GCR kernel on different matrices and configurations. The measurements were conducted on an AMD EPYC 7742 CPU @ 2.25GHz with 512 GB memory and an NVIDIA A100 (via SXM4) GPU with 40 GB memory.

8.6.3 Conclusion

In general, two different patterns can be observed in the above plot series. Either the matrix is bad conditioned, and one can see that most of the preconditioned runs perform better than no preconditioning. Or the matrix is good conditioned, and the preconditioned cases perform worse. In both cases the pattern within a block size repeats in other blocks sizes but shifted. We see two types of patterns. For bad conditioned systems, the pattern shows that the ideal case is at some point in the middle. On the other hand, for well-conditioned system, the pattern seems random, but an increase of run-time with more preconditioning (higher values of n_{cy} and n_{mr}) can be seen. Since the systems are already well conditioned, too much preconditioning can worsen the run-time.

Since the algorithm is applied to many different Dirac-operators among evolving HMC-trajectories – some well-conditioned, some ill-conditioned – it can be hard or even impossible to choose a set of parameters suitable for all cases. Especially, it is unavoidable to accidentally make a choice that falls on a configuration with exceptional long convergence time for a certain Dirac-operator within the long running HMC-simulation. It is therefore advisable to have the possibility to change the parameters during an active run or a configuration that adapts. This motivates the following proposal.

Proposal 8.3: Adaptive choice of parameters in SAP+GCR

Since the choice of parameters in the SAP+GCR kernel seems non-trivial, we propose an adaptive variant of this algorithm. The parameters n_{cy}, n_{mr} were chosen automatically in every iteration anew, the block size was chosen to be the largest possible. The adaptive choice was done as follows. If – after a Schwarz-cycle – the norm of the residual is not lower than the residual norm before the cycle, then the preconditioning phase is exited. Thus, at least one Schwarz-cycle is performed. A similar, but slightly more complicated strategy is applied to determine the number of MR-steps. There are 3 exit conditions for the MR-solver:

1) If – after at least 4 MR-steps – the norm of the residual on the block is larger than $\alpha = 0.9^a$ times the previous residual norm, the MR-solver exits, and the application

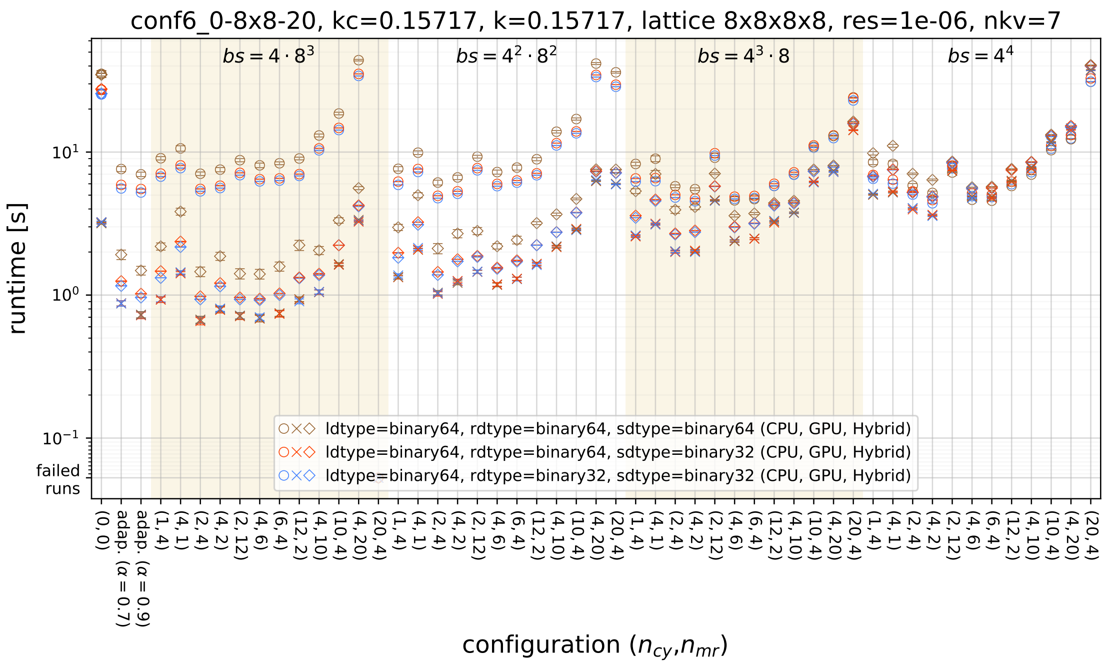

Figure 21: Time measurements for the SAP_GCR kernel on different matrices and configurations. The measurements were conducted on an AMD EPYC 7742 CPU @ 2.25GHz with 512 GB memory and an NVIDIA A100 (via SXM4) GPU with 40 GB memory.

continues processing the next block.

2) If the norm of the blocked residual becomes larger than the previous residual norm, the solver exits immediately, even if only one MR-step is executed.

3) If the norm of the blocked residual is smaller than the tolerance[b], the algorithm exit immediately too.

So, every block is treated differently in every cycle. A maximum of 20 Schwarz-cycles and 20 MR-steps on each block would be performed if the above exit conditions never kick in. The third exit condition above makes sure to not overshoot the mark if the algorithm performs a lot of Schwarz-cycles and MR-steps. Namely, if the problem is already solved while in the preconditioning phase. This can happen if the operator is very well-conditioned or in the very last GCR-step before converging to the desired relative residual. Therefore, the adaptive version tries to find the optimal configuration for every iteration of the GCR-solver, for every Schwarz-cycle and for every block separately. By empirical observation of the results, the adaptive variant usually performs nearly maximal amounts of preconditioning in the first few GCR-steps, then rapidly decreases after some steps to much smaller amounts and finally saturate to the minimal amount that stays until convergence.
The results on how this adaptive variant competes with static configurations can be seen in figures 18 - 22. The adaptive runs are indicated with configuration adap. ($\alpha = 0.9$) and adap. ($\alpha = 0.7$). Although the adaptive variant of the algorithm is not the fastest among all configurations, the plots show that it is certainly the most versatile one. It can be of benefit if the condition of the operator is not known beforehand and might even change drastically within a long running simulation.
A reference implementation was written directly in openQ*D and can be found in the GitLab repository ref. [52].

[a]Ironically, the choice for the value of $\alpha \in (0, 1]$ is again non-trivial. Small values cause less preconditioning while values close to 1 will end up in more or even the maximal number of MR-steps. But since we want to

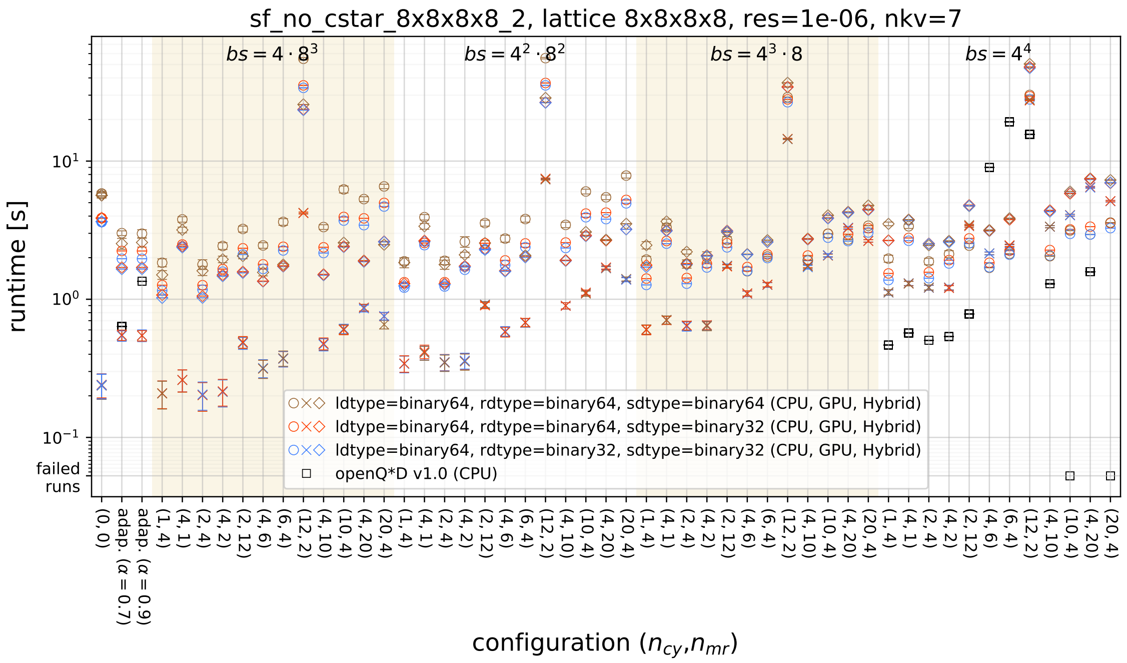

Figure 22: Time measurements for the SAP_GCR kernel on different matrices and configurations. The measurements were conducted on an Intel(R) 6130 @ 2.10GHz with 1.5 TB memory and an NVIDIA V100 (via PCIe) GPU with 16 GB memory.

optimise for ill-conditioned systems and the penalty for well-conditioned systems is acceptable, it is advisable to choose α large, such as $\alpha = 0.9$

[b]This is the tolerance calculated in the GCR solver divided by the number of blocks, $tol = res * \|\eta\|/n_b$, where `res` is the desired relative residual given as configuration option (see table 4), $\eta = \vec{b}$ is the source vector and n_b is the number of blocks.

9 Deflated SAP preconditioned GCR

Small quark masses corresponding to real physics are believed to be the cause for the spontaneous breaking of chiral symmetry in lattice QCD [53]. Numerical lattice QCD has the problem that with large lattice volumes and small quark masses simulation techniques become inefficient in the ***chiral regime*** (where chiral symmetry is spontaneously broken), because the Dirac-operators gets increasingly ill-conditioned. Thus, the presence of low eigenvalues is a source of difficulty [54]. According to the Bank-Casher relation, see [53], this is because the number of eigenvalues of D below a fixed value grows with $O(V)$, where V is the total 4D lattice volume. On the other hand, the computational effort scales even worse with $O(V^2)$, [47]. This behaviour goes under the name of V^2***-problem***.

A solving algorithm that has a flat scaling with respect to the quark masses can therefore lead to large speedups specially in the chiral regime. The topic of this section is the deflated SAP+GCR algorithm, DFL+SAP+GCR[57]. By deflating the Dirac-operator, it is possible to separate eigenmodes with small eigenvalues from the bulk. Thus, the space needs to be split in low and high modes without actually calculating the modes, else the problem would be solved already.

[57]Deflation – as described in the following section – can be applied to any linear solving algorithm.

9.1 Deflation

Theorem 9.1 (Deflation). *Let A be a linear, invertible operator acting on a vector space Λ, $\vec{b} \in \Lambda$ an arbitrary vector and P_L a projector*[58] *acting on Λ. Furthermore, define the linear operator P_R such that $P_L A = A P_R$*[59]*. Consider*

$$\vec{x}^\star := P_R \vec{x}_1^\star + (1 - P_R)\vec{x}_2^\star, \tag{9.1}$$

with $\vec{x}_1^\star$ and $\vec{x}_2^\star$ being solutions to the "smaller" (projected) systems

$$P_L A \vec{x}_1 = P_L \vec{b} \qquad \text{and} \qquad (1 - P_L) A \vec{x}_2 = (1 - P_L)\vec{b},$$

respectively. Then

1) P_R is a projector,

2) $\vec{x}^\star$ is the unique solution to $A\vec{x} = \vec{b}$.

Proof. Using that $P_L^2 = P_L$ is a projector and the defining relation $P_L A = A P_R$,

$$\begin{aligned} P_R^2 &= (A^{-1} P_L A)^2 \\ &= A^{-1} P_L^2 A \\ &= A^{-1} P_L A \\ &= P_R. \end{aligned}$$

By direct calculation,

$$\begin{aligned} A\vec{x}^\star &= A P_R \vec{x}_1^\star + A(1 - P_R)\vec{x}_2^\star \\ &= P_L A \vec{x}_1^\star + (1 - P_L) A \vec{x}_2^\star \\ &= P_L \vec{b} + (1 - P_L)\vec{b} \\ &= \vec{b}. \end{aligned}$$

Since A is invertible $\vec{x}^\star$ is unique, although $\vec{x}_1^\star$ and $\vec{x}_2^\star$ may not be unique, their projections are. □

Remark. If we find clever subspaces in which the projectors P_L and P_R project without involving A^{-1}, we can solve $A\vec{x} = \vec{b}$ by solving the two smaller systems of equations and then projecting the solutions using P_R.

Remark. Notice that $P_L A$ as well as $(1 - P_L)A$ are not invertible. There are infinitely many solutions $\vec{x}_1^\star$ and $\vec{x}_2^\star$[60]. Nonetheless the solution vector $\vec{x}^\star$ is still unique after the projection in equation (9.1), because P_R removes all ambiguity from $\vec{x}_1^\star$ and $\vec{x}_2^\star$.

Remark. Comparing deflation to left preconditioning, the difference is that in deflation P_L is a projector and $P_L A$ has condition number infinite (but finite effective condition number) while in case of preconditioning P_L is invertible (a good approximation of A^{-1}) and the condition number of $P_L A$ is expected to be smaller than the one of A.

Remark. If A is positive definite, $P_L A$ is positive semi-definite and has condition number infinite. Furthermore, $P_L A \vec{x} = P_L \vec{b}$ has infinite solutions. Fortunately, such a system can be solved as long as the right-hand side is consistent, meaning that there exists an $\vec{x}$ solving $A\vec{x} = \vec{b}$, [55].

[58] P_L does not have to be orthogonal or Hermitian.

[59] Such a linear operator P_R always exists – just set $P_R := A^{-1} P_L A$, since A is invertible.

[60] Let P be a linear projector (not the identity-operator) and A an invertible linear operator. The system of interest is $PA\vec{x} = P\vec{b}$. There exists at least one solution to this, namely the unique solution to $A\vec{x} = \vec{b}$. Since PA is not invertible, the only two possibilities are either zero or infinite solutions to $PA\vec{x} = P\vec{b}$, but it can't be zero solutions.

Therefore, it makes sense to define the relevant quantity.

Definition 9.1 (Condition number)**.** *The* ***condition number*** κ *and the* ***effective condition number*** κ_{eff} *of a matrix A are the ratios between the largest and the smallest eigenvalues*

$$\kappa(A) := \frac{|\lambda_{max}(A)|}{|\lambda_{min}(A)|}, \qquad \text{and} \qquad \kappa_{eff}(A) := \frac{|\lambda_{max}(A)|}{|\lambda_{pmin}(A)|},$$

where

$$\begin{aligned} |\lambda_{max}(A)| &:= \max|\sigma(A)|, \\ |\lambda_{min}(A)| &:= \min|\sigma(A)|, \\ |\lambda_{pmin}(A)| &:= \min\{|\lambda| \bigm| \lambda \in \sigma(A), |\lambda| > 0\}. \end{aligned}$$

Remark. The condition number and the effective condition number are equal if the matrix A is invertible.

Corollary 9.2. *Let A, Λ and $\vec{b}$ be as in theorem 9.1. Furthermore let $\{\vec{\omega}_i\}_{i=1}^N$ be an orthonormal basis of a linear subspace $\Omega \subset \Lambda$, called the* ***deflation subspace*** *and let the restriction of A to Ω, $\widetilde{A} := A\big|_{\Omega}$ called the* ***little operator****, be invertible. Define P_L by its action on an arbitrary vector $\vec{x} \in \Lambda$ as*

$$P_L\vec{x} := \vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle$$

and let $\vec{x}_1^\star$ be one of the (infinite) solutions to the ***deflated system*** *$\hat{A}\vec{x}_1 = P_L\vec{b}$, where $\hat{A} := P_L A$ is called the* ***deflated operator****. Consider*

$$\vec{x}^\star := P_R\vec{x}_1^\star + \sum_{i,j=1}^{N} \vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{b}\rangle, \tag{9.2}$$

with P_R satisfying $P_L A = AP_R$. Then $\vec{x}^\star$ is the unique solution to the linear system of equations $A\vec{x} = \vec{b}$.

Proof. Let's first show that $P_L^2 = P_L$ is a projector,

$$\begin{aligned} P_L^2\vec{x} &= P_L\left(\vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle\right) \\ &= \vec{x} - 2\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle + \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\sum_{k,l=1}^{N}\langle\vec{\omega}_j, A\vec{\omega}_k\rangle(\widetilde{A}^{-1})_{kl}\langle\vec{\omega}_l, \vec{x}\rangle \\ &= \vec{x} - 2\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle + \sum_{i,j,l=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_l, \vec{x}\rangle\underbrace{\sum_{k=1}^{N}\underbrace{\langle\vec{\omega}_j, A\vec{\omega}_k\rangle}_{=\widetilde{A}_{jk}}(\widetilde{A}^{-1})_{kl}}_{=\delta_{jl}} \\ &= \vec{x} - 2\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle + \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle \\ &= \vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle \\ &= P_L\vec{x}. \end{aligned}$$

The second term in equation (9.2) is equal to $(1 - P_R)\vec{x}_2^\star$ where $\vec{x}_2^\star$ solves the projected system $(1 - P_L)A\vec{x}_2 = (1 - P_L)\vec{b}$:

$$\begin{aligned}(1 - P_R)\vec{x}_2^\star &= A^{-1}(1 - P_L)A\vec{x}_2^\star \\ &= A^{-1}(1 - P_L)\vec{b} \\ &= A^{-1}\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{b}\rangle \\ &= \sum_{i,j=1}^{N} \vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{b}\rangle,\end{aligned}$$

which corresponds to the second term of $\vec{x}^\star$ in equation (9.2), therefore $\vec{x}^\star \coloneqq P_R\vec{x}_1^\star + (1 - P_R)\vec{x}_2^\star$. By application of theorem 9.1, $\vec{x}^\star$ is the unique solution to $A\vec{x} = \vec{b}$.

□

Remark. From P_L in corollary 9.2, the action of P_R on an arbitrary vector $\vec{x}$ can be determined using the defining relation of P_R as

$$\begin{aligned}P_R\vec{x} &= A^{-1}P_LA\vec{x} \\ &= \vec{x} - \sum_{i,j=1}^{N} \vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, A\vec{x}\rangle.\end{aligned}$$

Remark. An application of P_L to an arbitrary vector $\vec{x}$ involves solving the ***little system*** $\widetilde{A}\vec{\beta} = \vec{\alpha}$ on Ω for a given $\vec{\alpha} \in \Omega$. To see this, let's look at the k-th component of $P_L\vec{x}$,

$$(P_L\vec{x})_k \coloneqq x_k - \sum_{i,j=1}^{N} (A\vec{\omega}_i)_k(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle.$$

Define the vector

$$\vec{\alpha}_{\vec{x}} \coloneqq \begin{pmatrix}\langle\vec{\omega}_1, \vec{x}\rangle \\ \langle\vec{\omega}_2, \vec{x}\rangle \\ \vdots \\ \langle\vec{\omega}_N, \vec{x}\rangle\end{pmatrix} = \begin{pmatrix}(\vec{\omega}_1)_1 & \dots & (\vec{\omega}_1)_N \\ (\vec{\omega}_2)_1 & \dots & (\vec{\omega}_2)_N \\ \vdots & \ddots & \vdots \\ (\vec{\omega}_N)_1 & \dots & (\vec{\omega}_N)_N\end{pmatrix}\begin{pmatrix}x_1 \\ x_2 \\ \vdots \\ x_N\end{pmatrix}.$$

Then,

$$\begin{aligned}(P_L\vec{x})_k &= x_k - \sum_{i=1}^{N}(A\vec{\omega}_i)_k(\widetilde{A}^{-1}\vec{\alpha}_{\vec{x}})_i \\ &= x_k - \sum_{i=1}^{N}(A\vec{\omega}_i)_k\vec{\beta}_i.\end{aligned}$$

By similar analysis, an application of P_R has the same cost with one additional application of A. For an efficient implementation of P_L and P_R, the $2N$ vectors $\{A\vec{\omega}_i\}_{i=1}^N$ and $\{\vec{\omega}_i\}_{i=1}^N$ have to be kept in memory.

Remark. We assume that the condition number of A is high and the ***spectrum*** of A, $\sigma(A)$, is separable in a way such that

$$\sigma(A) = \sigma_l(A) \cup \sigma_h(A) \qquad \text{with} \qquad \max_{\lambda\in\sigma_l(A)} |\lambda| \ll \min_{\lambda\in\sigma_h(A)} |\lambda|. \tag{9.3}$$

The subscripts stand for "low" and "high", corresponding to the low and high modes of the operator A. The property in equation (9.3) states that the bulk of the low eigenvalues are somewhat clustered[61]. Consider the linear sub-spaces $\Omega_l, \Omega_h \subset \Lambda$ such that the low and high eigenvectors corresponding to the low and high eigenvalues of A are contained in Ω_l and Ω_h, respectively. Then the condition number of A restricted to the low modes is much smaller than the condition number of A. Therefore, if we are able to find an orthonormal basis $\{\vec{\omega}_i\}_{i=0}^N$ of the subspace Ω_l containing the bulk of the low eigenmodes of A, we can apply deflation from corollary 9.2 to solve the little equation that has a significantly smaller condition number than A. Then find one of the infinite solutions to the deflated system and construct a solution of the full system.

Lemma 9.3. *Let A, $\{\vec{\omega}_i\}_{i=1}^N \in \Omega$, P_L, P_R be as in corollary 9.2 and assume that the spectrum of A is separable, equation (9.3). Define the deflation subspace to be the subspace corresponding to the low eigenmodes, $\Omega := \Omega_l$. Then $\kappa_{eff}(\hat{A}) \ll \kappa(A)$ and $\kappa(\tilde{A}) \ll \kappa(A)$.*

Proof. Let's define the orthogonal projector $P^\perp$ to $\Omega^\perp$, the orthogonal complement of the deflation subspace Ω,

$$P^\perp \vec{x} := \vec{x} - \sum_{i=1}^N \langle \vec{\omega}_i, \vec{x} \rangle \vec{\omega}_i.$$

The deflated operator $\hat{A} := P_L A$ acts on the orthogonal complement,

$$\begin{aligned}
\hat{A} P^\perp \vec{x} &= P_L A P^\perp \vec{x} \\
&= P_L A \vec{x} - \sum_{k=1}^N P_L A \vec{\omega}_k \langle \vec{\omega}_k, \vec{x} \rangle \\
&= A\vec{x} - \sum_{i,j=1}^N A\vec{\omega}_i (\widetilde{A}^{-1})_{ij} \langle \vec{\omega}_j, A\vec{x} \rangle - \sum_{k=1}^N A\vec{\omega}_k \langle \vec{\omega}_k, \vec{x} \rangle + \sum_{i,k=1}^N A\vec{\omega}_i \underbrace{\sum_{j=1}^N (\widetilde{A}^{-1})_{ij} \underbrace{\langle \vec{\omega}_j, A\vec{\omega}_k \rangle}_{=\widetilde{A}_{jk}}}_{\delta_{ik}} \langle \vec{\omega}_k, \vec{x} \rangle \\
&= A\vec{x} - \sum_{i,j=1}^N A\vec{\omega}_i (\widetilde{A}^{-1})_{ij} \langle \vec{\omega}_j, A\vec{x} \rangle - \sum_{k=1}^N A\vec{\omega}_k \langle \vec{\omega}_k, \vec{x} \rangle + \sum_{k=1}^N A\vec{\omega}_k \langle \vec{\omega}_k, \vec{x} \rangle \\
&= A\vec{x} - \sum_{i,j=1}^N A\vec{\omega}_i (\widetilde{A}^{-1})_{ij} \langle \vec{\omega}_j, A\vec{x} \rangle \\
&= P_L A \vec{x} \\
&= \hat{A}\vec{x}.
\end{aligned}$$

This now means, that the low eigenmodes are projected to zero, and do not participate to the effective condition number of $\hat{A}$, which can now be upper bounded,

$$\kappa_{eff}(\hat{A}) = \frac{\left|\lambda_{max}(\hat{A})\right|}{\left|\lambda_{pmin}(\hat{A})\right|} = \frac{|\lambda_{max}(A)|}{\left|\lambda_{pmin}(\hat{A})\right|} \ll \frac{|\lambda_{max}(A)|}{|\lambda_{min}(A)|} = \kappa(A),$$

where property (9.3) was used in the second inequality. Similarly, for the condition number of the little operator $\kappa(\tilde{A})$ where we have $\sigma(\tilde{A}) = \sigma(A\big|_\Omega) = \sigma_l$ which directly implies $\kappa(\tilde{A}) \ll \kappa(A)$. □

Remark. Lemma 9.3 tells us that the deflated and the little system are significantly better conditioned than the full system and both are therefore solved in fewer iterations[62]. This holds only if

[61] The high eigenmodes may not be clustered, but still separated from the low ones.

[62] Additionally, the little system is much smaller in size than the full system, but the sparsity will not stay – it might even become dense. The precision up to which this system has to be solved in an application of the projector is usually a fraction of the desired precision of the full solution.

we can find the subspace corresponding to the low eigenmodes Ω_l and if the low eigenvalues of the matrix are somewhat isolated from the bulk of the eigenvalues. Equation (9.3) gives us a condition on A for an efficient deflation.

Lemma 9.4. *P_L as defined in corollary 9.2 is a projection to the orthogonal complement of Ω, i.e. $\langle \vec{\omega}_k, P_L \vec{x} \rangle = 0$.*

Proof. Let $\vec{x}$ be an arbitrary vector, and $k \in \{1, \dots, N\}$, then

$$\begin{aligned}
\langle \vec{\omega}_k, P_L \vec{x} \rangle &= \langle \vec{\omega}_k, \vec{x} \rangle - \sum_{i,j=1}^{N} \langle \vec{\omega}_k, A\vec{\omega}_i \rangle (\widetilde{A}^{-1})_{ij} \langle \vec{\omega}_j, \vec{x} \rangle \\
&= \langle \vec{\omega}_k, \vec{x} \rangle - \sum_{j=1}^{N} \langle \vec{\omega}_j, \vec{x} \rangle \sum_{i=1}^{N} \widetilde{A}_{ki} (\widetilde{A}^{-1})_{ij} \\
&= \langle \vec{\omega}_k, \vec{x} \rangle - \sum_{j=1}^{N} \langle \vec{\omega}_j, \vec{x} \rangle \delta_{kj} \\
&= 0.
\end{aligned}$$

□

9.2 Choosing the deflation subspace

Besides isolated low eigenmodes, the Dirac-operator should feature another property for deflation to be efficient. The computational cost to find an approximation of the low-mode deflation subspace should not be too high. It turns out that if the low eigenmodes of the Dirac-operator possess a certain property, only a few low eigenmodes are needed to construct a good approximation of the subspace of low eigenmodes Ω_l. If the subspace of low eigenmodes is only approximated, the method is called ***inexact deflation***.

Definition 9.2 (Local coherence)**.** *Let $1 \gg \varepsilon > 0$ and $\{\Omega_i\}_{i=1}^{n_b}$ be a decomposition of the full lattice Λ into n_b blocks (for example as in section 8.2 figure 15) and let P_i be the projector to block Ω_i. Furthermore define the **bootstrap set** $B = \{\vec{\beta}_j\}_{j=1}^{M}$ as a set of normalized fields satisfying $\left\| (1 - P_{exact})\vec{\beta}_j \right\|^2 \leq \varepsilon$, where P_{exact} is a projector to Ω_{exact}, a subspace of dimension $\dim(\Omega_{exact}) = N \gg M = \dim(\operatorname{span} B)$. Define the projected fields $\vec{\omega}_{ij} := P_i \vec{\beta}_j$, their subspace $\Omega_{inexact} = \operatorname{span}\{\vec{\omega}_{ij}\}_{i,j=1}^{n_b,M}$ and the projector $P_{inexact}$ which projects to $\Omega_{inexact}$.*

*The fields in Ω_{exact} are called **locally coherent** up to deficits ε, if there exists a non-empty bootstrap set B, such that for all normalized $\vec{\psi} \in \Omega_{exact}$,*

$$\left\| (1 - P_{inexact})\vec{\psi} \right\|^2 \leq \varepsilon.$$

*The value of ε is called the **deficit**.*

Remark. In the definition above the ***bootstrap fields*** $\vec{\beta}_j \in B$ are, to a good approximation, linear combinations of fields in Ω_{exact}, because – when projected to the complement of Ω_{exact} – their norm nearly vanishes. For the same reason, the ***bootstrap subspace*** $\operatorname{span} B$ is, to a good extent, a subspace of Ω_{exact}.

Remark. Definition 9.2 can be interpreted such that the fields in Ω_{exact} can be well approximated by the (much fewer) bootstrap fields (by projecting them to the blocks), since the approximation error is smaller or equal to ε.

Remark. Definition 9.2 can also (very informally) be restated as Ω_{exact} is approximately the span of the projected bootstrap fields.

Remark. It turns out that the low eigenmodes with subspace $\Omega_{exact} = \Omega_l$ of the Dirac-operator are locally coherent[63].

[63]This is not proven, but numerical experiments show a strong indication of local coherence, [47].

It remains to describe how to obtain the M bootstrap fields $\vec{\beta_j}$. Ideally, they should be low eigenmodes of the Dirac-operator or linear combinations thereof. Usually, one starts with randomly generated vectors and repeatedly applies the inverse power method (see theorem A.1 in appendix A). The resulting vectors may not be exact eigenvectors, but the components associated to the complement of the low modes quickly decrease after some inverse iteration steps (see corollary A.2 in appendix A). These vectors are then, up to some $\varepsilon > 0$, linear combinations of low eigenmodes and can thus be used as bootstrap fields to generate a subspace that approximates the low eigenmode subspace well. At this point the property of local coherence can be exploited and efficient deflation subspaces can be generated from these fields by projecting them onto some block decomposition.

To summarize the results, if the Dirac-operator has a spectrum that is separable in the sense of equation (9.3) and the low eigenmodes are locally coherent for a small ε, then inexact deflation can be applied to solve the ill-conditioned Dirac equation and the procedure is more efficient than without deflation, because the deflated system is better conditioned (lemma 9.3). Moreover, generating the inexact deflation subspaces can be done efficiently using only a few low eigenmodes.

9.3 Simulating DFL+SAP+GCR

9.3.1 Setup

The DFL+SAP+GCR-kernel was implemented in Python by extending the already existing implementation of SAP+GCR. Only the preconditioning phase has changed. The deflated variant of the algorithm takes the same arguments as the regular one, see table 4. In addition to these settings, the deflated algorithm requires configuration options for the deflation subspace and the settings concerning the solver for the little equation, see table 5.

setting	meaning	comment
`Ns`	number of bootstrap fields	
`bs`	size of the deflation blocks	not to be confused with the block size for SAP
`res`	desired relative residual	for solving the little equation
`nmx`	maximal number of GCR steps	for solving the little equation

Table 5: Additional settings for `DFL_SAP_GCR` and their meanings. The first two options concern the generation of the deflation subspace, while the last two options deal with the little equation.

The following plot series shows the interplay between deflation and Schwarz-preconditioning in terms of run-time. Figures 23 - 24 contain the measurements. The relative residual for the full system was chosen to the 10^{-9}, the block size for the Schwarz-preconditioning is 4^4 and the number of Krylov vectors until a restart was chosen to be $n_{kv} = 7$. The entire algorithm was executed on the CPU. Hybrid or pure-GPU runs are expected to look very similar in shape, but shifted in run-time. The number of required GCR-steps will be equal to the pure-CPU case. The plots show the dimension of the deflation subspace in x-direction and the amount of Schwarz-preconditioning in terms of (n_{cy}, n_{mr}) in y-direction. The colors denote the run-time and the numbers within the blocks show the absolute run-time in seconds needed to solve the system and the number of outer GCR-steps needed until convergence (the value of the `status` variable). The time needed for the generation of the deflation subspaces is not included in the measurements. For each time measurement, the average of 10 runs was taken. A vanishing dimension of the deflation subspace, $\dim \Omega = N_s n_b = 0$, indicates that no deflation is applied to the problem. All deflation subspaces were generated by starting with N_s random vectors and n_b blocks to project onto, followed by applying the procedure described in section 9.2. The number of inverse iteration steps was chosen to be $n_{inv} = 5$, where for each inversion one step of a SAP+GCR-solver was applied with $(n_{cy}, n_{mr}) = (5, 4)$.

9.3.2 Discussion of figures 23 - 24

The analysis was done on two Dirac-operators differing substancially in their condition number. The operator in figure 23 is bad-conditioned, while in figure 24 the operator is close to the identity. It is expected that both plots show some similarity. Namely the number of GCR-steps needed should decrease when going from the lower left point upwards or to the right. There will be more

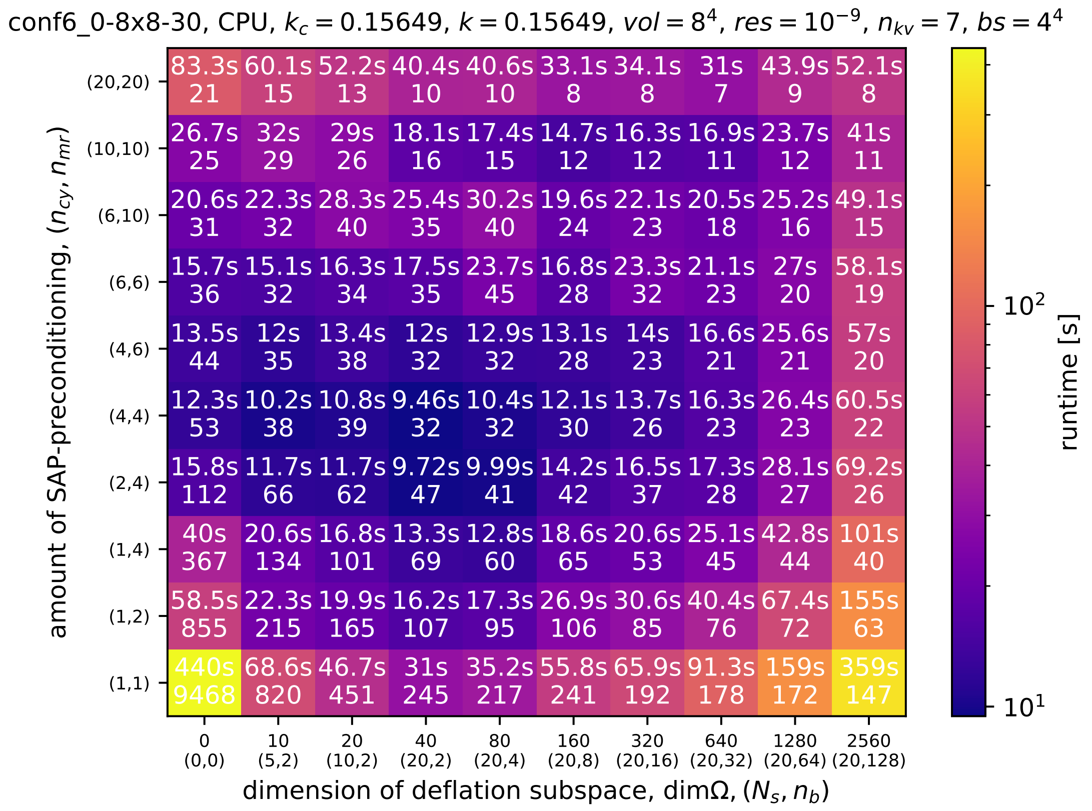

Figure 23: Time measurements for the DFL_SAP_GCR kernel on different matrices and configurations. The measurements were conducted on an Intel(R) 6130 @ 2.10GHz with 1.5 TB memory and an NVIDIA V100 (via PCIe) GPU with 16 GB memory.

preconditioning and deflation in the mentioned directions, leading to fewer needed GCR-steps. The configuration at the top right should thus have the least GCR-steps out of all configurations. However, this does not imply that it is the fastest, because each GCR-step involves heavy amounts of Schwarz-preconditioning, and the deflation subspace is the largest. Thus, the top right configuration is the one most intense in computational effort per GCR-step.

Figure 23: Figure 23 is of special interest because the operator is in bad condition, thus well suited for delfation as well as Schwatrz-preconditioning. The pattern we observe has a bowl-shape, where the minimum is placed at some certain configuration. Configurations in the neighbourhood of the minimum perform similar. Starting from the minimum, the run-time increases in every direction. The 100 configurations considered seem to sample the configuration space quite well. The plot shows that there is a non-trivial interplay between preconditioning and deflation. Clearly, solving the little equation reduces the components of the full residue associated to the low eigenmodes. On the other hand, preconditioning consolidates the spectrum around the value 1. This compactification affects the high modes more than the low modes, leading to a reduction of the high-mode components of the full residue. Therefore, the algorithm alternatives between treating the high and low mode components in the current solution vector.

Figure 24: In contrast to the previous operator, the matrix in figure 24 is well-conditioned. The plot tells us that in this case low-mode deflation is a waste of resources. Theoretically, this can be understood, because a well-conditioned operator has no separable spectrum in the sense of equation (9.3). Thus, the proofs in section 9.1 give us a reasoning why deflation is not efficient in this case. Applying the results from section 8.6, Schwarz-preconditioning should be kept within a tight limit for good conditioned systems. As stated previously, we expect the number of GCR-steps to decrease, when using larger deflation subspaces. On the contrary, when looking at the configuration $(n_{cy}, nmr) = (1, 1)$, we observe an *increase* to the right instead. This suggests that

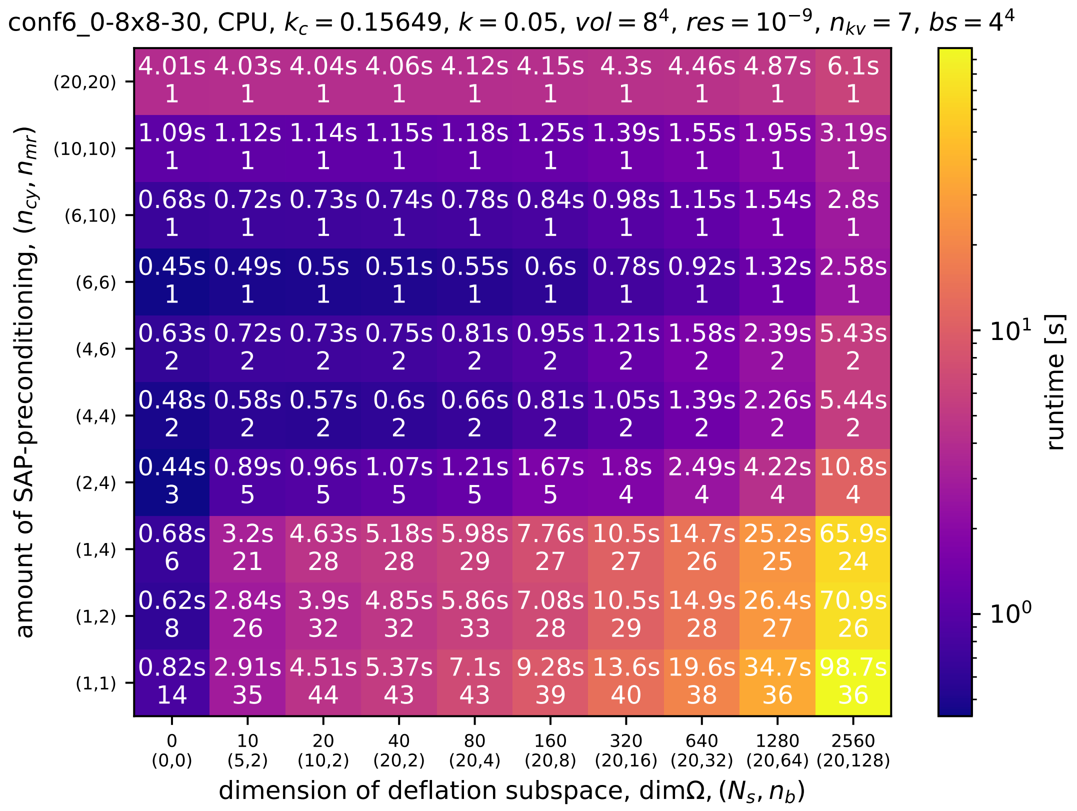

Figure 24: Time measurements for the `DFL_SAP_GCR` kernel on different matrices and configurations. The measurements were conducted on an Intel(R) 6130 @ 2.10GHz with 1.5 TB memory and an NVIDIA V100 (via PCIe) GPU with 16 GB memory.

too large deflation subspaces cause the solver to perform even worse, not only in run-time as it is with Schwarz-preconditioning, but also in the number of GCR-steps, finally leading to a regression. This specific operator has a clustering of all its eigenvalues around 1. The exact low-mode eigenspace is still of the same size as for the operator in figure 23, but the low-eigenvalues are not clearly separated anymore from the bulk of eigenvalues. A separation is thus harder to achieve, making the deflation subspace arbitrary and the deflation of the full system unprogressive and expensive. For deflation to be effective in such a case, one needs to determine the N_s bootstrap vectors more precisely by using more inverse iteration steps, making the generation of deflation subspace computationally more expensive. Obviously, the better alternative would be to drop deflation entirely and minimize Schwarz-preconditioning for such a system.

9.3.3 Conclusion

The run-time seems to be convex in the dimension of the deflation subspace. This suggests that the true subspace of low eigenmodes lies approximately at the minimum of that convex curve. As a justification imagine that the subspace of low eigenmodes has dimension d, but the algorithm produced a subspace of dimension $d_1 \gg d$. This implies that the generated deflation subspace must contain high eigenmodes as well, making the little system increasingly ill-conditioned. This results in longer run-times, because the little equation has to be solved in every (outer) GCR-step. On the other hand, let's assume the algorithm generates a deflation subspace of dimension $d_2 \ll d$. Since only a few low eigenmodes can be split off, the deflated equation is still in bad condition (lemma 9.3 does not apply). Although the little equation seems to be in good condition and small in its dimension, thus solved fast, it hardly contributes to the full solution. Again, resulting in longer run-times. Certainly, the perfect size for the deflation subspace is approximately the size of the subspace of the (exact) low eigenmodes. Of importance is that the bulk of the low eigenmodes is contained in it, such that the theorems of the previous section apply.

As mentioned before, the plots exhibit a bowl-shape, where the minimum gives the perfect balance between Schwarz-preconditioning and inexact low-mode deflation. The position of this minimum is certainly a property of the operator in consideration and depends on its condition number. If the number of low modes estimated with little computational effort, then together with the adaptive variant of the SAP+GCR algorithm one obtains a powerful and versatile solver.

Thinking about utilizing the GPU for the deflated solver, we see that the little equation should not be transferred to and solved on the GPU. Its dimensionality is a fraction of the full problem size, even for large lattices this is still small. Results in section 8.6 show the GPU being efficient only on large sub-problems, while on small blocks it is clearly outperformed by the CPU. On the contrary, if the deflation is combined with Schwarz-preconditioning, the blocked problems can still be solved on the GPU efficiently and the conclusions of section 8.6 apply.

10 Dirac operator

The lattice Dirac-operator (4.3) in openQ*D is implemented in terms of the various gauge-fields $U_\mu(n)$, only these are stored in memory[64]. In every HMC-trajectory the fields are updated, and the operator changes. In every Monte Carlo step the Dirac-operator has to be inverted for a given source vector to create the pseudofermion action, see equation (4.7). The construction of correlation functions requires the inverse of the operator from a given source point to any other point on the lattice. This requires 12 rows or columns of D^{-1}, thus 12 inversions. Or simply calculating the fermion-propagator from one lattice point to another requires an inversion as well (see equation (4.5)). In summary, inverting the Dirac-operator takes about 90% of computation time in a lattice simulation. It is thus a fundamental kernel in every numerical lattice QFT calculation. Optimizing it increases the performance of *any* solver.

10.1 Dirac-operator with threads

Modern processors today tend to have multiple cores. Two major programming models programmers are faced with are OpenMP [56] and MPI [57]. Where MPI favours the scalability of large clusters with distributed memory, OpenMP favours speed of shared memory systems and the advantages of Non-Uniform Memory Access (NUMA). Comparing the two solutions gives motivation to look at OpenMP in more detail, especially considering OpenMP for intra-node traffic or traffic within a NUMA-domain. The goal is certainly a hybrid solution that keeps the scalability of MPI, but exploits NUMA-systems using OpenMP, since OpenMP seems to outperform MPI in most cases on a single node [58].

10.1.1 Terminology

It makes sense to agree on terminology before analysing and discussing the results.

- ***Node***: Physical hardware device connected to the network of the supercomputer. Each node runs its instance of the operating system.
- ***multicore CPU***: Hardware device that is attached to a socket. It consists of one or more physical cores on the same chip.
- ***CPU, Processor, Physical core***: Hardware unit capable of independently performing basic arithmetic, logic, controlling, and (I/O) operations. A single physical core may correspond to one or more logical cores. Multiple physical cores may share L2 and L3 caches. Physical cores are associated to NUMA-nodes.
- ***Logical core***: Logical cores are the abilities of a single physical core to execute two or more threads or processes concurrently. Logical cores associated to the same physical core share their caches.

[64]To be precise openQ*D also implements the Sheikholeslami-Wohlert-term [21] which needs to store a set of fields as well and the boundary $O(a)$-improvement term that depends on two constants c_F, c'_F, whose values depend on the chosen boundary conditions.

- ***Process, Rank***: As opposed to threads, processes or ranks do not share resources such as memory. Different instances of processes need to setup a communication channel to be able to communicate (using communication libraries such as MPI).
- ***Thread***: Sequence of programmed instructions. Multiple threads can execute concurrently and share resources such as memory. A process has always at least one thread (the master thread). A thread is spawned and joined by another thread. Communication among threads on the same node happens via shared memory.
- ***NUMA-architecture***: A collection of NUMA-nodes linked by a point-to-point network. All cores, memory and devices are divided in NUMA-domains.
- ***NUMA-node, NUMA-domain, NUMA-region***: A group of cores and/or memory and/or devices with Uniform Memory Access (UMA). Local memory is accessible to the other NUMA-nodes, but latency is generally higher than to their own local memory. Memory-bandwidth on the other hand can even be higher to other NUMA-nodes than to the one associated to. Not to be confused with the physical node.

10.1.2 Setup

The Dirac-operator in single-precision was implemented as a variant using threads according to the OpenMP standard in openQCD version 1.6 [33]. The performance of the operator was compared to the native implementation without threads. The runs were conducted on two different nodes having two vastly different processors, see tables 6 and 7. On each node, 3 different lattice sizes were considered; 64^4, 32^4 and 16^4. For each lattice size many configurations (n_r, n_{th}) were analysed, n_r being the number of ranks and n_{th} the number of threads. The configurations were chosen such that the machine was utilized half, fully or over-subscripted. Every configuration was executed 10 times, the average is quoted in the table. The exported OpenMP environment was:

```
# each place corresponds to a core, threads are bound to cores
OMP_PLACES=cores
# block cyclic scheduling, where each thread gets floor(#iteration/#threads)
# contiguous iterations. In multiple parallel for-loops, all with the same
# scheduling policy and the same limits, the same threads get the same loop-slices
OMP_SCHEDULE=static
# assignment of threads to places goes successively through the available places
# as close to the master thread as possible
OMP_PROC_BIND=close
```

Listing 3: Environment variables for OpenMP.

The runs were disposed using the mentioned environment variables and Slurms CPU-bind option to pin processes (and its threads) to NUMA-domains. Threads of the same rank were closely bound to cores within the same NUMA-regions whereas ranks were spread among NUMA-regions.

10.1.3 Discussion

Table 6: In table 6 the number of physical and logical cores was a power of 2, therefore in both cases – native and with threads – all logical cores on the node could be utilized. In general, the best run-times were achieved with the native operator. The only exception for the 16^4 lattice was the run with 64 ranks and 2 threads per rank. With that lattice size the entire gauge-field configuration fits into L2 cache of one processor (see table 8), making consecutive runs very fast. Especially for the variant with threads, with twice as many threads the problem is not solved in half the time. This suggests that the overhead for dealing with threads is not negligible, and certainly needs improvement. It should be seen as a first working step towards hyper-threading. On the other hand, the native implementation lacks an ideal strong scaling, which can be seen by looking at the first two rows. Generally, the runs with only 128 ranks are faster than the runs using every of the 256 logical cores. This holds for all considered lattice sizes. Let's concentrate on the most interesting lattice size 64^4. The variant with threads brings no benefit compared to the native one, but there are configurations that are comparable in run-time, namely the ones with $(n_r, n_{th}) = (64, 4)$ as well as $(32, 8)$. Both make use of all logical cores on the node. Improving the thread implementation can therefore lead to a speedup increase (see conclusion). Generally, the less ranks, the worse the

operator	n_r	n_{th}	64^4 time [s]	32^4 time [ms]	16^4 time [ms]
native	256	N/A	1.06 ± 0.02	80.0 ± 2.0	9.53 ± 0.33
native	128	N/A	1.06 ± 0.01	65.5 ± 1.3	6.63 ± 0.73
omp	128	1	1.12 ± 0.06	69.2 ± 4.3	6.32 ± 0.13
omp	128	2	1.08 ± 0.01	71.8 ± 6.0	7.78 ± 0.37
omp	128	4	1.09 ± 0.01	70.6 ± 1.5	9.90 ± 2.68
omp	64	1	1.54 ± 0.08	93.0 ± 1.2	7.14 ± 0.17
omp	64	2	1.09 ± 0.00	66.6 ± 3.9	5.99 ± 0.34
omp	64	4	1.07 ± 0.01	67.9 ± 1.3	8.39 ± 0.40
omp	64	8	1.08 ± 0.00	70.7 ± 1.7	10.9 ± 1.5
omp	32	2	1.51 ± 0.00	94.1 ± 0.3	7.97 ± 0.13
omp	32	4	1.09 ± 0.00	68.3 ± 0.1	6.87 ± 0.24
omp	32	8	1.06 ± 0.01	72.6 ± 0.8	9.12 ± 1.94
omp	32	16	1.09 ± 0.00	80.2 ± 6.3	11.7 ± 0.7
omp	16	4	1.60 ± 0.01	101 ± 6	8.83 ± 0.10
omp	16	8	1.15 ± 0.00	79.9 ± 0.8	8.39 ± 0.13
omp	16	16	1.13 ± 0.01	88.8 ± 1.7	10.9 ± 1.0
omp	16	32	1.14 ± 0.01	92.7 ± 1.8	14.3 ± 0.5
omp	8	8	1.88 ± 0.00	131 ± 7	11.6 ± 0.4
omp	8	16	1.26 ± 0.01	102 ± 3	10.7 ± 0.1
omp	8	32	1.23 ± 0.01	110 ± 7	13.5 ± 0.8
omp	8	64	1.30 ± 0.03	111 ± 1	19.7 ± 0.6
omp	4	16	2.01 ± 0.01	150 ± 0	14.3 ± 0.1
omp	4	32	1.38 ± 0.00	124 ± 1	14.8 ± 0.4
omp	4	64	1.36 ± 0.02	133 ± 2	19.4 ± 0.4
omp	4	128	1.44 ± 0.01	134 ± 4	25.8 ± 0.7
omp	2	32	2.05 ± 0.03	169 ± 6	19.5 ± 0.4
omp	2	64	1.50 ± 0.05	147 ± 1	19.6 ± 0.6
omp	2	128	1.43 ± 0.01	167 ± 3	25.3 ± 1.4
omp	2	256	1.49 ± 0.03	165 ± 7	34.0 ± 1.3
omp	1	64	2.17 ± 0.00	142 ± 0	12.5 ± 0.2
omp	1	128	2.18 ± 0.01	144 ± 3	15.2 ± 0.5
omp	1	256	2.19 ± 0.01	157 ± 6	17.4 ± 0.8
omp	1	512	2.35 ± 0.01	171 ± 3	53.4 ± 5.3

Table 6: 10 consecutive invocations of the single precision Dirac-operator `Dw()` on random fields in openQCD version 1.6. The table shows runs with different rank/thread configurations (n_r, n_{th}) on different lattice sizes. The lattice size refers to the total lattice size on the node. The node consisted of 2 NUMA-nodes, 2 sockets each with an AMD EPYC 7742 processor @ 2.25GHz with 64 physical cores per socket and 2 threads per core; in total 256 logical cores, with 512GB of memory. The cache sizes of one processor are L1 data: 2MB (64×32KB), L1 instruction: 2MB (64×32KB), L2: 32MB (64×512KB), L3: 256MB (16×16MB).

operator	n_r	n_{th}	64^4 time [s]	32^4 time [ms]	16^4 time [ms]
native	16	N/A	7.33 ± 0.05	450 ± 3	28.2 ± 0.1
native	32	N/A	8.64 ± 0.01	538 ± 2	30.3 ± 0.1
native	64	N/A	4.41 ± 0.01	280 ± 0	21.6 ± 7.5
omp	64	1	4.72 ± 0.00	298 ± 0	25.1 ± 14.3
omp	64	2	4.74 ± 0.01	304 ± 14	24.3 ± 6.2
omp	64	4	4.75 ± 0.00	300 ± 0	22.0 ± 5.0
omp	64	8	4.77 ± 0.00	303 ± 0	24.9 ± 4.0
omp	32	1	9.33 ± 0.03	581 ± 1	33.7 ± 0.1
omp	32	2	9.34 ± 0.02	583 ± 1	34.2 ± 0.2
omp	32	4	9.36 ± 0.01	584 ± 1	35.4 ± 0.1
omp	32	7	9.39 ± 0.01	590 ± 0	37.1 ± 0.1
omp	16	2	4.29 ± 0.02	269 ± 2	18.8 ± 0.1
omp	16	4	4.62 ± 0.02	290 ± 0	18.3 ± 0.1
omp	16	7	2.84 ± 0.01	190 ± 1	14.1 ± 0.1
omp	16	14	4.04 ± 0.00	259 ± 1	20.8 ± 0.1
omp	8	4	4.55 ± 0.04	299 ± 2	21.9 ± 0.0
omp	8	8	4.81 ± 0.01	318 ± 0	21.7 ± 0.1
omp	8	14	3.08 ± 0.01	217 ± 9	17.7 ± 0.5
omp	8	28	3.10 ± 0.01	217 ± 0	20.6 ± 0.1
omp	4	7	5.29 ± 0.03	353 ± 1	25.2 ± 0.1
omp	4	14	3.09 ± 0.02	225 ± 8	18.0 ± 0.1
omp	4	28	3.34 ± 0.09	238 ± 1	20.0 ± 0.2
omp	4	56	3.32 ± 0.01	244 ± 1	25.3 ± 0.1
omp	2	28	4.87 ± 0.01	320 ± 0	21.8 ± 0.1
omp	2	56	4.75 ± 0.01	326 ± 1	23.7 ± 0.1
omp	2	112	4.73 ± 0.01	340 ± 2	33.7 ± 1.2
omp	1	56	5.50 ± 0.01	301 ± 1	17.3 ± 0.1
omp	1	112	4.98 ± 0.01	314 ± 1	18.6 ± 0.2
omp	1	224	5.13 ± 0.01	338 ± 2	36.3 ± 0.2

Table 7: 10 consecutive invocations of the single precision Dirac-operator `Dw()` on random fields in openQCD version 1.6. The table shows runs with different rank/thread configurations (n_r, n_{th}) on different lattice sizes. The lattice size refers to the total lattice size on the node. The node consisted of 4 NUMA-nodes, 4 sockets each with an Intel(R) E7-4830 v4 @ 2.00GHz with 14 physical cores per socket and 2 threads per core; in total 112 logical cores, with 2TB of memory. The cache sizes of one processor are L1 data: 448KB (14×32KB), L1 instruction: 448KB (14×32KB), L2: 3.5MB (14×256KB), L3: 35MB (1×35MB).

performance becomes. This can be explained by false sharing among the threads and ignorance concerning NUMA-effects when implementing the operator with threads.

Table 7: Table 7 shows a vastly different picture: most notable for the analysis is the fact that the total number of physical and logical cores was *not* a power of 2 on this node. This makes it difficult to utilize all resources using the native implementation. This fact is reflected by the results. The variant using threads has substantial performance increase compared to the best native pure-MPI implementation using 64 ranks. Especially the configurations with ranks and threads $(n_r, n_{th}) = (16, 7)$, $(8, 14)$ and $(4, 14)$ perform best. The reason for this lies in the non-trivial association of cores to NUMA-domains. The node has 4 NUMA-domains, where the logical cores are associated to NUMA-domains in a non-trival manner (see listing 4). 14 logical cores share an L3 cache. This explains why the mentioned configurations with 14 threads perform well. The jobs were disposed in a way that the 14 threads are bound to physical cores with the same L3 cache, thus cores in the same NUMA-domain. The job $(16, 7)$ that performed best was disposed such that each of the 4 NUMA-domains contained a group of 4 consecutive ranks with 7 threads each. All 112 logical cores were utilized.

```
$ lscpu -e
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE
0   0    0      0    0:0:0:0       yes
1   0    0      1    1:1:1:0       yes
2   0    0      2    2:2:2:0       yes
3   0    0      3    3:3:3:0       yes
4   0    0      4    4:4:4:0       yes
[...]
13  0    0      13   13:13:13:0    yes
14  1    1      14   14:14:14:1    yes
[...]
27  1    1      27   27:27:27:1    yes
28  2    2      28   28:28:28:2    yes
[...]
41  2    2      41   41:41:41:2    yes
42  3    3      42   42:42:42:3    yes
[...]
51  3    3      51   51:51:51:3    yes
52  3    3      52   52:52:52:3    yes
53  3    3      53   53:53:53:3    yes
54  3    3      54   54:54:54:3    yes
55  3    3      55   55:55:55:3    yes
56  0    0      0    0:0:0:0       yes
57  0    0      1    1:1:1:0       yes
58  0    0      2    2:2:2:0       yes
59  0    0      3    3:3:3:0       yes
60  0    0      4    4:4:4:0       yes
[...]
69  0    0      13   13:13:13:0    yes
70  1    1      14   14:14:14:1    yes
[...]
83  1    1      27   27:27:27:1    yes
84  2    2      28   28:28:28:2    yes
[...]
97  2    2      41   41:41:41:2    yes
98  3    3      42   42:42:42:3    yes
[...]
107 3    3      51   51:51:51:3    yes
108 3    3      52   52:52:52:3    yes
109 3    3      53   53:53:53:3    yes
110 3    3      54   54:54:54:3    yes
111 3    3      55   55:55:55:3    yes
```

Listing 4: Output of `lscpu -e`. Notice the association of logical cores (first column) to NUMA-domains (second column). The fourth column denotes the physical core. Each NUMA-domain has 14 logical cores, where core 0 and 56 belong to physical core 0 sharing their L1 and L2 cache on NUMA-node 0, logical cores 1 and 57 belong to physical core 1 on NUMA-node 0 sharing L1 and L2 cache and so on. The L3 cache is shared among logical cores 0-13 and 56-69, 14-27 and 70-83, and so on. One NUMA-domain has thus 14 physical and 28 logical cores.

lattice	gauge-field	quark-field
16^4	18MB	6MB
32^4	288MB	96MB
64^4	4608MB	1536MB

Table 8: Sizes of the gauge- and quark-fields in terms of lattice sizes.

10.1.4 Conclusion

The variant using threads is useful on machines where the number of logical or physical cores is not a power of 2, or can somehow not be fully utilized by the pure-MPI implementation. Since in openQCD (and openQ*D) the number of ranks in one direction can only be $2, 4, 6, \dots$ and the lattice size in one direction can only be $4, 6, 8, \dots$, these numbers are limited for a given lattice size and a processor where the number of CPUs is not a power of 2. Utilizing the entire node is not always possible[65]. On the other hand, the operator with threads can utilize the entire node, because the number of threads per rank is arbitrary.

When implementing threads, it is important to write NUMA-aware code. In the above implementation there are certainly performance degradations due to NUMA-effects and false sharing. It was tried to obey the first-touch policy as often as possible, by initializing the involved arrays with a parallel for-loop, just as they are accessed later in the main for-loop[66] within the Dirac-operator. However, a certain amount of false sharing was not preventable without significantly changing the original code.

Another peculiarity of the above implementation was that in `deo()` the 8 directions of the gauge-fields had to be separately locked in every lattice point, because in these code segments the output vector was written leading to race conditions among threads without locking. The initialization and de-initialization of the `VOLUME/2` locks involve a non-negligible overhead in every call of the Dirac-operator. The overhead can be estimated by comparing the results of $(128, N/A)$ and $(128, 1)$ in table 6 or $(64, N/A)$ and $(64, 1)$ in table 7.

Proposal 10.1: The Dirac-operator with threads

The analysis in section 10.1 shows that a version of the Dirac-operator employing threads can improve performance as well.
For future refinements of the thread implementation one can concentrate on false sharing of the current implementation. For example, in the main loop, instead of consecutively looping over the elements that are being read, loop over the elements that are written (the output spinor) in such a way that only a (consecutive) slice of the output spinor is written in each iteration. Each thread then has its own constant equally sized slice to write to, without having race conditions with other threads. This method avoids expensive locking. It implies to use `schedule(static)` and the same boundaries in all parallel for-loops, such that a thread always gets the same loop slice within different for-loops. The current test-implementation obeys that. A certain amount of false sharing is not preventable without changing the loop as described above. In the parts of the code where the output spinor `r` is written consecutively (`r+i` or `r+VOLUME/2+i`) only true sharing appears, because `r` was initialized in the same way (first touch policy). But within `deo()`, `r` is written on arbitrary offsets, leading to false sharing.
In the main parallel for-loop not all threads have the same amount of workload as implemented currently. This is due to the if-statement inside the loop. Using `schedule(dynamic)` here instead is not recommended, because we want the same threads to get the same data-slices as in previous parallel for-loops to reduce data movement due to cache coherency.
The thread implementation, with which the above results are obtained, was written directly in openQCD and can be found in the GitHub repository ref. [59].

10.2 Dirac-operator representations

It makes sense to look at different representations of the gauge-fields $U_\mu(n)$, since on GPUs sparse linear algebra is even more memory-bound than on CPUs. While collecting the data on GPUs in section 8.6, the GPU load was never higher than 20-30%. If we achieve higher arithmetic intensities in the calculations and less memory traffic in application kernels such as the Dirac-operator, we can achieve a 2-4 times faster GPU-implementation than the reference implementation in Python that was used in the analysis. This gives motivation to the following proposal and definition.

[65]For example the node in table 7, where only a maximum of 64 from a total of 112 logical cores could be utilized.
[66]See line 1295ff in `modules/dirac/Dw.c` in [59]

Definition 10.1 (Arithmetic Intensity)**.** *The* ***arithmetic intensity*** *I is the ratio of the* ***work*** *W and the* ***memory traffic*** *T appearing in a considered piece of code,*

$$I = \frac{W}{T}.$$

The work W needs to be given as number of floating-point operations and the memory traffic T in terms of stored and loaded bytes. The unit of arithmetic intensity I is then floating-point operations per byte and depends on the data type of the involved quantities.

Proposal 10.2: Representation of the Dirac-operator

For the implementation of the Dirac-operator on the GPU, the software library QUDA [45] is a good sample. To improve the performance of their Dirac-operator, the authors of QUDA used a representation of the $SU(3)$-fields with 8 real numbers, a gauge transformation to make almost all of the gauge fields in temporal direction to the identity-matrix and a change of basis in the γ-matrices, such that one of the four matrices has a quite simple form. The most interesting one is the $SU(3)$-representation with only 8 real numbers. In openQ*D the struct `su3_dble`[a] representing an $SU(3)$-gauge-field consists of 18 double precision numbers. The C-macro for a matrix-matrix multiplication of 2 such structs, `_su3_times_su3()`[b], consists of $18 \cdot 12$ floating-point operations, $2 \cdot 18$ loads and 18 stores. Using binary64, the arithmetic intensity is $I = 0.5$ floating-point operations per byte, making the problem memory-bound. Since the rows and columns of $SU(3)$-matrices form an orthonormal basis of $\mathbb{C}^3$, one representation of such matrices consists of only the first two rows/columns and the third row/column is calculated as the vector product of the former two [60]. This is a representation with 12 real numbers. A matrix-matrix multiplication of two such matrices ends up in 270 floating-point operations, $2 \cdot 12$ loads and 12 stores. This results in an arithmetic intensity of $I = 0.9375$ floating-point operations per byte using binary64 – still memory-bound.
If the representation of a $SU(3)$-gauge-field would be chosen such that the struct contains only 10 numbers [61], then a matrix $A \in SU(3)$ would be represented as ($a_{ij} \in \mathbb{C}$)

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & Na_{31}^* & Na_{21} \\ 0 & -Na_{21}^* & Na_{31} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & -Na_{13}^* & -Na_{12}^* \\ \frac{1}{N} & -Na_{11}^* a_{12} & -Na_{11}^* a_{13} \end{pmatrix}$$
$$= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & -N^2 \left(a_{13} a_{31}^* + a_{11}^* a_{12} a_{21}\right) & -N^2 \left(a_{12}^* a_{31}^* + a_{11}^* a_{13} a_{21}\right) \\ a_{31} & -N^2 \left(a_{13}^* a_{21}^* + a_{11}^* a_{12} a_{31}\right) & -N^2 \left(a_{12}^* a_{21}^* + a_{11}^* a_{13} a_{31}\right), \end{pmatrix}$$

where $N := \left(1 - |a_{11}|^2\right)^{-\frac{1}{2}}$. Notice that this representation has a singularity at $|a_{11}| = 1$. The 5 complex numbers $a_{11}, a_{12}, a_{13}, a_{21}, a_{31}$ are subject of the orthonormality constraints

$$|a_{11}|^2 + |a_{12}|^2 + |a_{13}|^2 = |a_{11}|^2 + |a_{21}|^2 + |a_{31}|^2 = 1, \tag{10.1}$$

leading to the observation that all 10 real numbers are in the interval $[-1, 1]$.
Finally, a minimal representation of 8 real numbers [61] can be obtained using the constraint above (10.1). If we write $a_{ij} = x_{ij} + iy_{ij}$ with $x_{ij}, y_{ij} \in \mathbb{R}$ we can eliminate 2 further numbers. One (of many) choices could be

$$y_{31} = \sqrt{1 - |a_{11}|^2 - |a_{21}|^2 - |x_{31}|^2}, \tag{10.2}$$
$$y_{13} = \sqrt{1 - |a_{11}|^2 - |a_{13}|^2 - |x_{13}|^2}. \tag{10.3}$$

Using this, only a_{11}, a_{12}, a_{21} and the real parts of a_{13} and a_{31} need to be stored in memory. For a breakdown of the different arithmetic intensities of the representations, see table 9.

Arithmetic intensities in floating-point operations per byte			
Reals	I(binary64)	I(binary32)	I(binary16)
18	0.5	1	2
12	0.9375	1.875	3.75
10	1.3667	2.7333	5.4667
8	1.9115	3.8229	7.6458

Table 9: Arithmetic intensities of $SU(3)$ representations with different requirement for real numbers for a matrix-matrix multiplication, `_su3_times_su3()`. In the calculation of the intensities a FLOP-count of 6 was used for the square root of a floating-point number and previous results were reused instead of recalculated.

In table 10 one invocation of `Dw_dble()` was called and the macros from `su3.h` where counted.

One invocation of `Dw_dble()`					
# calls	macro name	I(18)	I(12)	I(10)	I(8)
61440	`_vector_add_assign()`	0.0416	0.0416	0.0416	0.0416
24576	`_su3_multiply()`	0.3	0.625	0.9323	1.0989
24576	`_su3_inverse_multiply()`	0.3	0.625	0.9323	1.0989
12288	`_vector_add()`	0.0416	0.0416	0.0416	0.0416
12288	`_vector_i_add()`	0.0416	0.0416	0.0416	0.0416
12288	`_vector_i_add_assign()`	0.0416	0.0416	0.0416	0.0416
12288	`_vector_sub()`	0.0416	0.0416	0.0416	0.0416
12288	`_vector_i_sub()`	0.0416	0.0416	0.0416	0.0416
12288	`_vector_sub_assign()`	0.0416	0.0416	0.0416	0.0416
12288	`_vector_i_sub_assign()`	0.0416	0.0416	0.0416	0.0416

Table 10: C-macro calls for one call of `Dw_dble()` on a 8^4 local lattice with 4 ranks. Notice that the relevant macros for improvement are `_su3_multiply()` and `_su3_inverse_multiply()`. All other macros are variants of axpy and cannot be improved unless the representation of spinor-fields would be changed as well.

When modifying the C structs `su3` or `su3_dble`, one has to search the entire codebase for typecasts of these structs to other types, because such casts rely on guarantees of the ANSI C standard about memory layouts of fields within the struct. When changing the struct, the layout in memory is altered as well, making the typecast to another (unchanged) struct faulty, [62].

[a]See line 43 in `include/su3.h` in [17].
[b]See line 490ff in `include/su3.h` in [17].

11 GPU Implementation

There are multiple possibilities how to implement GPU-utilisation into the current state of the code. The resources of the GPU could be used to increase the number of ranks and thus increase the total lattice volume (see proposal 11.1). Motivated by the results in section 8.6, a pure-GPU implementation of all solvers (see proposal 11.2) or a hybrid variant similar to the one in the results could be considered (see proposal 11.3). Finally, a completely different approach harnessing both compute devices at the same time is presented in proposal 11.4.

Proposal 11.1: GPU Implementation Variant 1 - GPU ranks

Keeping in mind the results of section 8.6, the pure GPU implementation of `SAP_GCR` was by far the fastest. Thus, it makes sense to associate a local lattice to each GPU as well, to treat a GPU as an additional rank with its own local lattice. Each GPU would then act as another rank and the full lattice can be extended by as many local lattices as there are available GPUs. Some nodes might have GPUs, some not. Each GPU has a ***bystander process*** running on the CPU. Process contexts associated to a GPU (CPU) are from now on called ***GPU ranks*** (***CPU ranks***). The bystander process will not require a lot of CPU load, since it only hosts control and informational variables and is not involved in any calculations. The process will be in sleeping state almost all of the time when waiting for the GPU-calculation to complete. The communication from a GPU rank to another rank (CPU or GPU) in the system can be done either via the bystander process or via direct injection into the network bypassing the CPU and the bystander process. The former case will not require a change of the MPI communication code among ranks. Since the communication will only take negligible computational effort on the bystander process, it will not *steal* from the CPU ranks running on the same node. As an example: Let's assume the application runs on one single node with 8 cores and 4 GPUs attached to it. This would involve 8 CPU ranks (separate processes) and 4 GPU ranks with in total 4 (separate) bystander processes. Technically the machine is over-subscripted, and the operating system needs to schedule the processes. But since the bystander processes will be in sleeping state most of the time, this will not (or only negligibly) degrade the performance of the 8 CPU ranks.

- Advantage: The application can run on hybrid machines in the sense that some nodes can have GPUs attached to them while others have none.
- Advantage: Both – the CPU and the GPU – will be utilized *at the same time* since the ranks execute concurrently.
- Disadvantage: The full system consists of two types of ranks – GPU ranks and CPU ranks. Inevitably, either the GPU or the CPU ranks will be faster on the same local lattice size. This means that either the GPUs or the CPUs will have to wait for the others to finish in MPI-barriers or synchronzation points. Utilisation of resources is not perfect unless the local lattices sizes are perfectly concerted. This will require that GPU- and CPU-rank will not host the same local lattice sizes, making this implementation more challenging to write.

Proposal 11.2: GPU Implementation Variant 2 - pure-GPU solvers

Since the GPUs are fast on the solver algorithms, a pure-GPU implementation of all currently implemented solvers[a] can lead to a significant speed up of the program. The Dirac-operator must be held in main memory as well as in GPU memory.

- Advantage: Only a small subset of solver algorithm code has to be changed.
- Advantage: Significant speedups with little effort.
- Disadvantage: The Dirac-operator has to be held in the main memory as well as in the GPU memory. Both need to be in sync. This will lead to a lot of traffic from main memory to the GPU, which is usually slower than GPU-internal traffic. The Dirac-operator is stored in terms of its gauge-fields. They will be redundant and thus the full amount of GPU and CPU memory us not optimally utilised.
- Disadvantage: Since the solvers run on the GPUs, the CPUs will be stale during that time (although the waiting time will be smaller than the time, they would need to run the solvers themselves). Since the HMC-part is still performed on the CPUs, the GPUs will be stale meanwhile. Again, the full potential of performance is not utilised.

[a] CGNE, MSCG, SAP+GCR and DFL+SAP+GCR

Proposal 11.3: GPU Implementation Variant 3 - Hybrid SAP+GCR

If the problem is split among CPU ranks and GPU ranks, one will always have the waiting problem in the sense that either the CPU or the GPU ranks will have to wait for the others to complete (see proposals 11.1 and 11.2). In order for the GPU to speed up the process, every rank should receive the same amount of help from the GPUs. Every rank then solves the same problem faster than before. This is possible if all nodes participating in the calculation share the same specifications; the same number of GPUs (not zero) and the same amount of memory. Let's assume this is given. Starting with a hybrid implementation as in proposal 8.1, only the blocked problems are solved on the GPU and the internal color boundary operator as well as the full Dirac-operator are performed on the CPU as usual (see figures 18 - 22, diamonds; ◇, ◇, ◇). This means that *all* blocked problems of *all* ranks on a single node are solved (or rather `nmr` MR-steps are performed) on the GPU(s) of that node. Within this time the CPUs are stale. To solve this problem, we can use the fact that all blocks of the same color are independent of each other; the order of solving can be arbitrary. So, not all blocks need to be transferred to the GPU – some of them can still reside and be processed in the current rank on the CPU. The work here should be divided such that both – CPU and GPU – need approximately the same amount of time for the processing of their blocks. The question on *how* to divide the blocks still remains. The GPU might be able to process more blocks than a CPU rank in the same time, but this highly depends on its occupation. A robust solution might implement the division of blocks in an adaptive manner.

- Advantage: This proposal can be a good starting point from where to go further.
- Disadvantage: The GPU is only utilised in one part (although the main part) of one solver algorithm, else the GPU is stale.

All mentioned variants so far include the waiting problem; either the CPU or the GPU is not utilized completely. To achieve workload balancing the system needs to avoid idle time on the CPU and GPU devices, the average load thus remains low. Utilizing more than one kind of processing unit *at the same time* goes by the name of ***heterogeneous computing*** [63]. With proposal 11.4, which is probably the most versatile one, a worker queue is included. There are several approaches to implement a workload dividing system based on contention [64], problem size [65], location of data [66, 67], performance of past iterations [68, 69, 70, 71], machine learning [72], on-demand allocation of load balancing [73, 74], processing power [75, 76], priority scheduling [77] or a performance/execution model [67, 78]. Since the involved kernels are mostly memory-bound it makes sense to consider workload division as a function of the data location. It makes sense that idle devices steal chunks of work from others, even when this involves data transfer. This motivates the last proposal.

Proposal 11.4: GPU Implementation Variant 4 - Heterogeneous Computing

It makes sense to partition the workload into chunks. The size of a chunk should be suitable for all involved processing devices. The chunks are published in a queue and associated to idle workers running on the CPU or on the GPU. Data locality should be considered when distributing the work. Stale workers signal their idling and are associated new work. The queue is managed by one process per node or per NUMA-domain. A node can also have no GPU-accelerator. The participating worker processes/threads are then only of type CPU. When all workers on a node become idle, they may steal chunks from other nodes until all nodes are done and the application hits an MPI-barrier. At this point data still resides in various locations in memory, tracked by the management process(es). In the next iteration of the same or a different kernel, data locality can be exploited from anew when associating chunks to workers.

- Advantage: No waiting problem anymore.
- Advantage: Fully utilized CPU *and* GPU.
- Advantage: This system works on clusters having nodes with and without GPU accelerators – both can contribute to the solution.

- Disadvantage: More communication overhead due to the work-stealing nature.
- Disadvantage: Enormous imlementation effort.
- Disadvantage: This variant needs a lot of testing and researching for an optimal workload division.

12 Summary

This thesis has two major purposes: 1) to investigate solver algorithms and other important application kernels appearing in the lattice QFT application openQ*D and 2) to suggest and demonstrate implementations utilizing the GPU on modern supercomputing facilities. The main motive was an increase of performance in terms of run-time. The essential part of this thesis comprises 4 chapters, each of them dealing with a different application kernel in openQ*D.

The convergence analysis of the CG algorithm in section 7 suggested that common 16-bit data types such as binary16 or bfloat16 appearing on modern GPUs are suitable for an implementation of mixed precision solvers, where most of the iterative process is calculated in reduced precision. We found that the Dirac-operator can be formulated using such reduced data types.

The run-time analysis done for the SAP+GCR algorithm in section 8 showed that large parts or even the complete algorithm can be performed on the GPU and significant performance improvements can be expected therewith. From the collected data, an adaptive variant of the algorithm (independently of the CPU or GPU) can be derived with promising run-times.

A similar run-time analysis was performed for the DFL+SAP+GCR algorithm in section 9. The only new insight compared to the previous numerical results was that the size of the deflation subspace should approximately match the size of the low eigenmode subspace. The theoretical part of the section explains this result and contains a formal proof that deflation is efficient if the operator in consideration has locally coherent low eigenmodes and its spectrum is separable.

In a last analysis addressing the lattice Dirac-operator, section 10, a reference implementation using threads within openQCD was implemented and investigated. On certain NUMA-systems the thread variant outperformed the native one, if the number of threads and ranks was chosen to fit the NUMA-architecture of the node. Finally, different representations for the gauge fields were proposed in order to increase the arithmetic intensity and decrease the memory traffic in the Dirac-kernel, especially suitable for the GPU.

All proposals can easily be accessed by the list of proposals in the appendix D.

13 Outlook

Future research from here on can go in many different directions. Immediate next steps would be to explore the Dirac-operator further. The Python-implementation could be extended to represent the operator not in terms of a CSR-matrix, but instead in terms of different representations for $SU(3)$-matrices (see proposal 10.2). To implement such a representation in openQ*D directly is straight-forward and manageable in effort. Run-time measurements such as the ones in section 8.6 can then be repeated and studied. Such an implementation is expected to speed up the results on the GPU, because the kernels involve less memory traffic and are arithmetically more intense. Investigating how these representations perform on the CPU might be of interest as well.

Pursuing proposal 7.2 further might not be fruitful. It is included in this document only for the sake of completeness.

On the other hand, the OpenMP implementation in section 10.1 can act as a starting point to employ threads and as such exploit architectures involing NUMA. Since `Dw()` is already implemented with threads, it is straightforward to reimplement `Dwee()`, `Dwoo()`, `Dweo()`, `Dwoe()` and `Dwhat()` with threads as well since all problems that might arise are already solved in the reference implementation of `Dw()`. Only the block variants `Dwee_blk()`, ... must be considered separately, because although they have exactly the same structure in the loops, they have a completely different (and even variable) range. A straight parallelisation with OpenMP might work, but there will be false sharing, because the threads will recieve different loop-slices than in previous loops. One has to think about association of threads to blocks.

It is worth to further investigate the thread implementation suggested in proposal 10.1, where we want to loop over the output spinor consecutively instead of looping over quantities that are being read to avoid false sharing. Such a technique introduces a complete reimplementation of the Dirac-operator `Dw()`.

The OpenMP variant can be used to further implement offloading OpenMP directives to the accelerator [79]. This would be a straightforward way to implement a GPU-variant of the operator within affordable time and effort, which could act as a baseline for more sophisticated future GPU-realisations. As a certain advantage of this direction, the code will stay backward compatible since the pragmas will be ignored by the compiler when no OpenMP compiler flags are set.

The GPU implementation variant in proposal 11.4 only scratches the surface on heterogeneous computing that enable utilizing both CPUs and GPUs at the same time. There is a lot of potential in investigating such an implementation further. It is by far the most sophisticated one, but because of time reasons we had to formulate it very vaguely. It certainly needs a lot of testing and numerical measurements. This is a vast and active research field and investigating this direction alone could fill a thesis.

14 Acknowledgements

At this point let me thank my supervisor Prof. Thomas Schulthess for giving me the opportunity to deep dive into this fascinating topic and the co-supervisors Prof. Marina Marinković, Dr. Raffaele Solcà and Dr. Anton Kozhevnikov for treating me as a part of their groups and answering my many questions with patience. I have learned a lot from them. They gave me great support and assistance. It was a pleasure to work with them!

One of the greatest challenges of my thesis although was organizing a meeting date suitable for all participants, a task comparable to a complicated numerical problem with many boundary conditions, that ironically and very accurately resembled the challenges in numerical lattice QFT. On the other hand, when finally a meeting date was fixed, it tended to be as stable as the Bitcoin stock market price.

This section would not be complete if I do not thank the coffee and energy drink manufacturing industry. Without them it would certainly not have been possible to write all of this in 6 months[67].

[67] I'll probably owe them my first heart attack as well.

15 References

[1] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable domain-specific computing," *IEEE Design & Test of Computers*, vol. 28, no. 2, pp. 6–15, 2010.

[2] W. Pan, Z. Li, Y. Zhang, and C. Weng, "The new hardware development trend and the challenges in data management and analysis," *Data Science and Engineering*, vol. 3, no. 3, pp. 263–276, 2018.

[3] "Cloud Tensor Processing Units (TPUs)." https://cloud.google.com/tpu/docs/tpus. Accessed: 2021-06-10.

[4] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[5] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 199–213, 2018.

[6] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 24–35, 2013.

[7] B. Sun, L. Yang, P. Dong, W. Zhang, J. Dong, and C. Young, "Ultra power-efficient cnn domain specific accelerator with 9.3 tops/watt for mobile and embedded applications," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1677–1685, 2018.

[8] J. Fung and S. Mann, "Computer vision signal processing on graphics processing units," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. V–93, IEEE, 2004.

[9] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," *Integration*, vol. 40, no. 1, pp. 3–10, 2007.

[10] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.

[11] D. Steinkraus, I. Buck, and P. Simard, "Using GPUs for machine learning algorithms," in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pp. 1115–1120 Vol. 2, 2005.

[12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," tech. rep., Manubot, 2019.

[13] D. Luebke and M. Harris, "General-purpose computation on graphics hardware," in *Workshop, SIGGRAPH*, vol. 33, 2004.

[14] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science & Engineering*, vol. 13, no. 3, pp. 92–95, 2011.

[15] "Top 500: The List.." https://www.top500.org/. Accessed: 2021-06-07.

[16] J. D. Davis and E. S. Chung, "SpMV: A memory-bound application on the GPU stuck between a rock and a hard place," *Microsoft Research Silicon Valley, Technical Report14 September*, vol. 2012, 2012.

[17] I. Campos, P. Fritzsch, M. Hansen, M. K. Marinkovic, A. Patella, A. Ramos, and N. Tantalo, "openQ*D code: a versatile tool for QCD+QED simulations," *The European Physical Journal C*, vol. 80, no. 3, pp. 1–24, 2020. Accessed: 2021-01-06.

[18] C.-N. Yang and R. L. Mills, "Conservation of isotopic spin and isotopic gauge invariance," *Physical review*, vol. 96, no. 1, p. 191, 1954.

[19] P. Van Nieuwenhuizen and A. Waldron, "A continuous Wick rotation for spinor fields and supersymmetry in Euclidean space," *arXiv preprint hep-th/9611043*, 1996.

[20] K. G. Wilson, "Confinement of quarks," *Physical review D*, vol. 10, no. 8, p. 2445, 1974.

[21] B. Sheikholeslami and R. Wohlert, "Improved continuum limit lattice action for QCD with Wilson fermions," *Nuclear Physics B*, vol. 259, no. 4, pp. 572–596, 1985.

[22] M. Lüscher and P. Weisz, "On-shell improved lattice gauge theories," *Communications in Mathematical Physics*, vol. 97, no. 1, pp. 59–77, 1985.

[23] J. Kogut and L. Susskind, "Hamiltonian formulation of Wilson's lattice gauge theories," *Physical Review D*, vol. 11, no. 2, p. 395, 1975.

[24] Y. Shamir, "Chiral fermions from lattice boundaries," *Nuclear Physics B*, vol. 406, no. 1-2, pp. 90–106, 1993.

[25] H. B. Nielsen and M. Ninomiya, "Absence of neutrinos on a lattice:(i). Proof by homotopy theory," *Nuclear Physics B*, vol. 185, no. 1, pp. 20–40, 1981.

[26] R. Gupta, "Introduction to lattice QCD," *arXiv preprint hep-lat/9807028*, 1998.

[27] F. Fucito, E. Marinari, G. Parisi, and C. Rebbi, "A proposal for Monte Carlo simulations of fermionic systems," *Nuclear Physics B*, vol. 180, no. 3, pp. 369–377, 1981.

[28] J. A. Finkler and S. Goedecker, "Funnel hopping monte carlo: An efficient method to overcome broken ergodicity," *The Journal of chemical physics*, vol. 152, no. 16, p. 164106, 2020.

[29] S. L. Adler, "Stochastic algorithm corresponding to a general linear iterative process," *Physical review letters*, vol. 60, no. 13, p. 1243, 1988.

[30] S. Gupta, A. Irbäc, F. Karsch, and B. Petersson, "The acceptance probability in the hybrid monte carlo method," *Physics Letters B*, vol. 242, no. 3-4, pp. 437–443, 1990.

[31] J. E. Gentle, *Random number generation and Monte Carlo methods*, vol. 381. Springer, 2003.

[32] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth, "Hybrid monte carlo," *Physics letters B*, vol. 195, no. 2, pp. 216–222, 1987.

[33] M. Lüscher and S. Schaefer, "openQCD simulation program for lattice QCD with open boundary conditions," 2013.

[34] Institute of Electrical and Electronics Engineers. Computer Society. Standards Committee and Stevenson, David, *IEEE standard for binary floating-point arithmetic*. IEEE, 1985.

[35] Institute of Electrical and Electronics Engineers. Computer Society. Standards Committee and Stevenson, David, *IEEE standard for binary floating-point arithmetic*. IEEE, 2008.

[36] S. Wang and P. Kanwar, "Bfloat16: the secret to high performance on cloud TPUs," *Google Cloud Blog*, 2019.

[37] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy, "NVIDIA ampere architecture in-depth," *NVIDIA blog: https://devblogs. nvidia. com/nvidia-ampere-architecture-in-depth*, 2020.

[38] I. Buck, "Taking the plunge into GPU computing," *GPU Gems*, vol. 2, pp. 509–519, 2005.

[39] W. Cody, "Towards sensible floating-point arithmetic," tech. rep., Argonne National Lab., IL (USA), 1980.

[40] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[41] J. R. Shewchuk *et al.*, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[42] D. Göddeke, R. Strzodka, and S. Turek, *Accelerating double precision FEM simulations with GPUs.* Univ., 2005.

[43] P. W. Group *et al.*, "Posit standard documentation - Release 3.2-draft," *Posit Standard Documentation*, 2018.

[44] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," tech. rep., Citeseer, 2008.

[45] 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," *Computer Physics Communications*, vol. 181, no. 9, pp. 1517–1528, 2010.

[46] M. Lüscher, "Solution of the Dirac equation in lattice QCD using a domain decomposition method," *Computer physics communications*, vol. 156, no. 3, pp. 209–220, 2004.

[47] M. Lüscher, "Local coherence and deflation of the low quark modes in lattice QCD," *Journal of High Energy Physics*, vol. 2007, no. 07, p. 081, 2007.

[48] C. Vuik, "New insights in GMRES-like methods with variable preconditioners," *Journal of computational and applied mathematics*, vol. 61, no. 2, pp. 189–204, 1995.

[49] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[50] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.

[51] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[52] R. Gruber, "Adaptive branch in GitLab repository: adaptive implementation SAP+GCR." https://gitlab.com/roman.gruber/openQxD/-/tree/adaptive_sap_gcr, 2021. Accessed: 2021-06-03.

[53] T. Banks and A. Casher, "Chiral symmetry breaking in confining theories," *Nuclear Physics B*, vol. 169, no. 1-2, pp. 103–125, 1980.

[54] L. Giusti, C. Hoelbling, M. Lüscher, and H. Wittig, "Numerical techniques for lattice QCD in the ϵ-regime," *Computer physics communications*, vol. 153, no. 1, pp. 31–51, 2003.

[55] E. F. Kaasschieter, "Preconditioned conjugate gradients for solving singular systems," *Journal of Computational and Applied mathematics*, vol. 24, no. 1-2, pp. 265–275, 1988.

[56] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," 2015.

[57] D. W. Walker and J. J. Dongarra, "MPI: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[58] M. K. Chan and L. Yang, "Comparative Analysis of OpenMP and MPI on Multi-Core Architecture," in *Proceedings of the 44th Annual Simulation Symposium*, ANSS '11, (San Diego, CA, USA), p. 18–25, Society for Computer Simulation International, 2011.

[59] D. N. Roman Gruber, Michele Mesti, "OpenMP branch in GitHub repository: OpenMP implementation of Dw()." https://github.com/chaoos/openqcd/tree/omp, 2021. Accessed: 2021-05-31.

[60] P. De Forcrand, D. Lellouch, and C. Roiesnel, "Optimizing a lattice QCD simulation program," *Journal of Computational Physics*, vol. 59, no. 2, pp. 324–330, 1985.

[61] B. Bunk and R. Sommer, "An 8 parameter representation of SU(3) matrices and its application for simulating lattice QCD," *Computer physics communications*, vol. 40, no. 2-3, pp. 229–232, 1986.

[62] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps, "Coping with type casts in C," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 180–198, 1999.

[63] A. Shan, "Heterogeneous processing: a strategy for augmenting moore's law," *Linux Journal*, vol. 2006, no. 142, p. 7, 2006.

[64] C. Gregg, J. Brantley, and K. Hazelwood, "Contention-aware scheduling of parallel code for heterogeneous systems," in *2nd USENIX workshop on hot topics in parallelism, HotPar, Berkeley, CA*, 2010.

[65] S. Ding, J. He, H. Yan, and T. Suel, "Using graphics processors for high performance IR query processing," in *Proceedings of the 18th international conference on World wide web*, pp. 421–430, 2009.

[66] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pp. 82–91, 2010.

[67] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[68] L. Li, X. Li, G. Tan, M. Chen, and P. Zhang, "Experience of parallelizing cryo-EM 3D reconstruction on a CPU-GPU heterogeneous system," in *Proceedings of the 20th international symposium on High performance distributed computing*, pp. 195–204, 2011.

[69] H. J. Choi, D. O. Son, S. G. Kang, J. M. Kim, H.-H. Lee, and C. H. Kim, "An efficient scheduling scheme using estimated execution time for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 65, no. 2, pp. 886–902, 2013.

[70] G. Bernabé, J. Cuenca, and D. Giménez, "Optimization techniques for 3D-FWT on systems with manycore GPUs and multicore CPUs," *Procedia Computer Science*, vol. 18, pp. 319–328, 2013.

[71] M. E. Belviranli, L. N. Bhuyan, and R. Gupta, "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–20, 2013.

[72] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *International conference on compiler construction*, pp. 286–305, Springer, 2011.

[73] J. I. Agulleiro, F. Vazquez, E. M. Garzon, and J. J. Fernandez, "Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction," *Ultramicroscopy*, vol. 115, pp. 109–114, 2012.

[74] G. Teodoro, T. M. Kurc, T. Pan, L. A. Cooper, J. Kong, P. Widener, and J. H. Saltz, "Accelerating large scale image analyses on parallel, CPU-GPU equipped systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 1093–1104, IEEE, 2012.

[75] F. Lu, J. Song, X. Cao, and X. Zhu, "CPU/GPU computing for long-wave radiation physics on large GPU clusters," *Computers & Geosciences*, vol. 41, pp. 47–55, 2012.

[76] K. A. Hawick and D. P. Playne, "Parallel algorithms for hybrid multicore cpu-gpu implementations of component labelling in critical phase models," in *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'13)', number CSTN-177, WorldComp, Las Vegas, USA, p. PDP3297*, 2013.

[77] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[78] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *2008 IEEE international symposium on parallel and distributed processing*, pp. 1–10, IEEE, 2008.

[79] "OpenMP Accelerator Support for GPUs." https://www.openmp.org/updates/openmp-accelerator-support-gpus/. Accessed: 2021-06-07.

[80] R. Gruber, "GitHub repository: Source code of all implementations." https://github.com/chaoos/master-thesis-code, 2021. Accessed: 2021-05-31.

[81] R. Gruber, "GitHub repository: Collected raw data." https://github.com/chaoos/master-thesis-data, 2021. Accessed: 2021-05-31.

[82] R. Gruber, "GitHub repository: Tex document." https://github.com/chaoos/master-thesis, 2021. Accessed: 2021-05-31.

Appendices

A Inverse Power Method

Theorem A.1 (Inverse Power Method)**.** *Let $n \in \mathbb{N}$, A be an invertible $n \times n$-matrix with eigenvalues $|\lambda_1| < |\lambda_2| \leq \cdots \leq |\lambda_n|$ and corresponding eigenvectors $\vec{v}_i$. Let $\vec{b}_0 = \sum_{i=1}^{n} c_i \vec{v}_i$ be an arbitrary (random) vector with support in the smallest eigenspace, ie. $c_1 \neq 0$. Then,*

$$(A^{-1})^k \vec{b}_0 \xrightarrow{k\to\infty} \frac{c_1}{\lambda_1^k} \vec{v}_1$$

is proportional to the eigenvector corresponding to the smallest eigenvalue.

Proof.

$$\begin{aligned}(A^{-1})^k \vec{b}_0 &= \sum_{i=1}^{n} c_i (A^{-1})^k \vec{v}_i \\ &= \sum_{i=1}^{n} \frac{c_i}{\lambda_i^k} \vec{v}_i \\ &= \frac{c_1}{\lambda_1^k} \left(\vec{v}_1 + \sum_{j=2}^{n} \left(\frac{\lambda_1}{\lambda_j} \right)^k \frac{c_j}{c_1} \vec{v}_j \right) .\end{aligned}$$

Since $|\lambda_1| < |\lambda_j|$, for $j \geq 2$, $\left(\frac{\lambda_1}{\lambda_j}\right)^k \to 0$ for $k \to \infty$.

□

Corollary A.2. *Let A be as in theorem A.1, but with m smallest eigenvalues of approximately the same magnitude $|\lambda_1| \approx |\lambda_2| \approx \cdots \approx |\lambda_m| < |\lambda_{m+1}| \leq \cdots \leq |\lambda_n|$. Then,*

$$(A^{-1})^k \vec{b}_0 \xrightarrow{k\to\infty} \sum_{i=1}^{m} d_i \vec{v}_i$$

is a linear combination of eigenvectors corresponding to the m smallest eigenvalues.

Proof. By the same analysis as in the proof of theorem A.1,

$$\begin{aligned}(A^{-1})^k \vec{b}_0 &= \frac{c_1}{\lambda_1^k} \left(\vec{v}_1 + \sum_{j=2}^{m} \underbrace{\left(\frac{\lambda_1}{\lambda_j} \right)^k}_{\approx 1} \frac{c_j}{c_1} \vec{v}_j + \sum_{l=m+1}^{n} \left(\frac{\lambda_1}{\lambda_l} \right)^k \frac{c_l}{c_1} \vec{v}_l \right) \\ &\xrightarrow{k\to\infty} \frac{c_1}{\lambda_1^k} \left(\vec{v}_1 + \sum_{j=2}^{m} \frac{c_j}{c_1} \vec{v}_j \right) .\end{aligned}$$

□

Remark. The inverse power method applied to the Dirac-operator with its separable spectrum will, according to corollary A.2, converge even faster and give reasonable linear combinations of low modes already after few inverse iteration steps.

B Grassmann variables

In order to satisfy the Pauli exclusion principle and thus making fermions anti-commute, we need to introduce a new type of numbers, called Grassmann variables.

Definition B.1 (Grassmann variables)**.** ***Grassmann variables*** *are basis vectors of a vector space. They form an algebra $\mathbb{G}$ over a field $\mathbb{F}$, usually $\mathbb{R}$ or $\mathbb{C}$. Let $\theta, \eta \in \mathbb{G}$ be two Grassmann variables, their defining relation is*

$$\theta\eta = -\eta\theta.$$

Remark. Since they form an algebra, they commute with elements in the field. Let $a \in \mathbb{F}$, then $a\theta = \theta a$.

Remark. Grassmann variables are non-zero roots of zero: $\theta^2 = 0$, for $0 \neq \theta \in \mathbb{G}$.

We have to think about derivatives with respect to Grassmann variables as well as integrals over Grassmann variables, since they appear in the path integral formalism of fermion fields. For this it makes sense to impose some requirements such that the formulation is coherent and leading to the correct quantum theory governed by the Dirac equation. We want the integral over Grassmann variables to be the "inverse of the derivative". The conditions are

1. Linearity of derivatives and integrals,
2. Compose the integral in a way, such that we can use the partial integration formula,
3. Invariance of the integral under shifts of integration variable ($\theta \to \theta' = \theta + \eta$)[68].

Let $a, b \in \mathbb{F}$ and $\theta, \beta \in \mathbb{G}$. The Taylor-expansion of a smooth function of a Grassmann variable $f(\theta)$ can only look like

$$f(\theta) = \begin{cases} a + \beta\theta = a - \theta\beta \\ a + b\theta, \end{cases}$$

because terms $O(\theta^2)$ and higher are zero exactly. Thus, it makes sense to define the ***derivative with respect to a Grassmann variable*** as

$$\frac{d}{d\theta} f(\theta) = \begin{cases} -\beta \\ b, \end{cases}$$

and the ***Grassmann-integral*** of $f(\theta)$ over the complete range of θ as

$$\int d\theta = 0, \qquad \int d\theta\theta = 1, \qquad \int d\theta f(\theta) = \begin{cases} -\beta \\ b. \end{cases}$$

Notice that the derivative $\frac{d}{d\theta}$ as well as the integral measure $d\theta$ are Grassmann variables too.

Proposition B.3 (Grassmann integral formula)**.** *Let B be a complex $n \times n$-matrix and $\theta_i \in \mathbb{G}$, for $i = 1, \ldots, n$, then*

$$\prod_{k=1}^{n} \int d\theta_k^\star d\theta_k \exp\left(- \sum_{i,j=0}^{n} \theta_i^* B_{ij} \theta_j \right) = \det B.$$

Proposition B.4 (Gaussian integral formula)**.** *Let B be a Hermitian, invertible $n \times n$-matrix and $z_i \in \mathbb{C}$, for $i = 1, \ldots, n$, then*

$$\prod_{k=1}^{n} \int dz_k^\star dz_k \exp\left(- \sum_{i,j=0}^{n} z_i^* B_{ij} z_j \right) = \pi^n (\det B)^{-1}.$$

Remark. Notice the strong resemblance between propositions B.3 and B.4.

[68]This implies that the path integral measure is invariant under shifts as well, $\mathcal{D}\theta = \mathcal{D}\theta'$, which is the central ingredient when deriving Ward-Takahashi identities.

C Code

All code produced for this thesis is licenced under the GNU General Public License v3.0 and can be found in the GitHub repository [80], the collected raw data and the document are licenced under the Creative Commons Attribution Share Alike 4.0 International licence and can be found in the GitHub repositories [81] and [82], respectively.

D List of Proposals

Acronyms

BiCGSTAB Biconjugate Gradient Stabilized. 54

BLAS Basic Linear Algebra Subprograms. 37

CG Conjugate Gradient. 4, 21, 54, 83

CGNE Conjugate Gradient on the normal equations. 37, 60, 82

CSR Compressed Sparse Row. 39, 54, 84

CUDA Compute Unified Device Architecture. 54

DFL+SAP+GCR Deflated Generalized Conjugate Residual with Schwarz Alternating Preconditioning. 4, 65, 70, 82, 83

GCR Generalized Conjugate Residual. 49, 54–59, 63

GMRES Generalized Minimum Residual. 54

HMC Hybrid Monte-Carlo. 20, 62

MPI Message Passing Interface. 73

MR Minimal Residual. 54, 59

MSCG Multishift Conjugate Gradient. 52, 82

NaN not a number. 23, 24, 26, 44

NUMA Non-Uniform Memory Access. 73

OpenMP Open Multi-Processing. 73

QCD Quantum chromodynamics. 3

QED Quantum electrodynamics. 19

QQCD quenched QCD. 19

SAP Schwarz Alternating Procedure. 49, 53, 54, 57

SAP+GCR Generalized Conjugate Residual with Schwarz Alternating Preconditioning. 4, 49, 59–61, 63, 70, 73, 82, 83

SF Schrödinger functional. 39, 53

UMA Uniform Memory Access. 74

Glossary

bfloat16 Googles Brain float [36] floating point number representation with encoding in length of 16 bits. 23, 25, 28, 39, 44–46, 48, 49, 59, 83

binary16 IEEE754 2008 [35] conformant floating point number representation with encoding in length of 16 bits, also called half precision. 23, 25, 28, 29, 39, 42–49, 59, 83

binary32 IEEE754 2008 [35] conformant floating point number representation with encoding in length of 32 bits, also called single precision. 21, 23, 25, 28, 29, 37–40, 44–48, 59, 60

binary64 IEEE754 2008 [35] conformant floating point number representation with encoding in length of 64 bits, also called double precision. 21, 23, 25, 37–40, 42, 44–48, 58–60, 79

fused multiply–add A multiply-add operation $a + bc$ in one shot, where the rounding is deferred. 27

posit16 Posit Standard [43] conformant storage format for real number representation with encoding in length of 16 bits and an exponent size of `es=1`. 27, 28, 39, 42–47

posit32 Posit Standard [43] conformant storage format for real number representation with encoding in length of 32 bits and an exponent size of `es=2`. 27, 28, 39, 44–46

posit64 Posit Standard [43] conformant storage format for real number representation with encoding in length of 64 bits and an exponent size of `es=3`. 27

posit8 Posit Standard [43] conformant storage format for real number representation with encoding in length of 8 bits and an exponent size of `es=0`. 27, 39, 43–46

quire Posit Standard [43] conformant special fixed-size data type that can be thought of as a dedicated register that permits dot products, sums, and other operations to be performed with rounding error deferred to the very end of the calculation [40]. 27, 41, 42, 44–47

rank In MPI a process is identified by its rank, with is an integer between $[0, N - 1]$, where N is the size of the MPI process group. 30

sparse matrix A matrix, where most of the entries are 0. 30

tensorfloat32 Nvidias TensorFloat-32 [37] floating point number representation with encoding in length of 32 bits, but only 19 bits are used. 23, 25, 28, 39, 45–47, 49, 59