Linear Algebra on Lattices: Simit Language Extensions with Applications to Lattice QCD

by

Gurtej Kanwar

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Linear Algebra on Lattices: Simit Language Extensions with Applications to Lattice QCD

by

Gurtej Kanwar

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis presents language extensions to Simit, a language for linear algebra on graphs. Currently, Simit doesn't efficiently handle lattice graphs (regular grids). This thesis defines a stencil assembly construct to capture linear algebra on these graphs. A prototype compiler with a Halide backend demonstrates that these extensions capture the full structure of linear algebra applications operating on lattices, are easily schedulable, and achieve comparable performance to existing methods.

Many physical simulations take the form of linear algebra on lattices. This thesis reviews Lattice QCD as a representative example of such a class of applications and identifies the structure of the linear algebra involved. In this application, iterative inversion of the Dirac matrix dominates the runtime, and time-intensive hand-optimization of inverters for specific forms of the matrix limit further research. This thesis implements this computation using the language extensions, while demonstrating competitive performance to existing methods.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Saman Amarasinghe, for his patience and guidance in working with me for the past two years. From my first moments joining the group, having had the barest of brushes with academia, to working through a paper and then a Master's project, Saman has always worked with me to find where I would be happy and helped me get there. I certainly would not have found myself working on significant research projects had it not been for him fostering a warm academic environment.

Besides Saman, I would like to offer my sincerest thanks to Fred Kjolstad, for always being willing to talk to me as a peer, to discuss even my most inane of ideas, and for ensuring that the path I took through this project, and in life, would make me a happier, healthier person. As a future graduate student and a mentor to others down the road, I would be thrilled to even barely match the positive impact Fred has had on me.

I would also like to thank Dr. Andrew Pochinsky for dedicating so many of his hours to helping me gain an understanding of the rich field of Lattice QCD theory and methods. He has always been willing to answer my most basic of questions, and candidly tell me whenever I am wrong on a point, both things that have spurred my understanding far faster than I could have ever done on my own.

I also offer my gratitude to Prof. Will Detmold for providing context on Lattice QCD and the physics of the Standard Model through several conversations and a semester's worth of lectures. Will and Andrew are unquestionably the reason I find myself beginning a career on the "dark side" (theoretical physics) rather than following any of the other more tedious paths I could have taken.

I owe many thanks as well to the folks of the COMMIT group, who were always willing to provide feedback on my ideas. In particular, I would like to thank Shoaib Kamil for finding the time to bring to bear his depth of experience with Halide and stencil computations whenever I was having difficulty, and also for dedicating many hours to providing feedback on the writing of this thesis.

Finally, I owe my sanity during the writing process to Parker Tew and Gaurav Singh, for walking the path with me, and helping me understand that we all felt as lost as I did.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Theoretical physicists have investigated the strong nuclear force through Lattice Quantum Chromodynamics (QCD) calculations since Ken Wilson's initial formulation in 1974 [54]. Today, there are several collaborations and research groups [10, 12, 2, 6] working on generating data ensembles and performing calculations using these generated ensembles. These groups seek to improve our theoretical understanding of nuclear structure and investigate discrepancies between experiment and theory. Both generating ensembles and calculating predictions based on these data require large scale computation, often measured in hundreds of TFlop-years [8], and this constrains the range of physical investigations.

Current Lattice QCD research is based on linear algebra on a 4D lattice. In particular, computations are dominated by iterative inversion of the Dirac matrix, a sparse matrix with values between sites of the lattice and their nearby neighbors. Existing methods use libraries that have been tuned to invert specific forms of the Dirac matrix, corresponding to specific physical investigations. The narrow scope of these operations hinders exploration of a wide variety of physical scenarios: gaining a statistically significant understanding of new physics requires hand-optimizing inversion of each new form to make efficient use of limited computational resources.

This is exactly the form of problem in which Domain-Specific Languages (DSLs) provide an advantage through *flexibility of expression*. In scientific computing, we believe there has so far existed a trade-off between the flexibility of expression offered

by DSLs, and the *targeted performance* offered by optimized libraries. We believe each approach has benefits, and in fact the two can often complement each other, as in cases where DSLs delegate performance-critical evaluation to underlying libraries.

In our investigation of linear algebra on lattices, we find a wealth of library approaches [5, 4, 7, 8], but a lack of flexible, performant language approaches. Motivated by this gap, and specifically by the growing need for a flexible language approach in current theoretical physics investigations, we develop a set of language constructs for linear algebra on lattices (regular grids) that provide an alternative to existing rigid library approaches.

We describe an extension to the Simit language, which is originally designed for linear algebra on arbitrary graph structures, to support lattice graphs efficiently. Lattice graphs have additional structure over arbitrary graphs, and in our extensions we allow the user to identify lattice graphs and use the structure in definitions of matrices. The additional lattice structure also enables efficient compilation by removing memory indices describing graph structure.

By leveraging Halide, an existing stencil pipeline DSL (described in Chapter 3), we build a prototype to evaluate the expressiveness and efficiency of our language. We find that, versus Simit, our language allows simpler description of matrices on lattice graphs and allows compilation of much more efficient code in these graphs. Specifically, this thesis evaluates the performance of code generated from a prototype of this system for two common stencils and in the context of Lattice QCD computations, and shows that it performs better than Simit and comparably to existing optimized USQCD library code. We also demonstrate the ability of our language to be scheduled independently of the algorithm, a key feature that allows quick development of performant, correct code [40]. To this end, our use of Halide as a backend provides a solid stepping stone: its scheduling language allows convenient parallelization, vectorization, and tiling, among other scheduling optimizations.

These strong results are due to the ability of our DSL to combine (1) information on the structure of matrix and vector representations on the lattice, (2) the flexible index expression approach to linear algebra, and (3) the concept of separation of

schedule and algorithm. These features together facilitate a system that generates efficient stencil-based descriptions of linear algebra on lattices and allows an understandable scheduling of the generated code.

We hope that these promising early results will spur adoption of DSL methods in the physics community, and in particular will open doors to new Lattice QCD experiments. We also believe this language is well suited for other computationally intensive physical applications performed on lattices, for example the grid-based hydrodynamics used in astrophysical simulations [53], stencil-based seismic simulations [39, 34], and weather prediction [49, 41]. Finally, this language has applications beyond physics. One particular example of interest is low-level image processing using Gaussian Markov Random Fields on grids. This application is well-described using linear algebra and iterative matrix inversion of matrices with regular stencil structure [52]. Our extensions can be applied to provide a natural, efficient description of these computations.

The language described in this thesis fits well into the existing Simit programming model, and we think a promising avenue forward would include an official extension to Simit based on the concepts presented here, such that Simit may be efficiently applied to all linear algebra on lattices.

Summarizing the main contributions of this thesis, we present:

- An overview of the field of Lattice QCD, with a focus on its computational challenges (Chapter 2)
- A description of an extension to the Simit programming model to support linear algebra on lattices (Chapter 5)
- A detailed design of a prototype compiler which compiles a subset of the existing and new Simit language constructs (Chapter 6)
- An evaluation of the performance of this system on common stencils and in the context of the Lattice QCD application (Chapter 7)

Chapter 2

Lattice QCD Application

In the following, we describe the physical motivation behind Lattice QCD computations (Sections 2.1, 2.2, and 2.3). We then summarize the major computational elements involved in Lattice QCD simulations, and condense this information into an algorithmic listing (Section 2.4). Finally, we identify the set of linear algebraic constructs involved, and demonstrate that they can all be reduced to stencil descriptions (Section 2.5).

2.1 Overview of the Standard Model of Physics

The Standard Model has been enormously successful at describing the majority of small-scale observations about our universe. At the highest level, the Standard Model places fields of various types on a spacetime backdrop and pairs this with quantum mechanics to give us a *quantum field theory* description of particle physics. Using the tools of quantum field theory, one can use the Standard Model to predict properties of multi-particle objects and the outcomes of particle collider experiments. While the Standard Model has matched many experiments to great accuracy, there are observations which do not fit within our understanding of particle physics [35, 23]. A more detailed understanding of the Standard Model as well as physics beyond the Standard Model are both active areas of research.

The Standard Model provides a quantum field theory description of gauge bosons,

Higgs bosons, leptons, and quarks [38]. Quantum chromodynamics (QCD) in particular is the study of gluons, the gauge bosons of the strong force, and their coupling to quarks, the constituent pieces of protons, neutrons, and other more exotic multiparticle objects. Lattice QCD provides one tool to specifically investigate phenomena dominated by the strong force.

The path integral formalism of quantum field theory is particularly useful in developing a description of Lattice QCD [17]. We describe the physical motivations behind the path integral formalism in Appendix A, and simply state the results here: expectation values of quantum *operators*, or "observables", are computed using a functional integral over all possible configurations of fields. Expectation values of operators can be used in a variety of ways to extract physical information [18, Sec III]. The task of making physical predictions therefore reduces to evaluation of this integral.

The path integral evaluation of the expectation value of a particular operator $\mathcal{O}$ is written as a functional integral over all physical fields, ϕ_i:

$$\langle 0|T(\mathcal{O})|0\rangle = \frac{\int \mathcal{D}\phi_i \mathcal{O} e^{iS[\phi_i]}}{\int \mathcal{D}\phi_i e^{iS[\phi_i]}} = \frac{1}{Z}\int \mathcal{D}\phi \mathcal{O} e^{iS[\phi_i]}$$

In this description, the action, $S[\phi_i]$, encodes the physics of the system. The action is typically written as the integral of a "Lagrangian" over all of spacetime. The Lagrangian specifies the *localized* description of the physics: $S[\phi_i] = \int d^4x \mathcal{L}[\phi_i]$. In exploring the strong force, one picks out the pieces of the Standard Model Lagrangian that correspond to the gluon and quark interactions and evaluates the path integral for physically interesting operators using those pieces.

2.2 The Strong Force: Quantum Chromodynamics

Quantum Chromodynamics (QCD) is the theory of the quark and gluon strong interactions in the Standard Model. QCD does *not* include how quarks behave as charged particles or under the weak force. For any complete calculation involving quarks in the Standard Model, we should include both charge and the weak force, but when we focus on certain observables of bound states like nuclei, we find that the contributions

from the electromagnetic and weak forces are small compared to the strong force contribution. As a result we can choose to neglect these effects for broad calculations of nuclear phenomena [18, Sec IV.D]. From here on out, we will proceed with a focus on only the QCD sector of the Standard Model.

2.2.1 The QCD Lagrangian

To begin, we present a compact form for the QCD Lagrangian, for simplicity presented with only one flavor of quark. We can use this in conjunction with the path integral form above to write down expectation values of interest.

$$\mathcal{L}_{\text{QCD}} = \underbrace{-\frac{1}{4}\text{Tr}(F_{\mu\nu}F^{\mu\nu})}_{\text{pure gluon term}} \underbrace{-\bar{\psi}(i[\gamma^\mu(\partial_\mu - igA_\mu)] - m)\psi}_{\text{quark term}}$$

There are a lot of pieces to this Lagrangian. Let's tease them apart individually:

1. $A_\mu(x)$: The *gluon field*, taking vector values in the adjoint representation of SU(3) at every spacetime point. Put more concretely, for every spacetime dimension, $\mu \in t, x, y, z$, $A_\mu(x)$ is an 8-component object, representing the coefficients of the $\mathfrak{su}(3)$ *algebra* generators. Combining the vector and $\mathfrak{su}(3)$ dimensions, $A_\mu(x)$ is concretely a $4*8 = 32$ dimensional object. Appendix B discusses the SU(3) group, and provides an example basis for the $\mathfrak{su}(3)$ generators.

2. $\psi(x)$: The *quark field* for a single flavor, e.g. up quarks. This field takes values in the fundamental representation of SU(3) at every spacetime point: at the top-level it is a 3-vector of values, which are acted on by 3×3 matrices in SU(3) by matrix multiplication. This field contains additional "spinor" substructure. Each value in the SU(3) 3-vector is a complex 4-vector of *anti-commuting* values [55, Sec 9.5]. This anti-commuting nature makes quarks difficult to treat in the path integral, as we shall discuss shortly.

3. $\bar{\psi}(x) = \psi^\dagger(x)\gamma_0$: The quark conjugate field. This conjugation involves a transpose and multiplication with a gamma matrix (described below), resulting in a scalar value when paired with the quark field.

4. $F_{\mu\nu}(x)$: A composite object made up of A_μ values. $F_{\mu\nu}$ also takes values in the adjoint representation of SU(3), for all combinations of $\mu, \nu \in t, x, y, z$. Explicitly, $F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu - ig[A_\mu, A_\nu]$, with the commutator taken over the 3×3 matrix representations of A_μ and A_ν, and g a constant.

5. Tr: The sum over the 8 adjoint representation coefficients of the F^2 term inside. The μ, ν indices inside have an implied summation, resulting in a scalar value overall.

6. $\partial_\mu - igA_\mu$: The "covariant" derivative of SU(3) values. This can intuitively be thought of as incorporating the SU(3) mixing in a spacetime direction, to allow properly taking the difference between infinitesimally close ψ values. Importantly, this results in an interaction between the gluon and quark field.

7. γ^μ: Spinor matrices, having 4×4 representations. These correspondingly incorporate spinor mixing in a spacetime direction. Together with the covariant derivative above, the full term is often condensed using a "slashed" notation: $\gamma^\mu(\partial_\mu - igA_\mu) = \gamma^\mu D_\mu = \not{D}$.

8. m: The mass of the quark flavor in question. Together with the $\not{D}$ term, the full term between $\bar{\psi}$ and ψ can be written $i\not{D} - m = M$. This is the *Dirac matrix* and plays an important role in evaluation of Lattice QCD.

Altogether, we have a lot of pieces containing spacetime vector, spinor, and color (SU(3)) structure, all of which is reduced over in specific ways to give us a scalar-valued Lagrangian at the end of the day. It is beyond the scope of this thesis to discuss why each piece looks the way it does, but we refer the reader to Weinberg's sequence of textbooks on quantum field theory, including discussion of QCD in Volume II [56, Chap 15 & Sec 18.7]. From here on out, we will take this Lagrangian as given and discuss calculations in the context of this particular description of physics.

2.2.2 Difficulties in Evaluating the QCD Path Integral

Section A.4 discusses one method of evaluating the path integral using a perturbative expansion of interaction terms of the action (those terms involving the product of more than two fields). Importantly, this method relies on the coefficient of these terms being much smaller than 1 to allow truncating the series after only a few terms.

Returning to QCD, and ignoring quarks for a moment, we can isolate the gluon piece of the QCD Lagrangian to demonstrate why QCD fundamentally presents difficulties with perturbative evaluations of the path integral. Our gluonic Lagrangian is just:

$$\begin{aligned}
\mathcal{L}_g &= -\frac{1}{4}F_{\mu\nu}F^{\mu\nu} \\
&= -\frac{1}{4}(\partial_\mu A_\nu - \partial_\nu A_\mu - ig[A_\mu, A_\nu])(\partial^\mu A^\nu - \partial^\nu A^\mu - ig[A^\mu, A^\nu]) \\
&= -\frac{1}{2}(\partial_\mu A_\nu \partial^\mu A^\nu) + \frac{1}{2}(\partial_\nu A_\mu \partial^\mu A^\nu) + \frac{ig}{4}(\partial_\mu A_\nu - \partial_\nu A_\mu)(A^\mu A^\nu - A^\nu A^\mu) \\
&\quad + \frac{ig}{4}(A^\mu A^\nu - A^\nu A^\mu)(\partial_\mu A_\nu - \partial_\nu A_\mu) + \frac{g^2}{4}(A_\mu A_\nu - A_\nu A_\mu)(A^\mu A^\nu - A^\nu A^\mu)
\end{aligned}$$

We find that there are indeed $3A$ and $4A$ interaction terms in the Lagrangian. If the coupling constant $g \ll 1$, then we can proceed with a perturbative calculation in gluon-only QCD. It turns out, however, that in order to avoid infinities in the theory, g must be a function of energy scale (see [56, Sec 18.7] for a detailed dicussion of renormalization of QCD). In the case of QCD, we find that $g \ll 1$ only for high energies, while at low energies g becomes large. This reasoning carries over into the full description of QCD. Thus for high-energy scenarios, such as quark plasma, we can perform perturbative QCD calculations and find good results [51], but for bound states at rest, our perturbation theory breaks down and we must find another way.

2.3 Quantum Chromodynamics on a Lattice

In 1974, Wilson proposed a solution to this problem by introducing an alternative to the above Lagrangian for QCD. Instead of defining fields as continuous functions of spacetime, he defined them on a discrete lattice. He showed that this alternative Lagrangian over lattice fields correctly gave the continuum model in the limit of the lattice spacing approaching zero [58]. Discretization introduces a new method of computing values, and the finite lattice size puts bounds on the set of computations we have to do to find an answer. A calculation on a lattice is necessarily an approximation of the continuum calculation, but by taking the small lattice spacing limit one can reliably extrapolate calculations to physical values [29, Sec 2.3][3, 28].

In Wilson's formulation, spacetime is discretized as a finite lattice of N^4 lattice points, or *sites* (we choose an $N\times N\times N\times N$ hypercube for simplicity, but in practice the lattice can be, and often is, rectangular). Nearest neighbor lattice points are connected by *links*, including links connecting the boundaries in a toroidal fashion.

In this structure, the quark field takes values on a discrete set of sites, x_i, rather than all of spacetime: $\psi(x) \rightarrow \psi(x_i)$. The gauge fields require slightly carefully handling, as each $\mathfrak{su}(3)$ value is "infinitesimal" (lives in a Lie Algebra) and has a vector form (the μ index). On the lattice, Wilson chose to place the gauge field on the links. Because links have finite extent, the values should belong to the $SU(3)$ Lie *Group*: $A_\mu(x) \rightarrow U_\mu(x_i) = e^{iA_\mu}$. The result is a derived field, $U[A]$, which lives in the fundamental representation and thus takes on 3×3 matrix values for every $\mu \in (t, x, y, z)$. We take the convention that $U_\mu(x_i)$ lives on the link between sites x_i and $x_i + \hat{\mu}$, where $\hat{\mu}$ is a hop of one lattice spacing in the μ direction. The Hermitian conjugate, $U^\dagger_\mu(x_i)$, lives on the link in the reverse direction, from $x_i + \hat{\mu}$ to x_i. In general, we can compute $U^\dagger_\mu(x_i)$ from $U_\mu(x_i)$ as needed.

Figure 2-1 diagrams a 2D slice of the lattice, pictorially representing the forms of the quark and gluon fields on the lattice.

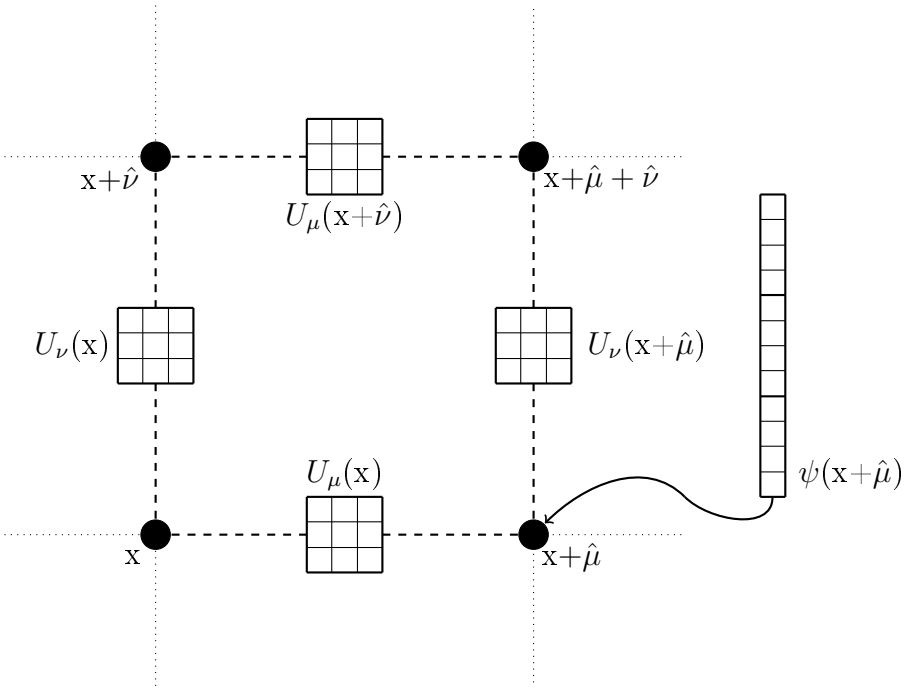

Figure 2-1: 2D slice of the lattice demonstrating gauge and quark field representations.

2.3.1 Lattice QCD Action

The form of the action, previously an integral of a Lagrangian density at every space-time point, becomes a discrete sum over the lattice:

$$S_g = \sum_x \left(-C_1 \left[\sum_{\mu,\nu} \mathrm{Tr}(U_\mu(x) U_\nu(x+\hat{\mu}) U_\mu^\dagger(x+\hat{\nu}) U_\nu^\dagger(x)) \right] \right)$$

$$S_f = \sum_x \left(- \left[C_2 m \bar{\psi}(x)\psi(x) + C_3 \sum_\mu \bar{\psi}(x)(1-\gamma_\mu) U_\mu(x) \psi(x+\hat{\mu}) \right. \right.$$
$$\left. \left. - \bar{\psi}(x-\hat{\mu})(1+\gamma_\mu) U_\mu^\dagger(x-\hat{\mu}) \psi(x) \right] \right)$$

$$S_{\mathrm{latt}}[U, \psi, \bar{\psi}] = S_g[U] + S_f[U, \psi, \bar{\psi}]$$

In this version of the action, known as the Wilson action, derivatives are replaced with discrete differences, and the $\mathrm{Tr}(F_{\mu\nu}F^{\mu\nu})$ term has been reformulated as a sum of traces of Us circulating around 1×1 boxes, known as a "plaquettes". Wilson showed that in the limit of zero lattice spacing these terms reduce to exactly the continuum

action described previously [58].

It is worth noting that there are other forms of lattice Lagrangians for QCD, which equivalently reduce to the continuum action in the limit of zero lattice spacing. These forms generally involve terms with traces around other forms of link loops and terms between further separated fermion sites [47]. In practice, it is often useful to extend the Wilson action with these higher-order terms, to achieve smaller statistical errors or faster convergence [30, Sec 3]. For the sake of simplicity, we will not delve into specific forms for these corrections, but will keep in mind that added terms generally either follow the form of a generalized plaquette, i.e. taking the trace of a product of Us around a closed loop, or of the discrete derivative, i.e. accessing nearby values of ψ, transporting the values by multiplication through a series of links to the central site, and finally multiplying with $\bar{\psi}$ at the central site.

2.3.2 Evaluating the Path Integral on a Lattice

With this new form of the action in hand, we can return to the path integral and find that this allows us to make progress on physical calculations even in a large-coupling situation. Our path integral in continuum QCD is formulated in terms of the *functional* integral of fields $\int \mathcal{D}\psi(x)\mathcal{D}\bar{\psi}(x)\mathcal{D}A_\mu(x)$, which is an uncountably infinite number of integrals, one per spacetime point. On an N^4 lattice, this is replaced with N^4 integrals per field component: $\int \mathcal{D}\psi(x_i)\mathcal{D}\bar{\psi}(x_i)\mathcal{D}U_\mu(x_i)$. Combining this with our lattice action, we have the following form for the lattice path integral:

$$\langle 0|T(\mathcal{O})|0\rangle = \frac{1}{Z}\int \mathcal{D}\psi(x_i)\mathcal{D}\bar{\psi}(x_i)\mathcal{D}U_\mu(x_i)\mathcal{O}e^{iS_{\text{latt}}}$$

Since we have reduced ourselves to a finite number of integrals for a given lattice spacing, we could in principle numerically integrate each component in sequence via evenly distributed sampling and arrive at an answer for a given path integral. However, the high-dimensional and sharply peaked nature of the integral due to contributions around classical solutions suggests the use of Monte Carlo techniques for evaluation [37]. But, in order for Monte Carlo techniques to apply, our problem must

be reformulated to look like an integration over a probability distribution. As it stands, we have two issues: first, our integral includes *complex* phases, and second, we are integrating over anti-commuting Grassmann numbers.

To solve our first issue, we make use of a Wick rotation, defined as a rotation of the integration contour from the real to the imaginary axis of the time component of the action [57]: $t \to i\tau$. As a result, the integration measure of our continuum action transforms as $d^4x \to id^4x_E$, and our action exponential becomes entirely real: $\exp\left(i\int_x \mathcal{L}\right) \to \exp\left(-\int_{x_E} \mathcal{L}_E\right)$. We write this transformed coordinate as x_E because the inner product of vectors in the transformed coordinate space, (τ, x, y, z), matches the 4-D Euclidean inner product. As a result, this is often termed the Euclidean form of the path integral.

This Wick rotation is equally valid for our discretized lattice action. Rewriting our lattice path integral, we can interpret the integral as a probability distribution of our operator $\mathcal{O}$:

$$\langle 0|T(\mathcal{O})|0\rangle = \frac{1}{Z}\int \underbrace{\mathcal{D}\psi(x_i)\mathcal{D}\bar{\psi}(x_i)\mathcal{D}U_\mu(x_i)e^{-S_{\text{latt},E}}}_{\text{probability distribution}} \underbrace{\mathcal{O}}_{\text{integrand}}$$

To address the issue of anti-commuting numbers, we can replace our ψ quark field with a "pseudo-fermionic" *commuting* field, χ. To do so, we use the properties of Gaussian integrals of commuting and anti-commuting numbers:

$$\int \mathcal{D}\psi\mathcal{D}\bar{\psi}e^{\sum_x\sum_y \bar{\psi}(x)M(x,y)\psi(y)} \propto \det M$$

$$\int \mathcal{D}\chi\mathcal{D}\bar{\chi}e^{\sum_x\sum_y \bar{\chi}(x)A(x,y)\chi(y)} \propto \frac{1}{\det A}$$

Where ψ and χ are anti-commuting- and commuting-valued fields respectively. These identities, together with $(\det A)^{-1} = \det A^{-1}$, allow us to rewrite [38, Sec 18]:

$$\int \mathcal{D}\psi\mathcal{D}\bar{\psi}e^{\sum_x\sum_y \bar{\psi}(x)M(x,y)\psi(x)} \propto \int \mathcal{D}\chi\mathcal{D}\bar{\chi}e^{\sum_x\sum_y \bar{\chi}(x)M^{-1}(x,y)\chi(y)}$$

There is an important constraint here: in order for the Gaussian integral to con-

verge, we must have positivity of the inverted matrix [14]. As a result, initial numerical work on Lattice QCD was often restricted to the unphysical case of two mass-degenerate quark flavors. This is described by a Lagrangian with two quark fields of the same mass, and thus two copies of the matrix in the path integral:

$$\mathcal{L}_{2q} = \mathcal{L}_g + \bar{\psi_1}(i\not{D} - m)\psi_1 + \bar{\psi_2}(i\not{D} - m)\psi_2$$

$$\begin{aligned}
\langle 0|T(\mathcal{O})|0\rangle &= \frac{1}{Z}\int \mathcal{D}U\mathcal{O}(\det M)^2 e^{-\sum \mathcal{L}_g} \\
&= \frac{1}{Z}\int \mathcal{D}U\mathcal{O}(\det M^\dagger)(\det M)e^{-\sum \mathcal{L}_g} \\
&= \frac{1}{Z'}\int \mathcal{D}\chi\mathcal{D}U\mathcal{O}e^{-\sum(\mathcal{L}_g + \bar{\chi}(M^\dagger M)^{-1}\chi)}
\end{aligned}$$

In the second step, we made use of the important property that the Dirac matrix determinant is real [30]. This transformation guarantees positivity, and clears the final obstacle to performing numerical calculations of physical values in Lattice QCD. There are methods for extending the numerical technique to odd or non-degenerate flavors of quarks, but the core of the problem lies in being able to solve this basic case [38, Sec 18.2.1], and as such we will assume we are always working with the positive definite $M^\dagger M$. With a handle on the physics of Lattice QCD, we move on to describing the computational aspects of Monte Carlo evaluation of our final form of the path integral:

$$\boxed{\langle 0|T(\mathcal{O})|0\rangle = \frac{1}{Z'}\int \mathcal{D}\chi\mathcal{D}U\mathcal{O}e^{-\sum(\mathcal{L}_g + \bar{\chi}(M^\dagger M)^{-1}\chi)}}$$

2.4 Lattice QCD as a Computational Task

Under Monte Carlo evaluation, computing the expectation value of a particular QCD observable, $\mathcal{O}$, can be broken down into the following steps:

1. Randomly generate a finite ensemble of gauge configurations, U_i, with probability $(\det M)^2 e^{-S[U_i]} = e^{-S[U_i] - \sum \bar{\chi}(M^\dagger M)^{-1}\chi}$.

2. Evaluate $\mathcal{O}[U_i]$ on all states, averaging to approximate the expectation value $\langle\mathcal{O}\rangle$.

While on the surface this consists of several complex linear algebraic operations on lattice vectors and matrices, we show in detail that these can all be broken down into vector and matrix sums, scalings, and products. In addition, the locality of matrices generated from Lattice QCD means their action on vectors can be computed using *stencils*: a computation kernel over a grid which, for each site, accesses neighboring cells in the same way [59, p. 221]. We diagram specific forms of stencils used in various pieces of Lattice QCD computations.

Inverting the Dirac matrix plays an important role in both generating ensembles of gauge configurations and evaluating operators. We begin with computational strategies for inverting the Dirac matrix (Section 2.4.1), then discuss how one can use Dirac matrix inversion to evaluate the action (Section 2.4.2) and generate gauge ensembles (Section 2.4.3). Finally, we discuss how one particularly important operator can be described in terms of Dirac matrix inversion (Section 2.4.4).

2.4.1 Inverting the Dirac matrix

Several common observables, as well as gauge-field generation, require solving the Dirac equation for a known source, η:

$$M\psi = \eta \;\rightarrow\; \psi = M^{-1}\eta$$

Mathematically, the Dirac matrix is large: the value connecting two sites is one gamma matrix (4×4) for each element of a gauge matrix (3×3). If stored in a dense format, the whole matrix would have $4*4*3*3*(N^4)^2$ elements. For even moderately sized lattices this quickly expands beyond what we can store in memory. However, the Dirac interaction is highly local, meaning M is a banded matrix: only neighboring pairs of lattice sites have non-zero values. Of $O((N^4)^2)$ possible elements, only $O(N^4)$ will be non-zeros. This is exactly the form of matrix that is well-represented by sparse formats. We may choose to represent the matrix fully-assembled or factored into the

gamma and gauge components, but in either case we asymptotically save space by storing values per link (a sparse structure) rather than per pair of sites (a dense structure).

The inverted matrix, M^{-1}, has no similar locality in general. For large lattices, storage of a dense matrix is extremely expensive, prohibiting generation of M^{-1} via a direct solve. Even if we could store such a solution, in these types of sparse systems iterative solvers perform better than direct solvers and avoid accumulating round-off errors [9].

Iterative solvers generally convert the problem of inverting a matrix M to an iterative convergence of a solution estimate vector $\psi_0 \to \psi_1 \to \ldots \to \psi_n$. One common example of such a solver is the Conjugate Gradient method, in which each step consists of matrix-vector multiplications, vector algebra, norm, and scalar multiply operations [48].

Iterative solvers may require many iterations to converge, which can be mitigated by preconditioning the matrix. Preconditioners generally seek to improve the condition number $\kappa(M)$ of the matrix by transforming the solution equation [42, Chap 10]. The condition number is directly correlated with the number of iterations for convergence in an iterative solver, so finding ways to reduce it can result in significant gains [48, Sec 10].

One demonstrative example is the even-odd preconditioner, which is commonly used in Lattice QCD. It takes advantage of the direct locality of the Dirac matrix to split the lattice in a chessboard fashion into two subsets (even and odd). Of the four submatrices, the even-even and odd-odd matrices are proportional to the identity, because the Wilson action contains only nearest-neighbor terms. These submatrices are thus trivially invertible, allowing a factorization resulting in an improved condition number [22]. This method of preconditioning typically results in reducing the condition number to less than half the original value, resulting in significantly fewer iterations to convergence [31].

Naive Dirac matrix inversion via iterative methods only requires vector-vector additions and inner products, and matrix-vector multiplications. Vector-vector oper-

ations can be represented as stencils of a single lattice site: each site of one vector is multiplied or added with the value of the other vector at exactly that site. Multiplication by the Dirac matrix can be represented by a generalized von-Neumann stencil, shown in Figure 2-2 for the case of a 2D lattice. Each row of the Dirac matrix corresponds to one row, i.e one site, of the output vector. The calculation of this row involves accessing vector values one hop away in all lattice directions.

Dirac matrix inversion using even-odd preconditioning requires vector-vector operations plus matrix-vector multiplications of the four Dirac submatrices. The even-even and odd-odd submatrices can be represented as stencils of a single lattice site: with only one-hop terms in the Wilson action, values between pairs of distinct even or distinct odd sites are all zero. The even-odd and odd-even submatrices can be represented as a von-Neumann stencil, minus the central site, applied to the even or odd sublattices.

2.4.2 Action Computation

The Wilson action described in Section 2.3 has two components: the gauge kinetic term and the pseudofermion term. The pseudofermion component in the action can be written in terms of Dirac matrix inversions, $S_{pf} = \bar{\chi}(M^\dagger M)^{-1}\chi$. We therefore focus on computation of the gauge piece.

As a reminder, the gauge piece of the lattice action takes the form:

$$S_g = \sum_x \left(-C_1 \left[\sum_{\mu,\nu} \mathrm{Tr}(U_\mu(x) U_\nu(x+\hat{\mu}) U_\mu^\dagger(x+\hat{\nu}) U_\nu^\dagger(x)) \right] \right)$$

The gauge kinetic term multiplies loops of links in each of the six planes of 4D spacetime (t-x, t-y, t-z, x-y, x-z, y-z). The computation in each plane can be represented using the *plaquette* stencil pattern, as depicted in Figure 2-3.

As mentioned previously, improvements to the action follow the form of loops of links and beyond-nearest-neighbor hops between fermion sites. A stencil depiction of one improved form of the action, the Clover action, is given in Figure 2-4. The Clover stencil is very similar to the plaquette stencil, thus we focus on only the basic

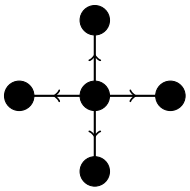

Figure 2-2: The 2D von Neumann stencil accesses immediate Cartesian neighbor links and sites.

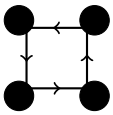

Figure 2-3: The 2D plaquette stencil accesses links in loops around every 1×1 box.

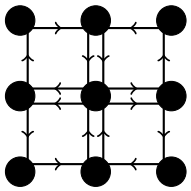

Figure 2-4: The 2D clover stencil accesses links in loops in all directions from the central site.

Dirac and plaquette stencils of the Wilson action as representative forms of the lattice operations involved in Lattice QCD.

Together, the von-Neumman and plaquette stencils allow us to evaluate the gauge and pseudofermion pieces of the basic Wilson action. Improvements to the action can be written in terms of more complex stencils, such as the clover stencil. We conclude that a stencil representation of linear algebra on the lattice allows a complete description of the action computation.

2.4.3 Gauge Field Ensembles

Generating gauge field ensembles stochastically is an example of a Monte Carlo algorithm applied to integro-differential equations, as first suggested by Metropolis et al. in 1949 [37]. In the case of gauge fields, it involves randomly sampling individual configurations to numerically approximate a solution to the analytically hard functional integral.

For the given Lattice QCD integral, U should be sampled with probability $p(U) = (\det M)^2 \exp(-S[U])$ to match the path integral probability distribution associated with computing an observable:

$$\sum_{U_i} \mathcal{O}[U_i] \approx \frac{1}{Z} \int \mathcal{D}U (\det M)^2 \exp(-S[U]) * \mathcal{O}[U]$$

The probability associated with each configuration is non-local, making it difficult to factor the problem into simple sampling of values at each site of the lattice. Instead

of attempting to directly sample U, we can construct an algorithm which stochastically builds a chain of states, each one based solely on the previous, which approaches a desired equilibrium distribution given a long enough chain. This method, known as Markov Chain Monte Carlo, was first proposed by Metropolis et al. in 1953, and later extended by Hastings in 1970 [33, 21], and has been applied with great success to gauge field generation in Lattice QCD. For brevity, we assume many of the properties of Markov Chains hold, given sufficiently well-behaved transition probabilities. For excellent discussions and proofs of the uniqueness of the normalized equilibrium state, eventual convergence to equilibrium, and the statistics of Markov Chains in the context of Lattice QCD, see [31, Sec 2.2].

We begin by discussing the Metropolis-Hastings method, a method to apply an accept or reject step to a sufficiently nice stochastic transition step to achieve a desired equilibrium state. Following this, we discuss one commonly used stochastic transition step, Hybrid Monte Carlo.

Metropolis-Hastings Method

Suppose we are given a transition probability between states, $T(U_i \to U_{i+1}) \in [0,1]$, and we have a desired equilibrium distribution $P(U_i) \in [0,1]$. To achieve this distribution as an equilibrium state of a Markov Chain, we can apply an acceptance probability on top of this stochastic transition, keeping the randomly selected U_{i+1} with probability $p(U_i \to U_{i+1})$ and otherwise reverting to U_i. The Metropolis-Hastings method prescribes the following form for p:

$$p(U_i \to U_{i+1}) = \min\left(1, \frac{P(U_{i+1})}{P(U_i)}\right)$$

This form of the acceptance probability satisfies the condition of *detailed balance*, one of a number of possible conditions that ensures our Markov Chain reaches the desired equilibrium probability [32, Sec 4.4]:

$$p(U_i \to U_{i+1}) * P(U_i) = p(U_{i+1} \to U_i) * P(U_{i+1})$$

For Lattice QCD specifically, we require $P(U_i) = (\det M[U_i])^2 * e^{-S[U_i]}$. With a given sufficiently nice transition function, we can thus use an acceptance probability as below to generate a Markov Chain which will settle to the desired distribution for a long enough chain:

$$p = \min\left(1, \frac{(\det M[U_{i+1}])^2}{(\det M[U_i])^2} * \frac{e^{-S[U_{i+1}]}}{e^{-S[U_i]}}\right)$$

Computationally, this means for each new state we generate, we must evaluate the pseudo-fermion action to incorporate the determinant of the Dirac matrix. As discussed in Section 2.4.2, this is dominated by iterative inversions of the Dirax matrix.

As expected from the stochastic nature of the process, the size of the ensemble affects the variance of the measured observable. From experimental use of Markov Chain Monte Carlo processes, it seems that chains on the order of 100 to 1000 states are sufficient to achieve relatively good results [31, Sec 2.1.2]. Though still computationally intensive, this clearly demonstrates that the Monte Carlo approach successfully takes an analytically impossible problem into the domain of tractable computation.

Hybrid Monte Carlo

The Metropolis-Hastings method described above does not select a particular underlying stochastic transition function. From the form of the acceptance probability we can see that a transition function that often generates U_{i+1} with low acceptance probability, $P(U_{i+1}) \ll P(U_i)$, will result in many stagnant iterations. This is undesirable, since it leads to slow equilibration of the Markov Chain. Ideally, we would like a transition function that generates U_{i+1} with $P(U_{i+1}) \approx P(U_i)$.

Duane et al. proposed in 1987 a unification of the existing Metropolis method with molecular dynamics techniques independently being used to construct appropriate configuration distributions [14]. This method, known as "Hybrid Monte Carlo" (HMC), or "Hamiltonian Monte Carlo", provides a mechanism to advance our U_i such

that we are likely to accept the final state U_{i+1}.

The key idea in HMC is to define a kinetic model which describes how to advance U_i based on a Hamiltonian description in some fictional "simulation" time (these are *not* the physical dynamics). This allows us to advance the state U_i forwards without sudden large changes in action. The HMC Hamiltonian is defined with the action as the potential energy, and a kinetic energy in terms of a new conjugate momentum field π. The field π is defined to be conjugate to the collective $U_\mu(x)$ and χ fields. Treating the dot product below as ranging over all of these indices, we can define the Hamiltonian as the following scalar:

$$H(U, \pi, \chi) = \frac{1}{2}(\pi \cdot \pi) + S_g[U] + S_{pf}[U, \chi]$$

Where we define:

$$S_{pf}[U, \chi] \equiv \bar{\chi}(M^\dagger M)^{-1}\chi$$

Using this Hamiltonian, we can advance U_i using the Hamiltonian equations of motion:

$$\begin{aligned} \partial_t\{U, \chi\} &= \pi \\ \partial_t\pi &= -\frac{\delta S_g}{\delta U} - \frac{\delta S_{pf}}{\delta U} - \frac{\delta S_{pf}}{\delta \chi} \end{aligned}$$

We can advance the fields discretely using a "leapfrog" integration scheme which tends to work well in practice [36, Sec 2.3]. At the end of a sequence of n steps of a total advancement time, τ, we have a new state, U_{i+1}, and a new momentum, π.

By defining an acceptance probability based on $H(U, \pi, \chi)$, $p(U_i \to U_{i+1}) = \min(1, e^{H_i}/e^{H_{i+1}})$, we build an ensemble with the desired statistics [36]. If we could perfectly integrate the equations of motion this probability would always be 1, but in using a discrete process we introduce integration errors. The choice of τ and n can be tuned to achieve a desired acceptance rate.

From this procedure we can identify the computational elements required to perform a single HMC update:

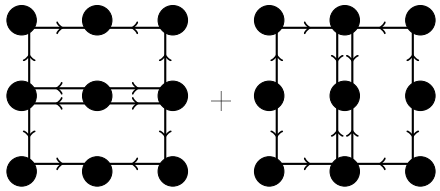

Figure 2-5: Staple stencil described in [31].

1. Random generation of the fields π, U, and χ on the lattice.

2. Linear algebra on π, U, and χ.

3. Derivative $\frac{\delta S_g}{\delta U}$: a lattice vector proportional to the staple sum stencil as shown in Figure 2-5 [31]. We omit the detailed derivation for brevity.

4. Derivatives $\frac{\delta S_{pf}}{\delta U} + \frac{\delta S_{pf}}{\delta \chi}$: a lattice vector derived from Dirac matrix inversions and a von Neumann stencil. This is generally the largest fraction of the computation, due to inclusion of Dirac matrix inversions [31, 18]. We again omit the detailed derivation for brevity.

2.4.4 Correlation Functions

The choice of physical operator depends on the experiment in question. One particularly useful operator is the *two-point correlator* for a given source and sink: $C(x,0) = \langle 0|T(\mathcal{O}_{\text{sink}}(x)\mathcal{O}^{\dagger}_{\text{source}}(0))|0\rangle$. Typically, the source is a collection of quark fields, $\bar{\psi}_0...\bar{\psi}_n\psi_0...\psi_m$, and the sink their conjugates, $\psi_0...\psi_n\bar{\psi}_0...\bar{\psi}_m$. By summing over all lattice points in a spatial slice and extrapolating to large time separation, one can extract the energy levels of physical objects with quantum numbers matching that of the operators [18, Sec III].

Correlation functions described by these forms of source and sink can be written inside the path integral entirely in terms of Dirac matrix inversions. Specifically, multiplying a source with its conjugate sink is equivalent to the sum over inverse Dirac matrix terms between source and sink locations for each possible pairing of identical quark and quark conjugate forms in the source and sink terms [38, Sec

18.2]. As a result of this equivalence, typical operators can be computed in terms of iterative Wilson inversions, as we have already discussed.

2.4.5 Pseudocode Description

We summarize the typical elements of a Lattice QCD program in the pseudocode listing below, demonstrating Monte Carlo evaluation of the path integral for a given correlation function, $C(x, 0)$:

---

input: τ, n_l, N

procedure GenerateGaugeEnsemble(n)

  $U_0 \leftarrow$ RandomU();

  $\chi_0 \leftarrow$ RandomPseudoFermion();

  $\pi_0 \leftarrow$ RandomConjugateMomentum();

  $H_0 \leftarrow \frac{1}{2}\pi_0^2 +$ ComputeAction(U_0, χ_0);

  for i ***in*** $[1, n-1]$

    $U_i, \chi_i, \pi_i \leftarrow$ LeapFrogIntegrate(U_{i-1}, χ_{i-1}, π_{i-1}, τ, n_l);

    $H_i \leftarrow \frac{1}{2}\pi_i^2 +$ ComputeAction(U_i, χ_i);

    $U_i \leftarrow$ WithProbability($\min(1, \exp(H_i - H_{i-1}))$, U_i, U_{i-1});

  end for

  return $U_0, \ldots, U_n$

end procedure

ensemble $\leftarrow$ generateGaugeEnsemble(N);

$C \leftarrow 0$;

for U_i ***in*** ensemble

  $C \leftarrow C + (1/N) *$ ComputeCorrelator(U_i);

end for

The runtime-limiting factors in this pseudocode are the ComputeAction, LeapFrog-Integrate, and ComputeCorrelator steps, all of which require iterative solvers for the Dirac equation. Typically, there are many HMC steps per gauge configuration added to the ensemble, and the cost of computing the ensemble therefore dominates.

2.5 Catalog of Lattice Linear Algebra

We conclude by summarizing all the pieces of lattice linear algebra we have identified above.

There are two types of vectors over the lattice:

1. Gauge-type spacetime vectors taking a 3×3 complex matrix value per link, or equivalently four 3×3 matrix values per site, one for each direction μ. We can write these as $V[x_i]_\mu^{jk}$, where x_i is a lattice coordinate, $\mu \in (t, x, y, z)$, and $j, k \in 0, 1, 2$ are gauge indices.

2. Quark-type vectors, taking a 3-vector (gauge) form of complex 4-vector (spinor) blocks, or equivalently a 4-vector (spinor) form of complex 3-vector (gauge) blocks, per site. We can write these as $V[x_i]_\alpha^j$, where x_i is a lattice coordinate, $j \in 0, 1, 2$ is a gauge index, and $\alpha \in 0, 1, 2, 3$ is a spinor index.

The Dirac matrix serves as the main matrix structure of the problem, and is parametrized by a given gauge configuration, U_i. Importantly, because the Dirac matrix has a regular sparse structure, we know that it has $O(\text{sites})$ elements, and that it has the same number of elements per row. It is these properties that allow us to write the matrix as a stencil over the lattice.

There are several operations we may choose to perform on the Dirac matrix and these vectors:

1. Evaluate the inner product of quark-type vectors, $\bar{\xi}\psi = \sum_x \bar{\xi}(x)\psi(x)$. Keeping in mind that $\bar{\xi} = \xi^\dagger\gamma_0$, this involves complex conjugation, a 4×4 gamma matrix multiplication, and a reduction over the entire lattice. This operation also allows us to evaluate the norm of quark-type vectors.

2. Multiplying the Dirac matrix into a quark-type vector, $\eta = M\psi$. For each row, this involves accessing the one-hop nearest neighbors of each site, which can be described as a von-Neumann stencil.

3. Iteratively solving the Dirac equation, $\eta = M^{-1}\chi$, which reduces to a sequence of Dirac matrix multiplications and inner product evaluations.

4. Preconditioning. In the common even-odd method, this involves dividing the lattice into two subsets, and breaking up the Dirac matrix accordingly. The result is two submatrices involving the nearest neighbor piece of the Dirac matrix, and two submatrices proportional to the identity.

All of these operations can be reduced to one core function: a stencil function mapped over a regular subset of sites of the lattice, and possibly reduced. Specifically, scalings, sums, and inner products of vectors can all be described by a trivial single-site stencil mapped over the entire lattice; Dirac matrix multiplications can be described by more complex stencils because the sparsity structure guarantees a fixed and locally identical operation per row; iterative matrix inversions can be written entirely in terms of matrix multiplications and vector algebra; and finally the even-odd preconditioner can be written in terms of simpler matrix multiplications on lattice subsets, and vector algebra.

Chapter 3

Simit and Halide Review

We will describe linear algebra on lattices as an extension to the Simit programming model. The Simit programming model allows a description of linear algebra over nodes and edges of graphs. We review this model and in particular highlight the dual views offered by Simit: a local graph view, and a global linear algebra view (Section 3.1). In our extension to Simit, we will describe how the local graph view can be enriched by a stencil description of lattice graphs while maintaining the powerful global linear algebra view.

In our evaluation of these extensions, we build a prototype compiler with a Halide backend (Chapter 6). We review the features of Halide and in particular highlight the stencil descriptions of stages in an image pipeline, and the scheduling language enabling optimization of these pipelines through manipulation of lattice indices (Section 3.2). These features allow us to quickly explore stencil code generation and optimization.

3.1 Simit

Simit is a language designed to allow global linear algebra operations on hypergraph structures defined as *sets of nodes* and *sets of edges* connecting other sets (either node or edge sets) [25]. By allowing the user to define global matrices via a local assembly construct mapped over either a node or edge set, the Simit compiler can

transform global linear algebra operations to local in-place operations over the graph. In comparison to existing sparse linear algebra libraries, which require the user to translate their custom graph structures to and from a common sparse matrix format, Simit avoids translation costs and can make use of the structure of the graph for efficiency.

In the following, we review the main Simit features relevant to our language design:

1. Simit syntax
2. The linear algebra type system
3. The assembly construct for matrix definitions
4. Storage of assembled sparse matrices
5. Translation of linear algebra to index expressions

This review cannot do justice to the entire Simit language, and for more detail we refer the reader to [25].

3.1.1 Simit Syntax

A Simit program consists of element definitions, declarations of externally bound sets, assembly functions, and general functions. Functions in Simit contain typical elements of a general purpose language: variable declarations, assignments, algebraic operations, conditionals, and loops.

A Simit element defines a list of fields of various primitive and higher-order types. Element definitions are delimited by the `element` and `end` keywords, and consist of a sequence of field and corresponding type declarations.

An externally bound set is declared using the `extern` keyword, and a `set` type. Both node and edge set types declare the underlying element type, and edge set types additionally declare a list of their endpoint sets. The element type of the set determines the set of fields that are stored on each node or edge of the set.

Simit functions define a sequential list of commands to be executed. Internal functions are declared with the `func` keyword, while externally callable functions are declared with `extern func`. Externally callable functions typically manipulate global vector and matrix values via linear algebraic operations, allowing concise description of global transformations of the graph. The global vectors available to a function are each field of every extern set, and vectors that are assembled via `map` operations. The global matrices available to a function are always assembled via `map` operations. We discuss the semantics of assembling global matrices and vectors in 3.1.3.

We demonstrate the syntax of a full Simit program in Listing 3.1. This example simulates one step of a spring force integration on a mesh of springs and points [24]. The notable features are:

- Definition of elements stored on nodes and edges of the hypergraph. (Lines 1-8)
- Externally bound graph data: sets of nodes (Line 10) and edges connecting nodes (Line 11).
- Assembly constructs defining global matrices based on edge and node data. (Lines 13-22)
- An `extern` function which maps the assembly function, `f`, over the edge set to build a matrix, and subsequently performs linear algebra using this matrix and global vectors. (Lines 24-56)

Listing 3.1: Simit example program, demonstrating element definition, matrix assembly, and global linear algebra. This program executes a Conjugate Gradient iterative solver given a source vector of values on the nodes.

```
1 element Point
2   src       : float; % source values
3   solution  : float; % solution values
4 end
5
6 element Link
```

```
7     a : float; % link coefficient
8   end
9
10  extern points  : set{Point};
11  extern links   : set{Link}(points,points);
12
13  func f(l : Link, p : (Point*2)) -> (A : matrix[points,points](float))
14    A(p(0),p(0)) =  l.a;
15    A(p(0),p(1)) = -l.a;
16    A(p(1),p(0)) = -l.a;
17    A(p(1),p(1)) =  l.a;
18  end
19
20  func eye(p : Point) -> (I : matrix[points,points](float))
21    I(p,p) = 1.0;
22  end
23
24  export func main()
25    % build matrix to be solved
26    I = map eye to points;
27    A = I - 0.01 * (map f to links reduce +);
28
29    var xguess : vector[points](float) = 0.0;
30    var x : vector[points](float);
31
32  % begin Conjugate Gradient solver
33    tol = 1e-12;
34    maxiters = 100;
35    var r = points.src - (A*xguess);
36    var p = r;
```

```
37    var iter = 0;
38    x = xguess;
39
40    var rsq = dot(r, r);
41    while (rsq > tol) and (iter < maxiters)
42      Ap = A * p;
43      denom = dot(p, Ap);
44      alpha = dot(r, r) / denom;
45      x = x + alpha*p;
46      oldrsq = dot(r,r);
47      r = r - alpha * Ap;
48      rsq = dot(r,r);
49      beta = rsq/oldrsq;
50      p = r + beta*p;
51      iter = iter + 1;
52    end
53  % end Conjugate Gradient solver
54
55    points.solution = x;
56  end
```

In this example, the user would compile and instantiate the Simit `main` function from a C++ framework using the Simit runtime library. At the moment, Simit programs are compiled in memory, and as a result both compilation and evaluation would be performed within the same frame code. Simit allows multiple executions of a compiled function and in-place modification of the graph data. A typical use for this form of program would be to execute the Simit `main` function on the same data multiple times, perturbing either the source (via `points.src`) or matrix (via `links.a`) as external inputs to the system.

3.1.2 Linear Algebra Types

In its current iteration, Simit supports blocked *vectors* and *matrices*. Higher-order tensors are allowed by the general syntax, but have not yet been incorporated due to engineering constraints. A general Simit object is described by a blocked hierarchy of vector or matrix dimensions, with a primitive underlying type. Each vector or matrix dimension can be set-sized or constant-sized. As an example, one could write `matrix[points,points](matrix[3,3](float))`. This type describes a `points` by `points` matrix, with 3×3 blocks of floats as elements. This hierarchy can be arbitrarily nested.

In addition, it is possible to nest matrix-type blocks within vectors, and vice-versa: `vector[points](matrix[3,3](float))` and `matrix[points,points](vector[3](float))` are both valid types. However, these types are restricted in their use in linear algebra operations, as discussed below, and are often not useful to construct.

Combining types via linear algebra operations requires matching blocked dimensions order-by-order. In the case of an element-wise operation, such as matrix and vector addition or matrix and vector element-wise products, the types at all levels must be identical. In the case of a matrix-vector multiplication, the rightmost dimension of the matrix must match the corresponding vector dimension at all blocking levels, and the underlying types must also match. The following list demonstrates a few potential matrix-vector multiplications and describes their validity:

- `matrix[points,points](matrix[3,3](float))` × `vector[points](vector[3](float))`: Valid. Dimensions match at all blocking levels, and both underlying types are floats.

- `matrix[points,points](matrix[3,3](float))` × `vector[points](float)`: Invalid. The matrix has an additional blocking level which is not matched in the vector. One could choose to interpret the inner blocking as a scalar float multiplied into each 3×3 blocked matrix, but this introduces ambiguity and as such is forbidden. Instead, in this situation, the vector should be promoted to the appropriate type prior to multiplication.

- `matrix[points,points](matrix[3,3](int))`
  $\times$ `vector[points](tensor[3](float))`: Invalid. The underlying types do not match. One could choose to interpret this as an implicit promotion from `int` to `float` prior to multiplication, but Simit requires an explicit promotion.

3.1.3 Assembly Construct

The Simit assembly construct provides the user a method to relate graph information to global linear algebra constructs. An *assembly function* accepts local graph information, either a single node, or a single edge and its endpoints, and writes values to a global vector or matrix. A *map* applies this assembly function to either a node set or an edge set and returns a global vector of the type constructed by the assembly function. A map over a node set must be provided an assembly function which accepts a single node element, while a map over an edge set must be provided an assembly function which accepts a single edge element and its endpoint elements. Examples of the first and second type of assembly map are demonstrated in Figures 3-1 and 3-2.

Importantly, the Simit assembly function is restricted to writing matrix or vector values at locations indexed by the set elements it is passed. For example, if the assembly function of Figure 3-2 is passed an edge connecting nodes `a` and `b`, it may only output to locations `A(a,a)`, `A(a,b)`, `A(b,a)`, and `A(b,b)`. This restricts the sparsity structure of the matrix to match that of the set it is passed. In the case of a map over an edge set, this sparsity structure allows non-zeros only between pairs of nodes connected by an edge, while in the case of a map over a node set, the matrix may only contain diagonal elements.

3.1.4 Sparse Matrix Structures

A given row in an assembled nodes-by-nodes matrix represents elements between a single node and all of its *neighbors* through the defining set of the matrix. In the case of an edge set, this is exactly the nodes which share an edge with the given node. In the case of a matrix assembled over a node set, nodes have no neighbors, and the

```
1  % element Node defined elsewhere
2  extern nodes : set{Node};
3
4  func nodeMap(node : Node)
5      -> (A : matrix[nodes,nodes](float))
6    % ...
7  end
8
9  % ...
10 A = map nodeMap to nodes reduce +;
11 % ...
```

Figure 3-1: Node map syntax. The assembly function accepts one node.

```
1  % element Node, element Edge defined elsewhere
2  extern nodes : set{Node};
3  extern edges : set{Edge}(nodes,nodes);
4
5  func edgeMap(edge : Edge, ns : (Node*2))
6      -> (A : matrix[nodes,nodes](float))
7    % ...
8  end
9
10 % ...
11 A = map edgeMap to edges reduce +;
12 % ...
```

Figure 3-2: Edge map syntax. The assembly function accepts an edge and all endpoints.

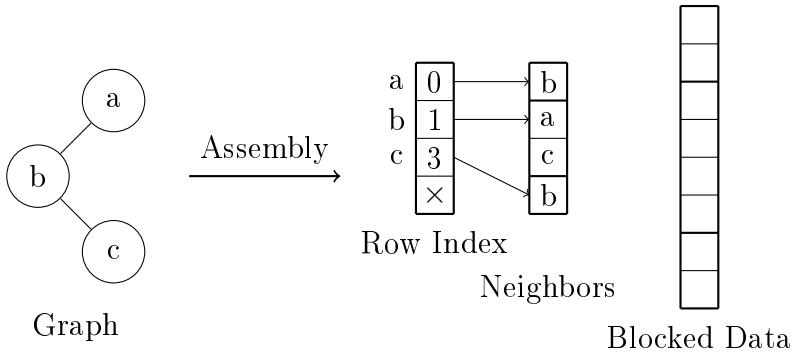

Figure 3-3: A small graph of 3 nodes and 2 edges is displayed on the left. We assume a general assembly function mapped over the edges of the graph producing a block matrix that is of type (points × points)×(1×2)(float). The resulting row index, neighbors list, and block data array are displayed on the right.

matrix is necessarily diagonal.

Non-diagonal matrices are represented in memory in a form resembling Blocked Compressed Sparse Row (BCSR) format [13]. Simit maintains edge set structural information through a neighbors list for each endpoint node. These neighbors lists are represented in memory in a condensed single list. In addition, Simit maintains a row index, with one pointer per node, pointing to the beginning of that node's neighbors section within the overall list. Each elements of the neighbors list corresponds to a non-zero block of the matrix. These block elements are laid out block-by-block following the order of the neighbors list. The neighbors list, row index, and blocked data array correspond exactly to the column list, row index, and blocked data arrays of BCSR.

Figure 3-3 demonstrates the memory structures constructed for a points-by-points matrix assembled from an edge set of a small graph. We omit the exact values of the assembly, focusing on the blocked sparse matrix structure.

Multiple matrices may be constructed via an assembly map over the same edge set. In this case, these matrices must necessarily share the same row index and neighbors structure. Simit chooses to therefore store the row and neighbors structures associated with a given edge set. The matrix-specific data are associated with a given matrix

assembly, with each data array following the blocked format defined by the edge set.

Simit also allows matrices of common dimensionality assembled from *different* sets to be combined via linear algebra operations. As an example, one can combine a diagonal points-by-points matrix and a non-diagonal points-by-points matrix, assembled by a map over a points set and an edge set, respectively. Similarly one may combine points-by-points matrices assembled by maps over two different edge sets which both connect points. The sparsity structure of the resulting matrix does not match that of the defining set of either original matrix. There are two distinct cases that Simit handles in this case: matrix addition and matrix multiplication.

In the case of matrix addition, the overall sparsity structure is a superset of the structures of the two matrices. Specifically, for each row Simit combines the neighbors lists of both matrices. The overall row index and neighbors list are generated in the usual manner from these updated local neighbors lists.

In the case of matrix multiplication, Simit generates the combined neighbors of a matrix product by identifying the neighbor-of-neighbors of every point, where the first neighbor is through the sparse structure of the first matrix and the second neighbor is through the sparse structure of the second matrix. The overall sparsity structure is then computed based on the neighbor-of-neighbors lists. For example, if matrix A has non-zero values between points `a` and both `b` and `c`, and matrix B has non-zero values between points `b` and `d` and between points `c` and `e`, then the product BA will have non-zero values between points `a` and both `d` and `e`.

These sparsity combination operations are arbitrarily composable, allowing Simit to generate a row index and neighbors list for any possible combination of matrices.

3.1.5 Linear Algebra to Index Expressions

In the Simit compiler, all linear algebra constructs are reduced to index expressions, in analog to Einstein notation from pure math [16]. We list below some common linear algebra operators and their corresponding index notation:

1. Vector addition: $\vec{a} = \vec{b} + \vec{c} \rightarrow$ `a = (i b(i) + c(i))`

2. Vector inner product: $a = \vec{b} \cdot \vec{c} \rightarrow$ `a = (b(+r) * c(+r))`, where `+r` indicates a reduced variable, one that is accumulated over its entire domain, in this case using the addition operator.

3. Matrix addition: $A = B + C \rightarrow$ `A = (i,j B(i,j) + C(i,j))`

4. Matrix-vector multiplication: $\vec{a} = B\vec{c} \rightarrow$ `a = (i B(i,+j) * c(+j))`

5. Matrix-matrix multiplication: $A = BC \rightarrow$ `A = (i,j B(i,+k)*C(+k,j))`

6. Blocked vector addition: $\vec{a} = \vec{b} + \vec{c} \rightarrow$ `a = (i,z1,..zn b(i,z1,...zn) + c(i,z1,...zn))`, where `z1,...zn` are indices running over the size of the block in all of its n dimensions.

In this notation, each index is either a "dense" index, which runs over a constant range of values, or a "set" index, which runs over an edge or endpoint set. These correspond to the constant-sized and set-sized dimensions, respectively, of Simit's type system. In the above, the block indices `z1,...zn` are dense indices, with known ranges at compile-time. The vector indices may be either dense or set indices, in the cases of element or global linear algebra respectively.

3.2 Halide

Halide is a Domain-Specific Language targeted at image processing pipelines [40]. To date, it has seen large-scale use in many of Google's photograph and video processing codes. The core tenet of Halide's philosophy is separation of the *algorithm* from the *schedule*. Writing a Halide pipeline involves defining a series of data-parallel stencil transformations on the input image, finally producing memory "realizations" of one or more of the resulting images. After defining the stencil algorithm of the pipeline, the user applies scheduling to each intermediate stage, defining where the stage should be computed and stored, and how to structure the loops over the image domain. We describe these two phases in detail in the following subsections. For a detailed description of the Halide model, we refer the reader to [40].

3.2.1 Defining a Stencil Algorithm

Halide algorithms are constructed in the context of a C++ program. The building block of Halide algorithms are `Func` objects. A Halide `Func` is defined by an `Expr` parameterized by a set of `Var`s. As an example, a simple x-directional gradient could be defined as:

```
// Example 1
Halide::Var x,y;
Halide::Func grad_x("grad_x");
grad_x(x,y) = x;
```

Halide `Func`s may call other `Func`s as part of their definition, resulting in a tree of related function definitions. These calls are parameterized by combinations of the input parameters, and importantly allow stencil definitions by indexing relative to input parameters. As an example, we could define an x-direction blur over an x-y gradient as:

```
// Example 2
Halide::Var x,y;
Halide::Func grad_xy("grad_xy"), blur_x("blur_x");
grad_xy(x,y) = x + y;
blur_x(x,y) = (grad_xy(x-1,y) + grad_xy(x,y) + grad_xy(x+1,y))/3;
```

To make use of actual data, Halide provides the `Image` construct. An `Image` wraps a `Buffer`, a multidimensional block of data, such that it can be accessed as a `Func`. One can load `Image` object data from files, build them in memory manually, or be given one as a result of realizing a `Func` over a given domain. We show an example of realizing a `Func` to an `Image`, applying a blur, and receiving the resulting realization as another `Image`:

```
// Example 3
Halide::Var x,y;
Halide::Func grad_xy("grad_xy");
```

```
grad_xy(x,y) = x + y;
Halide::Image<int> input = grad_xy.realize(10,10);
Halide::Func blur_x("blur_x");
blur_x(x,y) = (input(x-1,y) + input(x,y) + input(x+1,y))/3;
// Allocate an image smaller in x by 2 to avoid overrunning
// the 10x10 grad_xy buffer.
Halide::Image<int> output = alloc_img({1,8},{0,9},sizeof(int));
blur_x.realize(output);
```

Halide also allows an "update" definition, in addition to the initial "pure" definition. These definitions update the function values, potentially over a different domain than the initial definition. An update definition is allowed to recursively reference the previous value of the function in the definition. For example, we could define an x-y gradient, then update the definition to replace the 0th row by the 5th row:

```
// Example 4
Halide::Var x,y;
Halide::Func grad_xy("grad_xy");
grad_xy(x,y) = x + y; // Pure definition
grad_xy(0,y) = grad_xy(5,y); // Update definition
```

Any `Func` which references another `Func` puts demands on the domain over which the referred-to `Func` is provided. We must realize `blur_x` over the restricted domain $[1,8]\times[0,9]$ in Example 3, to avoid accessing the `grad_xy` realized buffer outside its domain.

In many image processing application, as in physics applications, one may want a particular set of boundary conditions to extend the domain of an `Image` beyond the provided data. Halide provides shortcuts for defining anonymous `Func`s over the `Image` to achieve several common variations. We describe three useful shortcuts:

- `BoundaryConditions::repeat_image` extends the `Image` domain by wrapping

accesses outside the domain. I.e. if one were to access values just to the left of the left boundary of a repeated `Image`, one would receive values from the right side of the `Image`.

- `BoundaryConditions::mirror_image` extends the `Image` domain by adding a flipped copy of the original image beyond the boundary in each direction. I.e. if one were to access values just to the left of the left boundary of a mirrored `Image`, one would receive values from the left side of the `Image`.

- `BoundaryConditions::constant_exterior` extends the `Image` domain by adding a constant value padding beyond the boundary of the `Image` in all directions.

3.2.2 Defining a Schedule

The strength of Halide lies in exposing the performance trade-offs of an algorithm to the user. Halide achieves this through a *scheduling language*. Once a user has defined the algorithm, they use the scheduling language to choose how the computation will be organized. This scheduling language allows users to explore trade-offs between redundant computation, locality, and parallelism. By quickly reorganizing the computation without changing the meaning of the algorithm, the user can find schedules that have good performance characteristics on their target machine, and retarget the application to other platforms as needed.

By default, each `realize()` call triggers a Just-In Time compilation phase which produces a fully-inlined schedule: the definitions of all referred-to `Func`s are inlined into the realized `Func`, and placed within a loop nest spanning the realization domain. Halide provides several scheduling primitives that allow the user to specify modifications to this default evaluation schedule.

Defining a schedule is divided into:

- The *Call Schedule*: at what loop level to compute and store each intermediate `Func`.

- The *Domain Order*: iteration scheduling (loop splitting, fusing, and reordering), and iteration parallelization (threaded parallelism and vectorization).

Call Schedule

Compute and store levels in Halide determine where in the loop nest a particular intermediate will be computed and stored. At the two ends of the spectrum, an intermediate may be computed inline (the default, or using the `compute_inline()` method) or computed as an independent root (using the `compute_root()` method), i.e. in an entirely distinct loop nest. These come with associated implied storage levels: a `Func` scheduled inline is by default stored inline as well, and thus individual values are computed temporarily and discarded; a `Func` scheduled as a root is by default stored at the root level as well, i.e. in a global array holding the entire demanded domain for this `Func`. Listings 3.2 and 3.3 demonstrate a two-stage box blur kernel written in Halide with the intermediate `Func` scheduled inline. Listings 3.4 and 3.5 demonstrate the same example with root scheduling. These listings follow the form of the scheduling demonstration presented in [40].

As demonstrated in the loop nest pseudo-code, inline scheduling generally results in better locality of evaluation at the cost of extra redundant computation, while root scheduling eliminates redundant computation at the cost of locality. Which factor is more important is at the determination of the user.

Halide allows the user to specify storage of all intermediates at any loop level outside the compute level, since the storage must be available when computing the intermediate. This means we could, for example, compute `blur_x` inline as needed, but store it at the root level, avoiding redundant computation where we have already computed `blur_x`. In this case, we avoid redundant computation and retain some measure of locality.

Halide also provides intermediate levels of compute scheduling, through the use of the `compute_at()` method. This allows the user to choose a loop level within the loop nest of each consumer at which to be computed. For multi-parameter functions, this provides a more granular control over the redundant computation and locality

Listing 3.2: Default inline schedule.

```
Halide::Var x,y;
Halide::Func blur_x, blur_y;
// Algorithm
blur_x(x,y) = (input(x-1,y)+input(x,y)+input(x+1,y))/3;
blur_y(x,y) = (blur_x(x,y-1)+blur_x(x,y)+blur_x(x,y+1))/3;
// Schedule
blur_x.compute_inline(); // Default
output = blur_y.realize(10,10);
```

Listing 3.3: Pseudo-code for produced inline loop nest.

```
for y:
  for x:
    uint8_t blur_x_down = (input(x-1,y-1)+input(x,y-1)+input(x+1,y-1))/3;
    uint8_t blur_x_mid = (input(x-1,y)+input(x,y)+input(x+1,y))/3;
    uint8_t blur_x_up = (input(x-1,y+1)+input(x,y+1)+input(x+1,y+1))/3;
    output(x,y) = (blur_x_down + blur_x_mid + blur_x_up)/3;
```

Listing 3.4: Root schedule.

```
Halide::Var x,y;
Halide::Func blur_x, blur_y;
// Algorithm
blur_x(x,y) = (input(x-1,y)+input(x,y)+input(x+1,y))/3;
blur_y(x,y) = (blur_x(x,y-1)+blur_x(x,y)+blur_x(x,y+1))/3;
// Schedule
blur_x.compute_root(); // Root scheduling
output = blur_y.realize(10,10);
```

Listing 3.5: Pseudo-code for produced root loop nests.

```
for y:
  for x:
    blur_x(x,y) = (input(x-1,y) + input(x,y) + input(x+1,y))/3;
for y:
  for x:
    output(x,y) = (blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1))/3;
```

trade-off.

Domain Order

Domain order scheduling consists of several pieces:

- Splitting and fusing loops
- Reordering loop variables
- Unrolling, vectorizing or parallelizing loops

A loop domain may be divided into an outer loop over inner loops of constant length. If we consider a loop variable `t` iterating over $[0, N-1]$, the result of a loop split is an outer variable, `to`, iterating over $[0, N/c-1]$ and an inner variable, `ti`, iterating over $[0, c-1]$. In the case where c evenly divides N, all accesses using the index `t` are simply replaced with `to`$*c+$`ti`. Halide handles the case where c does not evenly divide N by shifting the last iteration of size c to overlap with the previous loop by however elements account for the difference. Loop splitting results in additional Halide variables which may themselves be scheduled by further Domain Order scheduling.

Loop fusing combines two adjacent iteration variable into a single variable that traverses the product of the two domains. This can be particularly useful if the user wants to parallelize multiple dimensions.

Reordering loop variables exchanges the order in which the variables of a `Func` domain are looped over. This may be advantageous in cases where the user wishes to transform from linear iteration in all dimensions to a *tiled* order. This can be achieved by splitting two dimensions and reordering such that the outer variables of each split are outermost. If the iteration order corresponds to memory order, this can improve cache utilization for kernels which access nearby elements in both dimensions [27].

Vectorization and unrolling may be performed on constant loop dimensions. Typical vectorization involves splitting the innermost loop into vector-sized chunks then vectorizing the inner loop of the split.

Finally, loops in Halide may be parallelized. Halide `Func`s are inherently data parallel, allowing parallelization to be applied to any loop. Typically it is advantageous to parallelize the outermost loop of a given computation, such that each thread has sufficient work to offset the threading overhead.

Chapter 4

Related Work

We explore a variety of libraries which offer support for linear algebra on lattices (Section 4.1). In the case of Lattice QCD, we discuss specifically the USQCD libraries which provide domain-specific methods for applications (Section 4.1.3). While these libraries offer optimized code for linear algebra on regular grids, these approaches result in application codes with a mix of memory management, scheduling, and platform retargeting amongst the core algorithm. More importantly, these libraries either do not provide a stencil view of linear algebra on lattices, or provide unrestricted global indexing forms of lattice matrix construction which do not permit an optimized implementation of the matrix.

This lack of separation of stencils, linear algebra, and technicalities motivates our work in developing Simit language extensions: we seek to provide an alternative to these library approaches that separates out the local stencil description, the global linear algebra description, and the scheduling and retargeting of the generated code.

We also discuss existing linear algebra DSLs (Section 4.2). These existing languages focus on general sparse matrix definitions in their linear algebraic constructs. This excludes an important part of the description of linear algebra on lattices: the regularity of matrix assembly and multiplication due to the fixed shape of the defining stencil.

4.1 Linear Algebra Libraries

There are numerous libraries designed for linear algebra in the context of scientific computing. We discuss three specific libraries which provide sparse matrix methods: PETSc, LAPACK, and ScaLAPACK.

4.1.1 PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) provides a host of libraries designed to allow scalable, performant scientific computation in a variety of mathematical domains [5]. In particular, PETSc includes sparse matrix modules with iterative solvers such as the Conjugate Gradient method, and a wide variety of related choices. Writing an application based on iterative inversion of a system matrix involves assembly of the system matrix, initializing and running an iterative solver object, and finally cleaning up the memory allocations.

Matrix Assembly

Matrix assembly supports compressed sparse row format (CSR) by default, among a number of other formats. Assembly of compressed sparse row matrices is performed by defining the set of values and column indices for each row.

Blocked matrix formats are also supported, but focus on a small top-level matrix with system-level sparse matrices stored in nested compressed formats. This format corresponds to the type of matrices generated in multi-physics systems.

Finally, PETSc also supports matrix-free methods by allowing the user to provide a matrix-vector multiplication function.

Scheduling

PETSc provides Message Passing Interface (MPI) [19] support for multi-processor computations, with matrices, vectors, and solvers internalizing many of the details of interprocess communication. We demonstrate a representative example of a stencil-type assembly and multi-processor scheduling drawn from [5, Sec 1.4]:

```
/*
Create parallel matrix, specifying only its global dimensions.
When using MatCreate(), the matrix format can be specified at
runtime. Also, the parallel partitioning of the matrix is
determined by PETSc at runtime.

Performance tuning note:  For problems of substantial size,
preallocation of matrix memory is crucial for attaining good
performance. See the matrix chapter of the users manual for details.
*/
ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m*n,m*n);CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);
ierr = MatMPIAIJSetPreallocation(A,5,NULL,5,NULL);CHKERRQ(ierr);
ierr = MatSeqAIJSetPreallocation(A,5,NULL);CHKERRQ(ierr);
ierr = MatSeqSBAIJSetPreallocation(A,1,5,NULL);CHKERRQ(ierr);
/*
Currently, all PETSc parallel matrix formats are partitioned by
contiguous chunks of rows across the processors.  Determine which
rows of the matrix are locally owned.
*/
ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);
/*
Set matrix elements for the 2-D, five-point stencil in parallel.
 - Each processor needs to insert only elements that it owns
locally (but any non-local elements will be sent to the
appropriate processor during matrix assembly).
 - Always specify global rows and columns of matrix entries.
*/
```

```
ierr = PetscLogStageRegister("Assembly", &stage);CHKERRQ(ierr);
ierr = PetscLogStagePush(stage);CHKERRQ(ierr);
for (Ii=Istart; Ii<Iend; Ii++) {
v = -1.0; i = Ii/n; j = Ii - i*n;
if (i>0)   {J = Ii - n;
            ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
            CHKERRQ(ierr);}
if (i<m-1) {J = Ii + n;
            ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
            CHKERRQ(ierr);}
if (j>0)   {J = Ii - 1;
            ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
            CHKERRQ(ierr);}
if (j<n-1) {J = Ii + 1;
            ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
            CHKERRQ(ierr);}
v = 4.0; ierr = MatSetValues(A,1,&Ii,1,&Ii,&v,ADD_VALUES);CHKERRQ(ierr);
}
/*
Assemble matrix, using the 2-step process:
MatAssemblyBegin(), MatAssemblyEnd()
Computations can be done while messages are in transition
by placing code between these two statements.
*/
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = PetscLogStagePop();CHKERRQ(ierr);
```

Structured Grids

PETSc also provides support for interaction between linear algebra and structured grid data in the Distributed Arrays module. Specifically, users can define data distribution in one, two, or three dimensions. In two or three dimensions, users additionally choose between a box or star stencil to determine whether or not corners are available in the ghost zones [26] of a local piece of the grid data.

Matrix assembly on structured grids proceeds similarly to CSR sparse matrices, but allows defining column indices using absolute structured grid coordinates rather than a single column index.

Discussion

The scheduling example given above highlights both the strengths and weaknesses of PETSc. PETSc wraps raw MPI communications in a sensible set of matrix semantics (row-by-row division), gives the user detailed control over scheduling using these semantics, and produces efficient matrix assembly code. That said, the assembly provided in PETSc forces the user to explicitly handle the bounds of the parallelized schedule, interleaves scheduling technicalities with the core matrix assembly, and is defined at a global level, preventing PETSc from exploiting the regular nature of the assembly.

The structured grid support provided by PETSc is convenient for users when defining schedules, and provides users a simple structure-based distribution mechanism. However, the structured grid methods of PETSc still require the users to define the matrix at a global level, preventing PETSc from exploiting the stencil structure of the matrix. In addition, users must take care to define their ghost zones manually, and must consider the bounds of their local piece of the distributed grid when assembling the matrix. Finally, structured grid methods are only supported for 1D, 2D, or 3D cases. This makes PETSc unsuitable for the 4D grids of Lattice QCD applications, our motivating example.

4.1.2 LAPACK

The Linear Algebra Package (LAPACK) is a set of Fortran methods for common linear algebra operations and high-level routines that have been optimized for a large variety of machines [4]. These routines are based on optimized local linear algebra procedures contained in the BLAS library. LAPACK does not offer support for general sparse matrices, but does support banded diagonal matrices, such as those produced by stencils. In particular, LAPACK provides direct solvers for inverting matrices describing linear systems of equations.

The ScaLAPACK project continues the development of LAPACK to support scalable computation on distributed hardware [7]. These developments are based on a parallel version of BLAS, termed PBLAS. ScaLAPACK provides parallel direct solvers analogous to the LAPACK package.

Together these libraries provide efficient and scalable means to perform direct solves for banded matrices, such as those produced by stencils. These libraries are not applicable to iterative methods, however, and as such do not have applicability to the types of sparse matrix methods that benefit from iterative solvers over direct solvers, such as the Lattice QCD application.

4.1.3 USQCD Libraries

Beginning with Department of Energy funding in 2001, leading members of the Lattice QCD community in the United States have developed a nationally-maintained set of libraries for Lattice QCD computations [10]. These libraries are targeted at scientific computing hardware consisting of distributed commodity clusters and supercomputer clusters [8]. The USQCD libraries are effective for current users, but require time-intensive hand-optimization for fast future operations and platforms. This indicates programmer time could be saved by development of a platform-flexible system with independent algorithm and scheduling.

4.2 Linear Algebra Domain-Specific Languages

Existing linear algebra DSLs focus on general sparse matrix forms. These DSLs do not take advantage of the regularity of linear algebra on lattices, and as a result require additional indexing and indirection in matrix representation and multiplication. We specifically discuss MATLAB and Simit.

4.2.1 MATLAB

MATLAB is designed to perform scientific computations involving general linear algebra. It provides support for creation of sparse banded and diagonal matrices, such as those created in stencil methods. MATLAB also provides support for sparse matrix initialization from (row,col,value) triplets. In addition to sparse matrix assembly, MATLAB supports iterative solvers, including the Conjugate Gradient method.

While MATLAB provides the basic support needed for sparse matrices and inversions, the interface does not support any stencil description of matrices. In addition, as with library methods, the solvers are provided as built-ins, with no methods to modify the underlying behavior. Methods on sparse matrices derived from graph structures result in poor performance and memory characteristics, and as a result don't support scalable applications very well [25].

4.2.2 Simit

The Simit programming model provides an efficient and expressive method for structural matrices of arbitrary graphs. We identify two issues with the existing Simit model in the case of lattice graphs:

1. Simit does not take advantage of the regular nature of the graph to eliminate unneeded indices. This results in extra memory usage and indirection.

2. Simit matrix assembly focuses on matrix forms that directly correlate to edge set structures, and does not provide support for more complex stencils. This stems from the fact that in arbitrary graphs complex stencil shapes are ambiguous: the

local structure varies from node to node. Restricting to lattice graphs enables a greater degree of expressiveness than is available in the Simit model.

Chapter 5

Language Definition

We define a language which extends Simit's syntax to support linear algebra on lattices. The additions to the language are:

- Extension to the edge set type, to support lattice edge sets (Section 5.1)
- Stencil-based matrix and vector assembly (Section 5.2)

These changes (1) provide the compiler with the information that we are operating on a regular graph, (2) provide the user with a more natural and expressive stencil description of matrices, and (3) allow the compiler to manipulate matrices based on the stencil definition. We discuss compiler changes enabled by this extra information in Chapter 6.

5.1 Lattice Edge Sets

Existing Simit edge sets are defined by elements connecting a list of endpoints drawn from one or more sets of nodes. These edge sets are bound externally, receiving both data and structure during runtime. We categorize these forms of edge sets as *Unstructured* edge sets, and define an additional type of edge set that may be defined and bound: *Lattice* edge sets.

Lattice edge sets impose a regular grid structure, known at compile time, on their endpoint sets. Specifically, they are defined to take the form of a grid of $N_1 \times ... \times N_d$

points, with edges connecting nearest neighbors in all cardinal directions. We define these edge sets to have a toroidal boundary condition, i.e. stepping off one edge of the lattice in a given direction puts you at the beginning of the lattice on the other side. We make this choice for simplicity of compilation, and expect a future iteration of this work would offer other boundary conditions.

Lattice edge sets are constrained to have exactly two endpoints drawn from the same node set, i.e. to be cardinality two, homogeneous edge sets. Declarations of Lattice edge sets additionally specify the *number of lattice dimensions*, d, of the imposed lattice structure. We define syntax for such a declaration to be:

```
extern <name> :  lattice[<d>](<endpointset>);
```

Lattice edge sets are handled differently at runtime than Unstructured edge sets. Rather than building a list of edges, individually defined by their endpoints, the user specifies d size parameters, $N_1, ...N_d$, which fully specify the desired lattice structure. In both cases, the user may then define data on a per-edge basis. In the case of Lattice edge sets, the runtime library assembles this data into a canonically-ordered list of data per field of the set. Beyond the d dimensions defining the start of each lattice edge, Lattice edge sets also have a directional dimension, $\mu \in [1, d]$. We define the canonical ordering to iterate over the dimensions N_1 through N_d innermost to outer, with μ outermost. Figure 5-1 demonstrates a 2×2 lattice with the canonical order for the edge data.

Endpoint sets with imposed lattice structure, i.e. those that have a Lattice edge set declared over them, are also required to be assembled in a canonical order: iterating N_1 innermost to N_d outermost. These sets are specified by users in the usual manner, but are ordered before being bound by the runtime library. Figure 5-1 also demonstrates the canonical order of the endpoint set with imposed lattice structure.

Canonical ordering in the runtime library ensures that the compiler need not build memory indices to refer to elements of the lattice. Instead, the compiler can generate code that infers the structure from this canonical ordering in memory of both Lattice

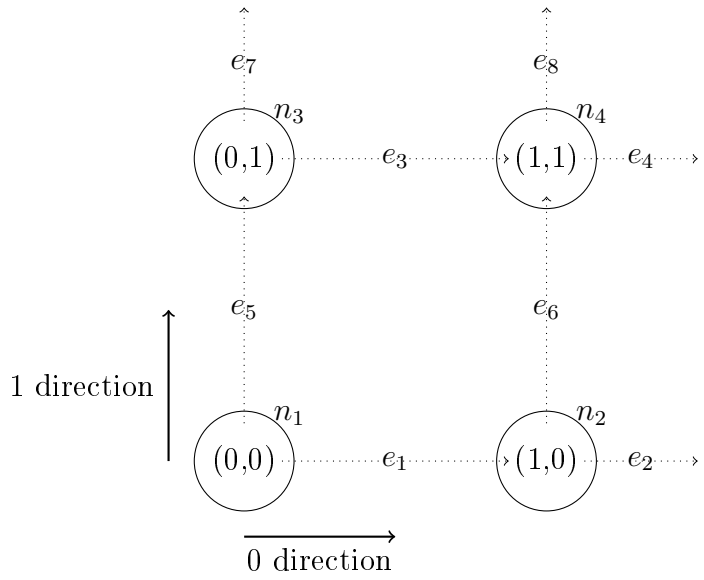

Figure 5-1: Canonical order of a Lattice edge set and the endpoint set with imposed structure on a 2×2 lattice. The Lattice edge set defines the links of the lattice, while the endpoint set defines the site. Note that there $4 * N_d = 8$ links due to the toroidal boundary condition.

edge sets and their underlying endpoint sets.

5.2 Stencil Assembly

We define additional semantics for the assembly construct that allow matrix and vector assembly from Lattice edge sets. Stencil assembly is defined by *relative indexing* in the lattice dimensions.

Stencil assembly fits within the existing map syntax, and is instead distinguished by the edge set passed to the map and a modified kernel function. Specifically, the kernel function must accept both the Lattice edge set and the underlying node set as arguments, optionally preceded by arguments to be bound by the partial arguments passed to the stencil expression. In Simit, a map over an Unstructured edge set corresponds to invoking the assembly function on each edge of the set. In maps over Lattice edge sets, we require that the kernel accept the entire edge and node sets, but constrain set accesses by using lattice indexing relative to a local lattice origin: for each node of the underlying set, we invoke the assembly function with that node

bound as the local lattice origin.

We specify the syntax of relative indexing of the endpoint set and Lattice edge set in distinct ways:

- Relative indexing of the *endpoint set* is described by d relative indices, $(i_1, ...i_d)$, which select the site offset from the local origin by the relevant index in each direction. The relative indices are constrained to be constant integers, limiting the size and form of the stencil. In this indexing, there is an implied toroidal boundary condition. The syntax of this relative indexing is: `nodes[i1,i2,...]`.

- Relative indexing of the *Lattice edge set* is described by d relative indices, $(i_1, ...i_k, ...i_d)$, followed by another d relative indices, $(i_1, ...i_k \pm 1, ...i_d)$, which together select the edge between the two indexed sites. The syntax of Lattice edge set indexing is:
  `links[i1,i2,...;j1,j2,...]`.

We demonstrate a stencil assembly defining a sparse matrix based on a von-Neumann stencil on a 2D lattice:

```
element Point
  a : float;
  x : float;
end
element Edge
  b : float;
end


extern points : set{Point};
extern edges : lattice[2]{Edge}(points);
func assemble(edges : lattice[2]{Edge}(points), points : set{Point})
    -> (A : matrix[points,points](float))
  A(points[0,0],points[0,0])  = points[0,0].a;
  A(points[0,0],points[1,0])  = edges[0,0;1,0].b  * points[1,0].a;
```

```
  A(points[0,0],points[−1,0]) = edges[0,0;−1,0].b * points[−1,0].a;
  A(points[0,0],points[0,1])  = edges[0,0;0,1].b  * points[0,1].a;
  A(points[0,0],points[0,−1]) = edges[0,0;0,−1].b * points[0,−1].a;
end

extern func main()
  A = map assemble to edges;
  points.x = A*points.x;
end
```

5.3 Discussion of Decisions

Our language design was motivated by the additional structure introduced by lattice graphs. A lattice graph provides a simple d-dimensional *global* coordinate scheme which is not present in arbitrary graphs, and allows the compiler to:

1. Remove all edge indices
2. Express matrices as stencils
3. Easily schedule iteration over the graph domain

In principle, any sort of regular graph permits a global coordinate scheme. We could imagine, for example, defining a higher-cardinality edge set which corresponds to the planes of a grid, rather than the edges. While higher-cardinality structures such as these may occasionally be useful for specific computations, the simplest possible regular structure that captures a d-dimensional regular grid is a lattice of links. For this reason we choose a cardinality-two, homogeneous link structure for Lattice edge sets.

The choice to bind Lattice edge sets and underlying endpoint sets via a memory ordering convention was guided by the desire to remove all edge indices (point 1). We note that this choice prevents the runtime system from binding Lattice edge sets of

differing sizes over the same point set. Applications using linear algebra over lattices generally perform all computations on a single lattice structure for the entire problem, and as such we chose to focus our design around this case. The removal of indices is a significant advantage provided by allowing a restricted Lattice edge set form, and we believe it is valuable to offer this trade-off to users.

The choice to define matrices via stencil constructs was guided by the desire to remove matrix indices (point 2). A matrix generated from a lattice stencil has additional structure over a general matrix, and it is this structure that allows us to define a memory-less index for this type of matrix. Specifically, in a stencil definition of a matrix, (1) the structure of the stencil is known at compile-time, and (2) the structure of the stencil is the same across the entire lattice. In our language definition, we guarantee these properties by demanding that relative lattice indexing in assembly functions be constant offsets from an implicit local origin.

To make the index-free form concrete, consider the example of the 2D von-Neumann stencil, as diagrammed in Figure 5-2, and written in code above. This stencil corresponds to *one row* of the assembled matrix, and as such we know that each row of this sparse matrix will contain exactly 5 non-zero entries. This allows us to access the jth element of the ith row of the matrix at location $i * 5 + j$ of the data array, with no indirection through an in-memory row index. This can be considered an analog to the DIA format [42, Sec 3.4], designed to store a multi-diagonal matrix. In the DIA format, one only needs to store the values of the matrix and the offsets of each diagonal. In a stencil-defined matrix, the stencil itself defines the offsets, and we need only store the values of the matrix. Beyond eliminating a set-sized memory index, this also eliminates indirection in data loads. This permits the compiler to easily vectorize and tile data accesses and computations.

Finally, the choice to assume an ordering of dimensions from inner-most first to outer-most last was motivated by a desire for engineering simplicity in implemented scheduling (point 3). This ordering matches the one used by Halide Image buffers by default, and allows a direct translation from lattice indices to Halide indices. As discussed in Chapter 8, a future iteration of this compiler could expose dimension

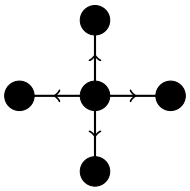

Figure 5-2: The 2D von Neumann stencil accesses immediate Cartesian neighbor links and sites.

order as a user parameter for scheduling.

Chapter 6

Prototype Compiler

We present a detailed design description of the prototype compiler built for evaluation of our methods. We hope that these design elements may eventually be folded into the Simit compiler itself, which, along with future work on lattice and unstructured linear algebra interoperability, would provide a more complete linear algebra domain-specific language.

Our objectives in designing this prototype are to:

- Demonstrate compiling a representative subset of the language defined in Chapter 5
- Demonstrate performance relative to existing methods

To efficiently meet our objectives, we build the prototype compiler as an extension to the existing Simit compiler. This allows us to take advantage of the existing parsing and lowering machinery, while only having to update the handling of maps to support stencil assembly, tailor the lowering machinery to emit index expressions, and replace the code generation with our own.

We specifically choose to use Halide as a backend for code generation. Halide provides the ability to easily schedule generated functions in terms of their defining variables, and this flexibility enables us to quickly experiment with code schedules. Halide's indexing method also matches lattice indexing in stencils, and thus provides a natural expression of stencil-based computations.

6.1 Scope

We make several decisions, detailed below, which restrict the scope of our compiler to efficiently meet the objectives of our prototype design. We believe this compiler design and language description provide an effective starting point for future work to develop a full compiler for linear algebra on lattices. Specifically, we choose to:

1. Compile only unblocked linear algebra
2. Forbid Unstructured edge set declaration
3. Represent our matrices entirely assembly-free
4. Forbid matrix multiplications

We choose to compile only unblocked linear algebra (point 1) to demonstrate linear algebra of system-level vectors and matrices with a minimum of engineering complexity. There are many applications, including our Lattice QCD case study, which demand a linear algebra representation on a blocked vector space. We leave as future work development of a fully featured compiler of the language which can handle these applications.

We choose to compile only Lattice edge sets (point 2) since compilation of Unstructured edge sets does not demonstrate any additional behavior unique to our language. Our language does not include constructs for interaction between Unstructured and Lattice edge sets, and as a result any code generated for Unstructured edge sets would decouple from that of Lattice edge sets.

We choose to represent our matrices entirely assembly-free (point 3) because we can explore this space effectively using Halide scheduling and temporaries. For complex forms of matrices seen in typical applications, matrix construction can be rewritten by the user into a sequence of smaller matrix pieces applied to intermediate vectors. With Halide scheduling primitives, we can offer the choice of root or inlined computation of these intermediates, which simulates the same trade-off in redundant computation versus locality explored by assembled or assembly-free matrices.

Finally, we restrict our compiler to forbid matrix multiplications (point 4) because in our assembly-free form, sequential matrix-vector multiplication fully demonstrates the semantics of our matrix representation. Without assembly, we cannot build extern matrices, thus all matrices must eventually be multiplied into vectors. Any matrix-matrix multiplications can therefore be written in terms of several matrix-vector multiplications composed using intermediate vectors. Matrix-vector multiplications also capture the ability to schedule matrix-level linear algebra. We believe it would be valuable to explore a future extension to our language which exposes a choice between assembled and assembly-free matrices, but for engineering simplicity in the prototype leave this to future work.

6.1.1 Miscellaneous Restrictions

Beyond the restrictions in scope, we mention a few engineering restrictions that could be extended in future work:

1. We assume edge data are symmetric. The memory ordering form of edges implies a directionality: two indices specify the base and one index a direction of the edge. For simplicity of implementation, the prototype compiler assumes all data stored in this way are independent of edge direction. Regardless of whether the edge is accessed from its source or sink in a stencil, the same value is retrieved.

2. Passes that manipulate the internal representation (IR) are designed around a single externally visible entry function, and often do not take care to maintain state supporting multiple extern functions. This is a simple engineering constraint that should be removed in future development.

6.2 Modifications to the Simit Compiler

We make the following major changes to Simit's compilation model:

1. Extend the type system to support Lattice edge sets

2. Extend index variables to allow derived variables in index expressions
3. Extend the IndexedTensor form to allow offsets
4. Add lattice indexing syntax for use in stencil assembly functions
5. Introduce new lowering steps, and remove some existing lowering steps, specifically:
   (a) Add normalizing row indices in stencil assembly functions
   (b) Add inlining matrix assembly into matrix-vector multiplications
   (c) Add rewriting system-level assignments to index expressions
   (d) Replace lowering field accesses with a custom step
   (e) Replace lowering maps with a custom step
   (f) Remove lowering index expressions to tensor reads and writes
   (g) Remove lowering tensor reads and writes to memory accesses
6. Add a Halide backend for code generation
   (a) Lower variable assigns to Single Static Assignment with realization barriers
   (b) Produce Halide definitions for each index expression assignment
   (c) Generate a recursive assembly of C++ lambda functions

6.2.1 Extended Types

The Simit compiler uses the SetType construct to distinguish between edge sets and endpoint sets. To support Lattice edge sets, we add an additional `Kind` parameter to Simit's SetType. We distinguish between existing edge sets, which we now identify with `Kind::Unstructured`, and Lattice edge sets, which we identify with `Kind::LatticeLink`. Rather than maintaining a list of endpoint sets, Lattice sets are defined with an integer `dimensions` parameter, and a single `IndexSet` defining the underlying endpoint set.

6.2.2 Derived Index Variables

Derived index variables are variables which follow the iteration of a referenced index variable but over a subset of a different domain. For example, one could imagine an index variable i over the domain $(x, y) \in [0, 2] \times [0, 2]$, and a derived index variable Di over a subset of a larger domain $(x, y, 0) \subset (x, y, z) \in [0, 2] \times [0, 2] \times [0, 2]$. For every (x, y) accessed by i, Di accesses $(x, y, 0)$.

Derived index variables are motived by lowering stencil-based matrix assembly. As we will describe, we eventually transform all stencil assembly into index expressions. For stencil assembly, our index variables span the lattice space, corresponding to iterating over all possible local origins for the assembly function. To write a relative-lattice-indexed *endpoint* element in terms of this iteration domain, we simply access the element indexed by all iteration variables offset by the respective lattice offsets. To write a relative-lattice-indexed *Lattice edge* element in terms of this iteration domain, we must access the element indexed by all iteration variables, plus one constant index corresponding to the directional index, together offset by the respective lattice and directional offsets. To define that these indices over two different Simit domains correspond to a common underlying lattice iteration, we represent the Lattice edge set index as an index derived from the endpoint index.

Derived index variables are implemented in the prototype compiler as `IndexVar` objects which wrap the `IndexVar` which they derive from. In IR listings, a derived index variable over the variable `i` is conventionally written `Di`. They are understood to span the space of the variable they derive from, reshaped to provide constant zero indices as needed in the full domain. In the prototype compiler, Lattice edge derived index variables are handled as a special case. We imagine, however, that this form of index variable will provide a useful tool for future work on assembled matrices, in which the stencil index may be a dense iteration independent of the iteration over the lattice domain.

6.2.3 IndexedTensor Offsets

Stencil assembly involves translating relative-lattice-indexed tensor reads and writes to index expressions. In our prototype compiler, we translate away all tensor write offsets (Section 6.2.5). We thus only need to handle offsets in tensor reads, which are represented as IndexedTensors.

We extend the IndexedTensor IR node to store offsets as a list of Simit `Expr`s, each of which offsets the corresponding index of the tensor. Index variables with dense domains expect to be paired with scalar integer offsets. Index variables over a lattice expect to be paired with a vector of d integer offsets. In our prototype compiler, we disallow blocking, and as a result only find offsets of the latter form.

In IR listings, we represent IndexedTensors with offsets by appending `+<offset>` to the relevant indices. For example, a 2D lattice vector, `vec`, accessed using the lattice index `l` with 2D offset `[-1,1]` is written `vec(l+[-1,1])`.

6.2.4 Lattice Indexing Syntax

Lattice indexing syntax is represented as a SetRead IR construct in the prototype compiler. A SetRead tracks the referenced set as a Simit `Expr` and stores the indices as a list of `Expr`s. The SetRead IR node is defined as a high-level node, meaning that it should never reach the backend. Instead, SetRead expressions within stencil constructs are lowered to index expressions with the indices treated as relative lattice offsets of the relevant index variable.

In stencil assembly functions, we see two different forms of SetReads: SetReads of endpoint sets with imposed lattice structure, and Lattice edge sets. In the case of endpoint sets, SetRead indices translate directly to one offset per dimension of the imposed lattice structure. In the case of Lattice edge sets, SetReads have two sets of indices, one for the edge source and one for the edge sink, but are translated to one set of lattice offsets plus a single additional directional offset inferred from the difference in source and sink indices.

6.2.5 Lowering Passes

The Simit compiler executes a sequence of *lowering passes* on the internal representation of a program before passing it to a backend for code generation. The prototype compiler modifies the full Simit lowering structure by modifying existing passes, removing passes, and adding passes.

The goal of modifying the lowering sequence is to arrive at a final *Index Expression Assignment Form*, which is then passed to the Halide backend for code generation. We design this form around representing all linear algebra in terms of index expression values because these indices naturally translate to Halide `Func` indices during code generation. We first define the desired final form then discuss the modified lowering passes which take us there.

Index Expression Assignment Form is defined by the one core linear algebra statement understood by the Halide backend, an Index Expression Assignment. This statement is defined as a linear algebra operation assigning into a potentially blocked matrix or vector from an expression involving potentially blocked matrices and vectors which are contracted, scaled, and added together. In full generality, we can express all valid Index Expressions via a recursive definition [25]:

$$
\begin{aligned}
\text{ElementWiseAdd:}\ & ([I = i_1, ...i_n \cup R = r_1, ...r_n] \\
& \text{ElementWiseMult}(I \cup R) + \text{ElementWiseAdd}(I \cup R)) \\
\text{ElementWiseMult:}\ & ([I \cup R]\ \text{Contraction}(I \cup R) * \text{ElementWiseMult}(I \cup R)) \\
\text{Contraction:}\ & ([I \cup R]\ \text{Value}(E_1 \subset I \cup R, K \subset R) * \text{Value}(E_2 \subset I \cup R, K)), \\
& \text{where } E_1 \cap E_2 = E_2 \cap K = K \cap E_1 = \emptyset \\
& \text{and } E_1 \cup E_2 \cup K = I \cup R \\
\text{Value:}\ & (\text{Const}(\emptyset) \mid \text{Vector}(i) \mid \text{Matrix}(i, j) \mid \\
& \text{BlockedVector}(i, b_1, ...b_n) \mid \text{BlockedMatrix}(i, j, b_1, ...b_n) \mid \\
& \text{ElementWiseAdd}(I \cup R))
\end{aligned}
$$

In the above statement, I represents the set of free indices and R the set of

reduced indices. When assigned to a variable, the set of free indices of the value are required to match the type structure of the variable. The assignment specifies a reduction over the domains of the reduction variables, composing using a given reduction operator. In the case of the Simit and prototype compilers, addition is the only available reduction operator.

This structure essentially specifies that index expressions can be recursively constructed, with element-wise additions and multiplications pairing free or reduction indices of all terms, contractions pairing a subset of the available reduction indices and dividing up the remaining indices between terms, and specific values being indexed by a fixed number and type of indices. In the above expression, each specific index variable must index into the *same dimension* everywhere it is used.

To offer a concrete example of an Index Assignment Statement in the context of Lattice QCD, we demonstrate lowering a quark vector assigned to a multiplication between the Dirac matrix and another quark vector, plus another quark vector. For clarity, we write the full index structure of all Lattice QCD objects and match these index names in the lowered form.

$$
\begin{aligned}
\xi[x]^i_\alpha &= \sum_{y,j,\beta} M[x,y]^{i,j}_{\alpha\beta} * \psi[y]^j_\beta + \chi[x]^i_\alpha \\
\xi &= (I = x, i, \alpha \cup R = y, j, \beta \\
&\qquad \text{Contraction}_{M,\psi}(I \cup R) + \text{BlockedVector}_\chi(x, i, \alpha)) \\
\xi &= (I = x, i, \alpha \cup R = y, j, \beta \\
&\qquad (\text{BlockedMatrix}_M(x, y, i, j, \alpha, \beta) * \text{BlockedVector}_\psi(y, j, \beta)) \\
&\qquad + \text{BlockedVector}_\chi(x, i, \alpha))
\end{aligned}
$$

In terms of Simit IR, with blocking explicit, this would be written:

```
xi = (x,i,alpha  M(x,+y)(i,+j)(alpha,+beta) * psi(+y)(+j)(+beta)
                + chi(x)(i)(alpha));
```

Our prototype compiler disallows blocked forms, and as such we only handle structures of a simplified Index Expression Assignment Form which omits BlockedMatrix

and BlockedVector forms.

The broad strokes of rewriting the existing Simit lowering pipeline to achieve this form were to: remove any lowering beyond index expressions, replace lowering of maps to loops with lowering to index expression assignments, and some rewriting passes for stencil forms such that all matrix-vector multiplications are structured as gather stencils. Figure 6-1 diagrams the sequence of lowering passes used in the prototype compiler. We discuss specific lowering passes below.

Row Index Normalization

The prototype compiler chooses to represent a left-multiplication into a column vector by a *gather* stencil, rather than a scatter stencil. This allows efficient parallel scheduling and matches the form of `Func` definitions in Halide: one specifies which elements of other functions contribute to a single abstract parameterization of the defined `Func`. To produce this form, we specify the matrix in terms of the columns that contribute to a single row of the output.

We define a lowering pass, Row Index Normalization, which achieves this form by using translational invariance in the assembly function to shift all output tensor row indices to zero offset. There is a subtle detail that must be considered in this transformation: the user may have stored values derived from lattice indexing of either the Lattice edge set or endpoint set into local variables prior to using them in a tensor write. To handle these cases correctly, we choose to inline all temporary definitions into the right-hand side of tensor writes prior to applying Row Index Normalization. This ensures that all relative indexing is shifted simultaneously.

We demonstrate an example of Row Index Normalization below:

```
% Pre–transformation assembly statements
var tmp = nodes[0,0].a;
A(nodes[1,0],nodes[0,1]) = tmp + nodes[1,0].a + nodes[0,1].a;
A(nodes[0,1],nodes[1,0]) = tmp + links[0,0;1,0].b;

% Temporary–inlined statements
```

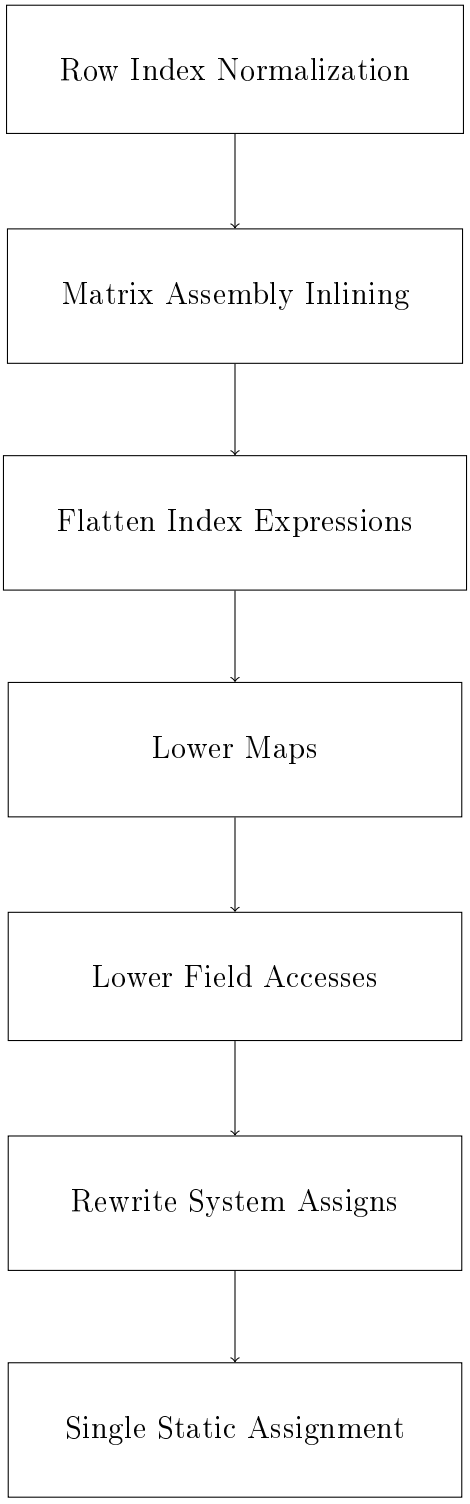

Figure 6-1: The set of lowering passes performed in the prototype compiler prior to Halide code generation.

```
A(nodes[1,0],nodes[0,1]) = nodes[0,0].a + nodes[1,0].a + nodes[0,1].a;
A(nodes[0,1],nodes[1,0]) = nodes[0,0].a + links[0,0;1,0].b;

% Shifted statements
A(nodes[0,0],nodes[−1,1]) = nodes[−1,0].a + nodes[0,0].a + nodes[−1,1].a;
A(nodes[0,0],nodes[1,−1]) = nodes[0,−1].a + links[0,−1;1,−1].b;
```

Matrix Assembly Inlining

After performing Row Index Normalization, all matrix writes are of the form:

```
A(nodes[0,0,...], nodes[i1,i2,...]) = ...;
```

At this point, the compiler has the choice of assembling the matrix directly or inlining the assembly into all uses. As discussed above, we choose to implement the latter in the prototype compiler.

In the Matrix Assembly Inlining pass, the compiler identifies all stencil definitions for matrices and simultaneously scans for and updates matrix uses, pattern-matching on left-multiplication of vectors by these matrices, while throwing errors on all other uses. This lowering pass operates after all linear-algebra has been translated to index expressions, so these left-multiplications always take the form of an indexed multiplication between a two-index object (the matrix) and a one-index object (the vector), with a reduction between the column index (right index) of the two-index object and the sole index of the one-index object. Written as Simit IR, a matrix multiplication between matrix `A` and vector `x` is `(j A(j,+i)*x(+i))`. Upon finding a left-multiplication, the compiler emits a temporary variable, defines it by a gather stencil, and replaces the multiplication with the temporary variable indexed by the row index of the matrix.

Building the gather stencil follows naturally from putting the stencil assembly function in Row Normalized form. In Row Normalized form, the column index of each tensor write in the assembly function dictates the offset of the multiplied vector to access, and the value of the tensor write dictates what value to multiply in. To

produce a given location of the output vector, we must sum all tensor writes using the multiplied-vector offsets and multiplied-in values dictated in this way. Conveniently, this can be written as another stencil of a *transformed* assembly function, with the multiplied-in vector passed as a partially-bound argument. We therefore choose to make this transformation by defining a transformed stencil kernel that performs the matrix-vector multiplication. This transformed kernel is rewritten to accept an additional vector argument and produce a *vector output*.

We make this more concrete by demonstrating a full example of inlining a stencil assembly function into a matrix-vector multiplication:

```
% Sets (links, nodes) and vectors (c,x,y) defined elsewhere
func assemble(links : lattice[2]{Link}(nodes),
              nodes : set{Node})
    -> (A : matrix[nodes,nodes](float))
  var tmp = nodes[0,0].a;
  A(nodes[1,0],nodes[0,1]) = tmp + nodes[1,0].a + nodes[0,1].a;
  A(nodes[0,0],nodes[1,0]) = tmp + links[0,0;1,0].b;
end

extern func main(x : vector[nodes](float),
                 c : vector[nodes](float))
  var A = map assemble to links;
  y = (j A(j,+i)*x(+i) + c(j)); % Ax + c as an index expression
end
```

Transformed using Row Index Normalization:

```
func assemble(links : lattice[2]{Link}(nodes),
              nodes : set{Node})
    -> (A : matrix[nodes,nodes](float))
  A(nodes[0,0],nodes[-1,1]) = nodes[-1,0].a + nodes[0,0].a
                            + nodes[-1,1].a;
```

```
  A(nodes[0,0],nodes[1,-1]) = nodes[0,-1].a + links[0,-1;1,-1].b;
end

extern func main(x : vector[nodes](float),
                 c : vector[nodes](float))
  var A = map assemble to links reduce +;
  y = (j A(j,+i)*x(+i) + c(j));
end
```

Transformed using Matrix Assembly Inlining:

```
func assembleAx(x : vector[nodes](float),
                links : lattice[2]{Link}(nodes),
                nodes : set{Node})
    -> (Ax : vector[nodes](float))
  Ax(nodes[0,0]) = ((nodes[-1,0].a + nodes[0,0].a + nodes[-1,1].a)
                    * x(nodes[-1,1]))
                  + ((nodes[0,-1].a + links[0,-1;1,-1].b)
                    * x(nodes[1,-1]));
end

extern func main(x : vector[nodes](float),
                 c : vector[nodes](float))
  var tmp : vector[nodes](float);
  tmp = map assembleAx(x) to links;
  y = (j tmp(j) + c(j));
end
```

Map Lowering

Though the prototype compiler does not support Unstructured edge sets, maps over endpoint sets are still valid. In the current Simit compiler, the map lowering pass

builds a loop nest over the domain of the set being mapped over and places the assembly function inside the loop nest with appropriate variable bindings. We modify this pass to achieve our desired Index Expression Assignment Form. Rather than generate a loop, the prototype compiler modifies this lowering stage to transform the assembly function in terms of an index variable which spans the domain of the endpoint set. To transform the assembly function, the prototype compiler:

1. Replaces all element accesses inside the kernel function with an IndexedTensor read from the appropriate set field. As an example, the original map lowering pass would replace a read of field `a` on element `p` in set `points` with a tensor load at the looped-over index: [`p.a`] → [`points.a[i]`]. In the prototype compiler, this instead becomes an indexed tensor: [`p.a`] → [`(i points.a(i))`], with no enclosing loop.

2. Replaces all output tensor writes inside the kernel function with an assign statement. This change relies on the the assigned value having previously been rewritten to an IndexedTensor. As an example, the original map lowering pass would replace a tensor write to the output variable with a tensor write at the loop index: [`out(p) = p.a`] → [`out[i] = points.a[i]`]. In the prototype compiler, this instead becomes a direct assign: [`out(p) = p.a`] → [`out = (i points.a(i))`].

As a result, we find all system-level operations resulting from a map are replaced with our desired Index Expression Assignment Form.

Stencil Lowering

In the case of maps over Lattice edge sets, we following a similar lowering form, but must additionally deal with:

- Having both a target set and neighbors set, as Lattice edge sets always come with an associated endpoint set

- Lattice indexing in the assembly function via SetReads

To handle accessing from two related sets, we need two *related* index variables for our stencil lowering: an index variable running over the Lattice edge set, and an index variable running over the endpoint set. In this case, we use our derived index technology to define our usual index variable for the endpoint set, and an additional derived index variable for indexing the Lattice edge set.

In a stencil assembly, lattice indexed fields are written as fields of SetReads. To lower these forms, the prototype compiler creates an IndexedTensor indexed by either the index or derived index, in the case of the endpoint or Lattice edge sets respectively. The constant integer indices of the SetRead are transformed to a single vector `Expr` and stored as an offset of the IndexedTensor. For the endpoint set, the compiler expects a number of indices equal to the lattice dimension, and converts these directly to a vector of that size as an offset. For the Lattice edge set, the compiler compares the set of indices for the source and sink of the edge, and requires that they be separated by exactly ± 1 in exactly one dimension. In our Lattice edge indexing convention, the numerically smaller index is designated as the base of the edge, with all edges conventionally pointing in a positive direction. Thus, in the case of a $+1$ offset in the ith direction, we write the derived index offset as [source offset, i], for the lattice and directional dimensions respectively. In the case of a -1 offset in the ith direction, we write the derived index offset as [sink offset, i].

To lower accesses to the output matrix of the assembly function, we pattern-match for all tensor writes to the output matrix variable indexed by a lattice indexed set element, e.g. `A(points[0,0],points[1,0]) = ...`. For simplicity of implementation, the prototype compiler demands that all writes to the output matrix take this form, disallowing, for example, a variable aliasing of the output variable prior to performing the write. By performing Row Normalization Indexing and Matrix Definition Inlining, the compiler guarantees that all writes reaching this lowering stage are to a vector-type output, and have zero offset. We thus directly replace this write by an assign where the right side contains all of the offsetting and indexing.

Extending our example of Section 6.2.5, we demonstrate lowering the stencil definition of the matrix multiplication to Index Expression Assignment form:

```
extern func main(x : vector[nodes](float),
                 c : vector[nodes](float))
  var tmp : vector[nodes](float);
  tmp = (i ((nodes.a(i+[−1,0]) + nodes.a(i+[0,0]) + nodes.a(i+[−1,1]))
            * x(i+[−1,1]))
         + ((nodes.a(i+[0,−1]) + links.b(Di+[0,−1,0]))
            * x(i+[1,−1])));
  y = (j tmp(j) + c(j));
end
```

In this example, we see lattice indexing lowered to index variables with offsets. In particular, the `[0,-1;1,-1]` offset was transformed to `[0,-1,0]`, because this link was pointing in the 0th direction, and the `0,-1` index was the base of the link. Had the offset been reversed, `[1,-1;0,-1]`, the offset would have been `[0,-1,0]` regardless. We also see the creation of the derived index variable, `Di`, for the Lattice link set. This index is defined to have the same iteration domain as `i` for the lattice coordinates (the first two), but remain constant in the dimensional index (the last one).

Field Access Lowering

Halide does not have any struct-type constructs, and as a result fields of Simit sets must be represented as independent Halide `Funcs`. To make this representation explicit, we replace set arguments to functions with individual field arguments. We create field arguments for field reads using a single dollar-sign notation, and field writes using a double dollar-sign notation. If, for example, a function reads fields `a` and `b` and writes field `c` of set `nodes`, the function would be rewritten to be explicitly parametrized by arguments `nodes$a`, `nodes$b`, and `nodes$$c`. During function argument binding, any set arguments are split up by fields and any read and written fields are bound based on the dollar-sign convention of parameter naming.

System-Level Assign Rewriting

To ensure all system-level assigns passed to the backend have index expressions as values, any system-level assigns and field writes are rewritten such that the right-hand values are index expressions indexed by variables covering their entire domain. These sorts of assigns appear in cases of variable copies, reads from fields into temporaries, and writes from temporaries back into fields. This lowering pass takes a fairly trivial form, simply inferring the domain of the assigned or written variable and enclosing it in an index expression.

6.2.6 Halide Code Generation

Once the Simit internal representation has been completely lowered to Index Expression Assignment Form, Halide code generation follows naturally from the index structure of each statement.

We perform code generation for a given Index Expression Assignment by assigning to the Halide Func associated with the left-hand-side variable, indexed by all the free indices, the expression generated from the Index Expression tree on the right-hand-side, indexed appropriately by all the free and reduced variables. Conveniently, the right-hand-side value can be generated by simply building Halide Expr objects corresponding to each Value, and combining them with C++ operators corresponding to each combining node, with the expected translation: ElementWiseAdd $\rightarrow +$, ElementWiseMult $\rightarrow \times$, Contraction $\rightarrow \times$.

During code generation, all Simit expressions are recursively compiled to HalideValue objects. We also maintain a symbol table mapping variables to HalideValue objects. HalideValue objects are a representation of an indexable Halide value: either a Halide Func, which can be indexed by all of the Halide Vars in its definition, or a Halide Expr, which has no indices. For bindable arguments, we choose to represent scalars as Halide Params, and vectors and matrices as Halide ImageParams, all of which are collected and passed to the runtime to be bound prior to execution. Halide Params and ImageParams can be cast to Halide Exprs and Funcs respectively, and

Simit IR

```
% a,b,y,z : vector
% x : scalar
y = (i a(i) + b(i));
x = (y(+j) * z(+j));
```

Halide generated code

```
Halide::Var i0, i1;
Halide::RDom j(pair<0, N0>, pair<0, N1>);
Halide::Func x,y;
y(i0,i1) = a(i0,i1) + b(i0,i1);
x() = Halide::sum(y(j.x,j.y) * z(j.x,j.y));
```

Figure 6-2: Halide code generation of endpoint set operations on a 2D lattice.

so fit neatly in the HalideValue model.

In code generation of indexed expressions, we translate all free indices to one or more Halide Vars, and all reduced indices to one or more Halide RVars. Each dense index is translated to a single Halide Var or RVar for a free or reduced index respectively. Each set index corresponding to a endpoint set with induced lattice structure is translated to d Halide Vars or RVars. In the case of a reduced lattice index, the ith RVar ranges over the domain $[0, N_i - 1]$, where N_i is a bindable Halide Param associated with the Lattice edge set inducing the structure. Each set index corresponding to a Lattice edge set is translated to $d + 1$ Halide Vars or RVars. The first d indices, representing the lattice domain, are translated identically to the endpoint set, while the last index is a dense Halide Var or RVar, ranging over the directional values of the links $\mu \in (0, ...d)$.

To make this concrete, we consider code generation for an endpoint set vector addition and inner product on a 2D lattice. The Simit IR and Halide generated code are shown in Figure 6-2. In Figure 6-3, we show the analogous case of vector addition and inner product for Lattice edge set vectors.

Single Static Assignment with Realization Barriers

Halide Func definitions permit only (a weakened form of) Single Static Assignment: a Halide Func may be defined any number of times prior to being used in an expression or being realized to memory, but may not be redefined once either of these events have taken place. This is in contrast with Simit's existing memory management, which is explicitly designed to compute and recompute values in place. In Simit, this design

Simit IR

```
% a,b,y,z : vector
% x : scalar
y = (i a(i) + b(i));
x = (y(+j) * z(+j));
```

Halide generated code

```
Halide::Var i0, i1, imu;
Halide::RDom j(pair<0, N0>, pair<0, N1>,
               pair<0, d>);
Halide::Func x,y;
y(i0,i1,imu) = a(i0,i1,imu)
             + b(i0,i1,imu);
x() = Halide::sum(y(j.x,j.y,j.z)
                * z(j.x,j.y,j.z));
```

Figure 6-3: Halide code generation of Lattice edge set operations on a 2D lattice. Note the extra μ indices, associated with edge directionality.

choice was made based on efficiency considerations: computing values in place is far more memory efficient than allocating a new variable per assignment, and as a result is often faster due to cache utilization [25, Sec 6].

Halide, however, focuses on defining input, ouputs, and intermediates as *separate stages* of a stencil pipeline, to allow manipulation of the schedule per stage. This was an important feature in our implementation of the prototype compiler, and following with this desired Halide form, the prototype compiler transforms all variable assignments to Single Static Assignment form, such that each variable corresponds to a stage in a Halide pipeline. To incorporate top-level control flow, the prototype compiler adds *realization barriers*, which specify points in the program where stages are realized to memory and thereafter drawn from the memory buffer.

Transformation from Simit's mutable variable semantics to our modified Single Static Assignment form is implemented in an additional backend-specific lowering pass applied to the internal representation. There are two regions in which the semantics of the transformation must be considered: entering a scope and within a scope. We discuss modified Single Static Assignment in terms of scopes rather than the more standard basic blocks because this more closely follows Simit's internal representation. In detail, we handle the two regions as follows:

1. When entering a scope, we must consider variables from the external scope that are both read and written. External variables that are read inside the

scope require no extra machinery: they are already in the symbol table as a HalideValue, and can be incorporated in inner computations as usual. External variables that are written require more careful handling. Assuming our scope represents a distinct basic block, we may or may not see the results of these writes. To allow this branching, we choose to add a realization barrier for every written variable immediately prior to entering the scope. At all points after the realization barrier, we treat the *memory buffer* as the definition of variable, leaving us free to update the memory if we branch into the scoped block, or leave the memory as-is otherwise.

2. Within a scope, we transform all variables to single assignment. In the prototype compiler, this follows standard generation-based single static assignment [11, Sec 5.2], with each successive reassignment, along with all of its downstream uses, transformed to a fresh variable. In addition, we take care to make each final-generation write to external variables visible. We do this by injecting a realization barrier immediately after every final-generation write of an external variable.

Realize statements are represented as IR statements in the prototype compiler. A Realize statement can take the form of single-variable realization, or a merge realization. In IR listings, a single-variable realization is written `realize x;` whereas a merge realization is written `realize target src;`.

Realize statements interact with *realization futures*. A realization future is defined as an object promising a valid `realize()` method at runtime which evaluates a Halide Func to a Halide Buffer. In the prototype compiler, these are implemented as objects with a handle to a Func and a Buffer. The Func handle is defined during compilation, while the Buffer handles are allocated and bound during function initialization.

Single-variable realizations cause the creation of a realization future with a handle to the Func associated with the variable, and the runtime-allocated Buffer associated with the same variable. Merge realizations cause the creation of a realization future with a handle to the `src` Func but the Buffer associated with `target`. This imple-

mentation makes use of the important fact that multiple realization futures may hold the handle to a common buffer, allowing branched Func definitions of the same buffer.

Control Flow and Realize Code Generation

Evaluating control flow must ultimately be phrased as a runtime realization of a Halide `Func`. To facilitate this, the prototype compiler restructures the IR in a backend-specific pass to rewrite conditions into temporary variable.

The prototype compiler then generates code for both control flow and realization barriers as top-level C++ `std::function` objects. These are emitted via lambdas with closures over a combination of other lambdas and realization futures. To code-generate a Realize statement, the prototype compiler simply emits a lambda function closed over the relevant realization future that calls the `realize()` method during execution. To code-generate control flow, the prototype compiler emits a lambda which realizes the condition variable, applies the respective C++ control flow statement over the condition value, and calls the relevant closed-over lambda for each branch. Finally, at the top level, each block of statements is condensed into a single function which iterates through, and executes, all generated lambdas in order.

We demonstrate single static assignment, control flow, and realization compilation through a concrete example of a while loop compiled to a single C++ `std::function` in Listings 6.1, 6.2, and 6.3.

6.2.7 Typedef Preprocessor

For convenience, we implement a Python preprocessor, which provides typedef resolution prior to Simit program compilation. In this extension, Simit typedefs take a form similar to C++: `typedef <expr> <name>;`

The Python preprocessor takes the simplest form possible, performing the following steps for text replacement:

1. Read program text by line, splitting on whitespace to build a list of tokens.

Listing 6.1: Simit code

```
proc main(x : vector[3](int))
  x = [1,2,3];
  iter = 0;
  while (iter < 5)
    x = x + [4,5,6];
    iter = iter + 1;
  end
  x = 2 * x;
end
```

Listing 6.2: Lowered Simit

```
proc main(x : vector[3](int))
  x = [1,2,3];
  var iter : int = 0;
  var cond : bool = iter < 5;
  realize x; % Single-variable realize
  realize iter; % Single-variable realize
  realize cond; % Single-variable realize
  while cond
    var x2 : vector[3](int);
    x2 = x + [4,5,6];
    realize x x2; % Merge realize
    var iter2 : int;
    iter2 = iter + 1;
    realize iter iter2; % Merge realize
    var cond2 : bool;
    cond2 = iter < 5;
    realize cond cond2; % Merge realize
  end
  var x2 : int;
  x2 = 2 * x;
  realize x x2;
end
```

Listing 6.3: C++ code

```
std::function<void()> whileBody = [x2Future, iter2Future,
                                   cond2Future](){
  x2Future.realize();
  iter2Future.realize();
  cond2Future.realize();
};

std::function<void()> whileLoop = [xFuture, iterFuture,
                                   condFuture, whileBody](){
  xFuture.realize();
  iterFuture.realize();
  condFuture.realize();
  while (condFuture.getBool()) {
    whileBody();
  }
};

std::function<void()> realizeX2 = [x2Future]() {
  x2Future.realize();
};

std::vector<function<void()>> block = {whileLoop, realizeX2};

// Top-level returned function which contains entire execution.
std::function<void()> top = [block]() {
  for (function<void()> f : block) {
    f();
  }
}
```

2. For lines consisting of exactly three tokens matching the typedef format given above add to a global map an entry from `<name>` to `<expr>`.

3. For any other lines, strip away line-ending comments and perform a regular expression replacement of the remaining text, choosing to delimit replaceable tokens by non-word characters. See Appendix C for a listing of the preprocessor code, including the full regular expression.

In future work, we imagine typedef resolution would be performed inside the parser, allowing restriction of replacements to type declarations only, as opposed to a broad text-matching replacement.

6.3 Exposing Scheduling Options

Simit provides a conveniently schedulable layer of internal representation in the form of indices. We allow scheduling of lattice code through manipulation of lattice indices specifically. Scheduling of lattice code expressed in index notation involves three steps:

1. Replace all lattice set indices with a set of indices over all dimensions of the lattice: `i` $\in$ `points` $\rightarrow$ `i1,...id` $\in [1, N_1], ...[1, N_d]$ and `j` $\in$ `links` $\rightarrow$ `j1,...jd,mu` $\in [1, N_1], ...[1, N_d], [1, d]$. Note that Lattice edge sets on the lattice are indexed by an extra directional index μ.

2. Split and reorder indices. In index expressions we are free to do this so long as we match the index structure on both sides of an element-wise operator such as assignment, element-wise multiplication, or element-wise addition, and match the index structure of reduced indices in a contraction.

3. Parallelize, vectorize, unroll or distribute indices.

In our prototype compiler, the first step is performed during code generation in the Halide backend. The second and third steps are exposed as options to the users by providing exposing an additional `setScheduling` method of the Simit `Function` class.

The `setScheduling` method accepts a map from compiled Halide `Func` names to scheduling commands. In its current form, this is limited to the following commands:

- `parallel <index> [<split>]`: Parallelize the index with the given name, optionally specifying a subproblem size into which to split the index before parallelization. This follows the form of the Halide `parallel()` method.
- `vectorize <var> <split>`: Vectorize the index with the given name, splitting the index into subproblems of the given size before vectorizing those subproblems. This follows the form of the Halide `vectorize()` method.
- `compute_root`: Compute and store at the root level.
- `compute_inline`: Compute inline at all uses.

The intended user workflow is to write the algorithm, perform an initial compilation and retrieve the listing of all generated Halide `Func` objects, and then experiment with scheduling of the intermediates and outputs to achieve performance on the target machine. We demonstrate an abbreviated example of a C++ frame code with scheduling:

```
simit::Function func = loadFunction("program.sim");


Set points;
// ... build set


Set springs(points,points);
// ... build set


// Schedule a matrix-vector multiplication
func.setScheduling({
  {"Ap", "compute_root"},
  {"Ap", "parallel d1"},
  {"Ap", "vectorize d0"}
```

```
});
func.bind("points", &points);
func.bind("springs", &springs);
func.runSafe();
```

While not implemented in this prototype, scheduling of linear algebra could similarly be expressed in terms of algebraic indices. We imagine a general set of scheduling options over an expanded form of Simit index expressions that incorporates both lattice and algebraic dimensions.

Chapter 7

Evaluation

We begin our evaluation by comparing outputs from our prototype compiler and the Simit compiler in two cases (Section 7.1). In the first case, we examine matrix assembly based on a 2D von-Neumann stencil. The Simit language can express this type of stencil in terms of an edge set assembly. We demonstrate that our language makes explicit the stencil form of the matrix assembly, and allows the prototype compiler to generate correct, index-free code because of this. In the second case, we examine matrix assembly based on a 3D star stencil. This stencil is motivated by the computationally intensive step of the Reverse Time Migration algorithm used in seismic simulation [39, 34]. We demonstrate that our language provides a simple, explicit description of the stencil form, which avoids building extra edge sets to express the matrix assembly.

We then analyze potential future impact of our designed language to the Lattice QCD domain (Section 7.2). We compare both the expressiveness of the language in describing Lattice QCD linear alegbra, and the performance of a manually written Halide program, representative of code that could be generated from our language in a future iteration of our compiler.

7.1 Common Stencils

We compare both von-Neumman stencil and 3D star stencil assembly in Simit and our language. We find that in the simple von-Neumann case, the Simit language can express the assembly but does not make explicit the form of the matrix to the user or compiler. The Simit compiler thus builds unnecessary memory indices which exhaust memory resources. Our language allows explicit representation of the stencil form of the assembly, and allows the prototype compiler to generate efficient code. In the 3D star stencil case, we find that the Simit language requires the user to jump through hoops to describe the assembly in terms of an edge set. This results in the Simit compiler constructing large extra memory structures. Our language provides the user a much more natural description of the stencil, and allows the prototype compiler to generate far more efficient code.

We evaluate these matrix assembly forms using a common matrix-multiplication frame. The syntax of the frame is common to both Simit and our language, and is the same in both the von-Neumman and 3D star stencil cases. This frame is demonstrated in Listing 7.1.

Listing 7.1: Test frame used for evaluation of matrix assembly.

```
extern func main()
  < assemble M >
  var iter = 0;
  while (iter < 100)
    points.a = M*points.a;
    iter = iter + 1;
  end
end
```

In performance comparisons, we excluded compile times and profiled specifically execution across the 100 iterations of matrix-vector multiplication described in the test frame. All performance comparisons were performed on one node of a 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster. Each node of the machine has

two sockets, with 12 cores each, and 128GB of memory.

7.1.1 2D von-Neumann Stencil

The 2D von-Neumann stencil involves memory accesses of all sites one hop away from the central point. In our language, this structure can be described entirely within the assembly function. We show the assembly function and map call used in Listing 7.2. In Simit, the user must build the lattice graph structure using the runtime library and specify matrix assembly in terms of the edge set representing the lattice links. We show the Simit assembly function and map call in Listing 7.3.

Listing 7.2: Assembly function and map used in 2D von-Neumann stencil assembly in our language.

```
func vonNeumann(l : lattice[2]{Link}(points), g : set{Point})
    -> (M : matrix[points,points](float))
  M(g[0,0],g[0,1]) = l[0,0;0,1].b;
  M(g[0,0],g[1,0]) = l[0,0;1,0].b;
  M(g[0,0],g[0,-1]) = l[0,0;0,-1].b;
  M(g[0,0],g[-1,0]) = l[0,0;-1,0].b;
end

< assemble M >: M = map vonNeumann to links;
```

Listing 7.3: Assembly function and map used in 2D von-Neumann stencil assembly in Simit.

```
func vonNeumann(l : Link, g : (Point*2))
    -> (M : matrix[points,points](float))
  M(g(0),g(1)) = l.b;
  M(g(1),g(0)) = l.b;
end

< assemble M >: M = map vonNeumann to links;
```

Size	Lattice Extensions	Simit	Comparison
100^2	10	11	1.1×
1000^2	1000	1041	1.0×
5000^2	30504	29500	1.0×

Table 7.1: Runtime comparison of von-Neumann stencil assembly on a variety of lattice sizes for our language compared to Simit. All runtimes are in milliseconds. The comparison column indicates how many times slower Simit was.

Size	Lattice Extensions	Simit	Comparison
100^2	0.10G	0.18G	1.7×
1000^2	0.19G	0.68G	3.5×
5000^2	2.45G	16.34G	6.7×

Table 7.2: Memory comparison of von-Neumann stencil assembly on a variety of lattice sizes for our language compared to Simit. All memory values are in gigabytes. The comparison column indicates how many times more memory Simit used.

The Simit assembly function does not describe the structure of the stencil, and as a result the Simit compiler produces memory indices to describe the assembled matrix structure. We compare runtimes and memory usage for both Simit and our language. Our results are shown in Tables 7.1 and 7.2.

We find that Simit consumes relatively more memory as the problem size scales. This matches our expectation, given the use of memory indices in the Simit compiler. We also find that our runtimes are comparable to Simit runtimes: in this small stencil, extra memory does not significantly affect performance.

7.1.2 3D Star Stencil

We also compare implementations of a 3D star stencil in our language and Simit. This stencil is described by accesses to points up to four hops away in each of the cardinal directions and therefore is described by 25 points. In our language, we describe this form entirely within the assembly function. We show the assembly function and map call used in Listing 7.4. In Simit, the user must build an additional edge set using the runtime library and specify matrix assembly in terms of the edge set representing

the lattice links. We show the Simit assembly function and map call in Listing 7.5.

Listing 7.4: Assembly function and map used in 3D star stencil assembly in our language.

```
func star(l : lattice[3]{Link}(points), g : set{Point})
    -> (M : matrix[points,points](float))
  % A future iteration of the prototype compiler should
  % support dense loops over lattice offsets.
  M(g[0,0,0],g[0,0,0]) = -1.0;
  M(g[0,0,0],g[1,0,0]) = 1.0;
  M(g[0,0,0],g[2,0,0]) = 1.0;
  M(g[0,0,0],g[3,0,0]) = 1.0;
  M(g[0,0,0],g[4,0,0]) = 1.0;
  M(g[0,0,0],g[-1,0,0]) = 1.0;
  M(g[0,0,0],g[-2,0,0]) = 1.0;
  M(g[0,0,0],g[-3,0,0]) = 1.0;
  M(g[0,0,0],g[-4,0,0]) = 1.0;
  M(g[0,0,0],g[0,1,0]) = 1.0;
  M(g[0,0,0],g[0,2,0]) = 1.0;
  M(g[0,0,0],g[0,3,0]) = 1.0;
  M(g[0,0,0],g[0,4,0]) = 1.0;
  M(g[0,0,0],g[0,-1,0]) = 1.0;
  M(g[0,0,0],g[0,-2,0]) = 1.0;
  M(g[0,0,0],g[0,-3,0]) = 1.0;
  M(g[0,0,0],g[0,-4,0]) = 1.0;
  M(g[0,0,0],g[0,0,1]) = 1.0;
  M(g[0,0,0],g[0,0,2]) = 1.0;
  M(g[0,0,0],g[0,0,3]) = 1.0;
  M(g[0,0,0],g[0,0,4]) = 1.0;
  M(g[0,0,0],g[0,0,-1]) = 1.0;
  M(g[0,0,0],g[0,0,-2]) = 1.0;
```

```
  M(g[0,0,0],g[0,0,-3]) = 1.0;
  M(g[0,0,0],g[0,0,-4]) = 1.0;
end

< assemble M >: M = map star to links;
```

Listing 7.5: Assembly function and map used in 3D star stencil assembly in Simit.

```
func star(s : Star, g : (Point*25))
    -> (M : matrix[points,points](float))
  % Defined in terms of an extra "star" edge set which connects
  % to all 25 points
  M(g(0),g(0)) = -1.0;
  for i in 1:25
        M(g(0),g(i)) = 1.0;
  end
end

< assemble M >: M = map star to links;
```

Again, the Simit assembly function does not describe the structure of the stencil. In this case, the Simit compiler must produce memory indices for a highly connected set, where each point has many neighbors due to the large stencil. We compare runtimes and memory usage for both Simit and our language in Tables 7.3 and 7.4.

In this case, we find that the significantly larger nature of the stencil favors the index-less approach. In particular, in Simit, the user is forced to construct a *high-cardinality edge set* to correctly describe the star stencil. In the Simit programming model, describing a matrix using this edge set involves a large neighbors list and results in a much higher memory usage and runtime cost due to indirection. This is most powerfully demonstrated in the 100^3 lattice case, in which the prototype compiler emits code which executes 40× faster and uses 80× less memory than the

Size	Lattice Extensions	Simit	Comparison
10^3	8	80	10×
50^3	236	8854	37.5×
100^3	1823	72588	39.8×

Table 7.3: Runtime comparison of star stencil assembly on a variety of lattice sizes for our language compared to Simit. All runtimes are in milliseconds. The comparison column indicates how many times slower Simit was.

Size	Lattice Extensions	Simit	Comparison
10^3	0.10G	0.05G	0.5×
50^3	0.12G	2.05G	17.8×
100^3	0.20G	16.19G	79.7×

Table 7.4: Memory comparison of star stencil assembly on a variety of lattice sizes for our language compared to Simit. All memory values are in gigabytes. The comparison column indicates how many times more memory Simit used.

Simit code.

7.1.3 Discussion

These results on common stencils powerfully demonstrate that a stencil description of matrix assembly on lattice-type graphs is more expressive and more efficient. Using a stencil description, one can describe more complex forms of assembly due to the additional *coordinate structure* of the lattice. Matrix assembly described as stencils can then be emitted as efficient index-less code by a compiler.

The difference in expressiveness and performance is exacerbated in large stencil cases. These cases are well-motivated by high-order discrete derivatives, such as the 3D discrete derivative used in the Reverse Time Migration algorithm. Our approach allows an efficient, high-level linear algebra description of these methods.

7.2 Lattice QCD Domain

We continue by evaluating the applicability of our methods to complex blocked stencils, such as those found in our motivating domain, Lattice QCD. As described in

Chapter 2, the linear algebra involved in Lattice QCD reduces to a few simple stencils over blocked values. Since our prototype compiler does not support blocking, we evaluate a manual Halide implementation representative of code generated from a future version of our compiler. We compare this implementation against existing USQCD library methods and a Simit implementation.

We find that our approach is on par with existing optimized libraries in small-block comparisons, but performs poorly in situations with large blocks. This is fundamentally related to our usage of Halide as a prototyping backend: Halide buffers are currently restricted to four dimensions, forcing us to implement blocking as unrolled computations outside Halide buffers. We believe our promising results in the small-block cases validate our methods and suggest future work built on top of a version of Halide extended to higher dimensionality, or a custom backend.

7.2.1 Description of the Application

Inversion of the Wilson action Dirac matrix is a representative example of the forms of linear algebra involved in Lattice QCD and is the performance bottleneck restricting larger-scale computation. Motivated by this, we use this application to compare both expressiveness and performance of our approach versus the general Simit language and the existing QOPQDP module of the USQCD libraries targeted at Lattice QCD simulation.

We specifically implemented an iterative Conjugate Gradient inversion of the Wilson action Dirac matrix applied to a point source term. In all cases, we performed a fixed 100 iterations of the Conjugate Gradient algorithm. The QOPQDP library does not provide a simple Wilson action inverter, instead including an LU factorization prior to the inversion, preconditioning the problem using even-odd subsets. The overall program follows the same form as our implementations, but introduces additional code complexity and initial runtime for the LU decomposition. In our evaluation, we factor our the preconditioning runtime for a fair comparison of the two methods.

Platform	Lines of Code
Simit	176
Lattice Extensions	160
QOPQDP (LU included)	380

Table 7.5: Lines of code required to implement the Conjugate Gradient solver for the Wilson action Dirac matrix in Simit, our language, and QOPQDP.

7.2.2 Simplicity of Expression

In our implementation of the Dirac matrix inversion, we find that both the Simit implementation and a description in our language result in programs of comparable size. Despite isolating specifically the non-preconditioning components of the QOPQDP implementation, we estimate the lines of code in the QOPQDP implementation as significantly higher than either implementation. We consider this a conservative estimate, as we exclude the lines of code required to implement libraries beneath QOPQDP that describe the element-wise linear algebra operations. We demonstrate this comparison in Table 7.5. Full listings of the code in our language and Simit are presented in Appendix D.

While lines of code are often a good approximation of simplicity of code, we think a more indicative statement is that QOPQDP is a very *rigid* implementation. It provides several inverters for specific kernels, but does not easily generalize to variations on these kernels. For example, in attempting to reconstruct a non-preconditioned Wilson inverter, we encountered several instances of dead code and unmaintained preprocessor branches, and were not able to produce a working non-preconditioned program.

We also note that while Simit and our language are on similar footing in terms of lines of code, Simit does not match the flexibility of the stencil assembly of our language in that it can only easily implement kernels which are of a von-Neumann stencil form. More complex structures require definition of higher-order edge sets on top of the lattice links. Not only does this require additional indexing, this also removes key pieces of the application to a runtime definition.

This comparison demonstrates the "sweet spot" of combing the powerful ideas of

the stencil assembly construct with system-level linear algebra to concisely express linear algebra on lattices.

7.2.3 Performance

We benchmarked the compared implementations by timing specifically the iterations of the Conjugate Gradient solver. We ignore overhead from memory setup and teardown, as for a real situation these overhead costs would be amortized over many uses of the Conjugate Gradient method in a single execution. As the performance critical section of current Lattice QCD programs, this operation is representative of performance on a whole Lattice QCD program.

We compare the Dirac matrix inversion on several lattice sizes, ranging from small 2^4 lattices to larger 64^4 lattices. We also make the comparison between different numbers of gauge colors, ranging from $N_c = 1$ to $N_c = 4$, which corresponds to small through large inner blocks. We additionally focused on the $N_c = 1$ case and demonstrated finding an optimal parallel schedule for the manual Halide code.

Again, all comparisons were evaluated on one node of a 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster. Each node of the machine has two sockets, with 12 cores each, and 128GB of memory.

A full table of the raw data collected in this comparison can be found in Appendix E. We specifically highlight several performance characteristics of the compared implementations:

- Non-viability of naive Simit for large lattices
- Scalability with size of the lattice
- Scalability with number of colors, N_c
- Gains from parallelization

Size	Simit	QOPQDP	Comp.
8	414	236	1.8×
16	8230	4006	2.1×
32	151838	69120	2.2×

(a) $N_c = 1$

Size	Simit	QOPQDP	Comp.
8	1625	366	4.4×
16	29207	7001	4.2×
32	470323	117928	4.0×

(b) $N_c = 2$

Size	Simit	QOPQDP	Comp.
8	3472	575	6.0×
16	57114	9806	5.8×
32	OOM	172010	-

(c) $N_c = 3$

Size	Simit	QOPQDP	Comp.
8	5560	876	6.3×
16	91426	17065	5.4×
32	OOM	272340	-

(d) $N_c = 4$

Figure 7-1: Comparison of the naive Simit and QOPQDP implementations. All times are in milliseconds, and the two entries marked with "OOM" indicate Simit ran out of memory on execution of these cases. The comparison column indicates how many times slower Simit was.

Non-Viability of Simit

We compare the implementations in QOPQDP and Simit without our lattice extensions and demonstrate that the additional memory costs make naive Simit non-viable for Lattice QCD applications, both because of poor runtimes and exhausting available memory resources. As shown in Figure 7-1, Simit performed more than 2× worse than the QOPQDP implementation in all cases larger than 8^4, and, in the 32^4 lattice for $N_c = 3$ and $N_c = 4$, ran out of memory and crashed. For the remaining comparisons, we focus on the manual Halide and QOPQDP implementations.

Scalability with Lattice Size

We demonstrate scaling of both the Halide and QOPQDP unscheduled implementations for a variety of lattice sizes in Figure 7-2. The data demonstrate a clear linear scaling in the size of the problem. This matches our expectation for an application dominated by a series of sparse matrix-vector multiplications.

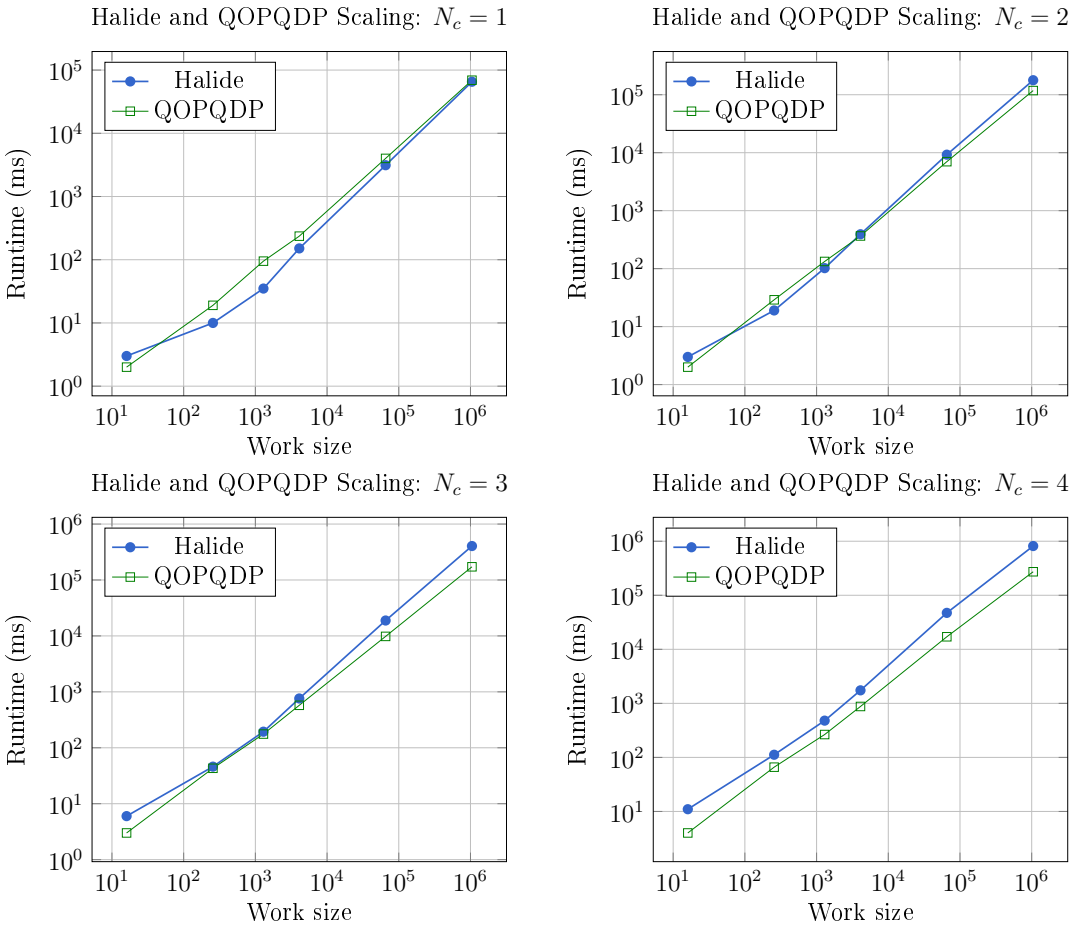

Figure 7-2: Scaling of the unscheduled Halide and QOPQDP implementations for $Nc = [1, 4]$. Lattice sizes evaluated were 2^4, 4^4, 6^4, 8^4, 16^4, and 32^4. This comparison demonstrates linear scaling in the size of the problem, as expected given the sparse nature of the Dirac matrix.

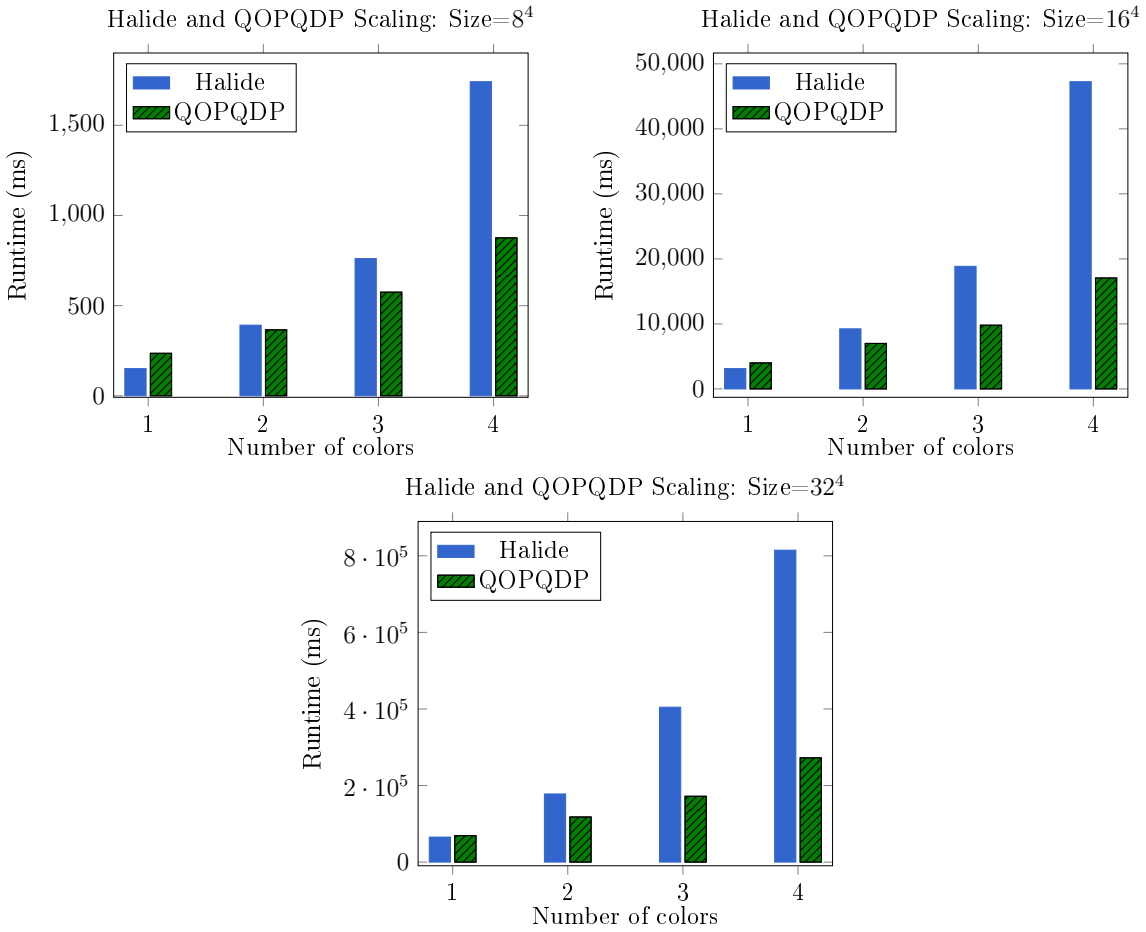

Figure 7-3: Scaling of the unscheduled Halide and QOPQDP implementations with respect to the number of colors on lattices of sizes 8, 16, and 32. This comparison demonstrates the weakness of the Halide backend to large inner blocks. We see competitive performance in the unblocked case corresponding to $N_c = 1$, but poor scaling due to a lack of memory locality.

Scalability with Number of Colors

For a given number of colors, N_c, the gluon field values on lattice links take the form of $N_c \times N_c$ matrices, while the quark field values on lattice sites take the form of $4 \times N_c$ vectors. The number of algebraic operations required to compute the Wilson action scales as the square of the number of colors, due to the gluon-matrix into quark-vector multiplications required for assembly at each site.

We demonstrate the comparisons of scaling in number of colors for unscheduled Halide and QOPQDP implementations in Figure 7-3. This comparison identifies a weakness of the Halide backend for regular grid computations: we are unable to

schedule dense linear algebra blocks inside lattice indices. This forced index ordering loses all locality in color index operations. In this case, we see a clear quadratic scaling in the Halide performance, corresponding to being limited by the non-locality of the color index, whereas QOPQDP scales linearly in colors, corresponding to scaling in the size of the quark vectors.

Gains From Parallelization

We isolate the $N_c = 1$, lattice size 32^4 case and demonstrate the gains offered by a flexible scheduling language. For this experiment, we isolated the matrix-vector multiplication of the Dirac matrix inversion and analyzed a large set of parallelization options averaged over 300 repetitions in each case. Specifically, we divided the z coordinate of the lattice into a variety of subtask sizes, ranging from 1 to 16, and evaluated the runtime for a spectrum of thread-pool sizes, ranging from 4 to 24. Figure 7-4 shows the threadpool size plotted against runtime for each subtask size. We find that the complete task division (subtask size 1) computed using 12 threads performs the best, giving gains of about $4\times$ over the single-threaded version. Note that we expect these properties to change based on lattice size, number of colors, and the machine specifications.

7.2.4 Implementation Details

We discuss details of the implementations in Simit, QOPQDP, and Halide below.

Simit implementation

The Simit implementation of Dirac matrix inversion is by far the easiest to understand, and as such we present it first. An abbreviated listing of the Simit code is shown in Figure 7-5. In particular, note that the Simit implementation represents lattice matrices and vectors as global objects in the main procedure, while isolating their definition to local assembly functions. In this comparison, the graph is externally initialized to a regular lattice structure, with toroidal boundary link connections

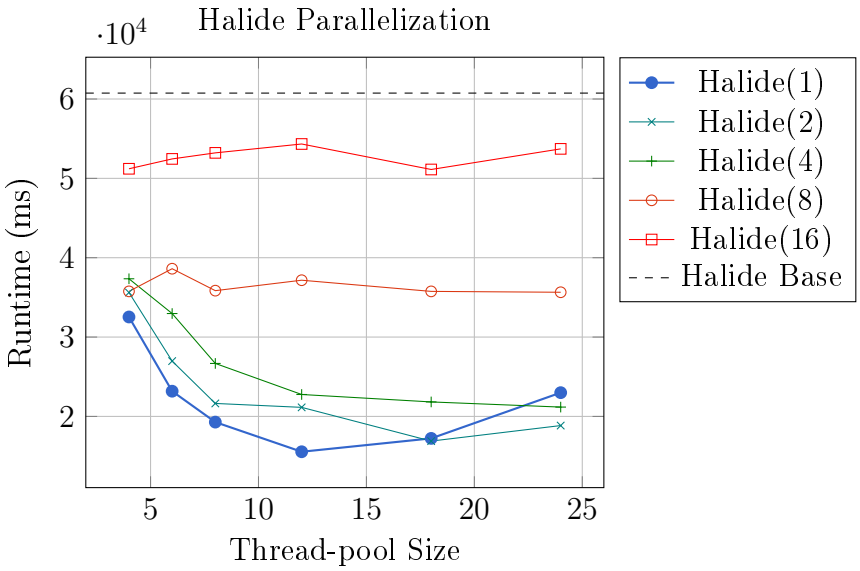

Figure 7-4: We evaluate 300 iterations of Dirac matrix-vector multiplications with $N_c = 1$ and lattice size 32^4 for a variety of thread-pool and subtask sizes and find that 12 threads with subtask size 1 performs the best.

in all four dimensions. We include the full Simit listing in Appendix D.

The Simit assembly of the Dirac matrix proceeds in two steps:

1. Assembly of the mass term, which is proportional to the identity and is thus a diagonal assembly map over the set of lattice points.

2. Assembly of the derivative and conjugated derivative term, which involve nearest-neighbor hops in all directions. In Simit, we describe these operations as maps over all links, writing down each link's contribution to the matrix elements between the two neighboring lattice points.

Following the assembly of the Dirac matrix (and conjugate), the Simit code follows a typical Conjugate Gradient solver process, running for a fixed 100 iterations and updating a solution vector in place.

While expressive, this Simit implementation suffers from memory overhead associated with actually assembling the Dirac matrix prior to running the Conjugate Gradient solver. In addition, Simit's materialization of an in-memory index for the gauge field and Dirac matrix is a further memory overhead. For the case of a 32^4 lattice, for example, the Simit implementation for $N_c = 3$ and $N_c = 4$ exhausted the

```
1  proc main
2    var src = map set_origin_src to fermions;
3
4    % Build Dirac matrix for our gauge config
5    var M_mass : matrix[fermions,fermions](matrix[Nct,Nct](gamma));
6    M_mass = map compute_mass_term to fermions reduce +;
7    var M_deriv_pos : matrix[fermions,fermions](matrix[Nct,Nct](gamma));
8    M_deriv_pos = map compute_deriv_term(<1.0,0.0>) to gauges reduce +;
9    var M_deriv_neg : matrix[fermions,fermions](matrix[Nct,Nct](gamma));
10   M_deriv_neg = map compute_deriv_term(<-1.0,0.0>) to gauges reduce +;
11
12   % Wilson action
13   M_pos = M_mass - M_deriv_pos;
14   M_neg = M_mass - M_deriv_neg;
15
16 % BEGIN CG SOLVE
17   const maxiters = 100;
18   var x = <1.0,0.0> * src;
19   var r = src - M_neg*(M_pos*x);
20   var p = r;
21   var iter = 0;
22
23   var tmpNRS = complexDot(r,r);
24   var rsq = complexNorm(tmpNRS);
25   var oldrsq = rsq;
26   while (iter < maxiters)
27     var beta = rsq/oldrsq;
28     oldrsq = rsq;
29     p = r + createComplex(beta,0.0)*p;
30
31     var Mp = M_neg*(M_pos*p);
32     var denom = complexDot(p,Mp); % p^{dag} M p
33     var denomReal = complexNorm(denom);
34     var alpha = rsq / denomReal;
35
36     x = x + createComplex(alpha,0.0)*p;
37     r = r - createComplex(alpha,0.0)*Mp;
38     var tmpNRS = complexDot(r,r);
39     rsq = complexNorm(tmpNRS);
40     iter = iter + 1;
41   end
42 % END CG SOLVE
43 end
```

Figure 7-5: The main procedure in the Simit implementation of Wilson action Dirac matrix Conjugate Gradient inversion.

128GB of memory available on our test machines and crashed. Simit's large memory structures, which do not fit into even the last-level cache of the test machine, cause memory overhead to translate into a runtime penalty beyond the expected penalty incurred from simple indirection.

QOPQDP implementation

The QOPQDP library provides an optimized implementation for the Wilson action Dirac matrix inversion. This implementation builds upon specifically tuned code for $N_c = 2$ and $N_c = 3$ linear algebra operations over the lattice provided in the QDP library module.

Notably, the QOPQDP implementation does not provide a method to perform a simple Conjugate Gradient inversion of the Dirac matrix, nor does it provide a method to run the Conjugate Gradient inverter for a fixed number of iterations. Instead, in this comparison, we used an existing Wilson inverter benchmark in the QOPQDP benchmark suite. This benchmark runs the Conjugate Gradient method to convergence repeatedly until a fixed number of iterations have been run. This incurs a small overhead from restarting the Conjugate Gradient solver multiple times, but this is ignored in the benchmark measurements, which profile the code by summing execution time within Conjugate Gradient iterations only. Additionally, the QOPQDP implementation differed from the other benchmarks in that it implemented an even-odd preconditioner before performing the Conjugate Gradient inversion. This implementation thus converged at a faster rate than the other comparison, but as we only compared runtimes per iteration of CG, and the QOPQDP benchmark matches the number of iterations performed in the other implementations, this does not affect our performance comparison.

Manual Halide implementation

In our manual Halide implementation of the Dirac matrix inversion we isolated each lattice linear algebra operation as a distinct Halide pipeline. These pieces were ahead-of-time compiled to C++ header and object files, which were compiled with a

frame code that allocated input and output buffers and called the appropriate Halide pipelines within the context of a Conjugate Gradient frame. The framework code included an overall timer to profile the Conjugate Gradient solve.

At the moment, Halide offers only 4 dimensional functions and buffers, and as a result the Halide implementation also manually unrolled the spinor and color dimensions. Halide does not offer blocking, and as such it was only possible to unroll these dimensions outermost. A quark field of dimensions $N_t \times N_x \times N_y \times N_z \times N_s \times N_c$, for example, corresponded to $N_s * N_c * 2$ Halide `Func`s parameterized by variables `t`, `x`, `y`, and `z`. Here N_s and N_c represent the sizes of the spinor and color dimensions respectively, and the factor of 2 appears because each field is a complex number. In our case, $N_s = 4$, while N_c ranged from 1 to 4. Color and spinor linear algebra was unrolled through loops over products of the Halide `Expr`s representing vectors and matrices of these dimensions.

We also manually implemented the "spin projection" algebraic optimization used by the QOPQDP for the Wilson derivative term. Recall that the nearest-neighbor hop terms of the Wilson action contain a multiplication by a the gamma matrix associated with the direction of the hop:

$$\bar{\psi}(x)(1 \pm \gamma_\mu)U_\mu(x)\psi(x+\mu)$$

In the chiral basis, the convention adopted by the USQCD libraries, these gamma matrices take the form [15]:

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

$$\gamma_2 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix} \qquad \gamma_3 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Taking γ_0 as an example, we can see that the $1 \pm \gamma_0$ terms that appear in the Wilson action produce redundant information when multiplied into an arbitrary spinor vector:

$$(1 \pm \gamma_0) \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \pm i \\ 0 & 1 & \pm i & 0 \\ 0 & \mp i & 1 & 0 \\ \mp i & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \pm id \\ b \pm ic \\ \mp ib + c \\ \mp ia + d \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \mp iv_2 \\ \mp iv_1 \end{pmatrix}$$

In fact, we need only compute two complex values rather than the full four for every gamma term of this form. It is useful to apply this technique before multiplying by U_μ, as this reduces the number of $N_c \times N_c$ matrix multiplications which are performed by half. After multiplying by U_μ, we must reconstruct the full 4-component spinor vector before performing the sum in all directions. In our example of $1 \pm \gamma_0$, we can choose to store only v_1 and v_2, multiply by U_0, then reconstruct the bottom two elements by multiplying the relevant v by $\mp i$.

The comparisons demonstrate that Halide allows generation of code that is competitive with existing library implementations where the blocking effects do not dominate. In addition, we demonstrated that in the case of $N_c = 1$, usage of a few Halide scheduling primitives allows identification of the optimal scheduling for a given machine. In our example, we gained a $4\times$ runtime improvement from parallelization by identifying the best subtask and thread-pool sizes for our machine. From these comparisons, we conclude that Halide is a viable prototyping backend for lattice linear algebra within Simit, and the system as a whole can produce code that is easily schedulable to match machine characteristics. We note that the lack of inner blocks restricted the performance in large block cases, and suggest that this feature be a focus of future work.

Chapter 8

Conclusion and Future Work

Our results show that the DSL approach to linear algebra on lattices is valuable. While maintaining clarity and flexibility of expression, we demonstrated a performant comparison to existing library approaches for our Lattice QCD case study. We also demonstrated a significant improvement over the naive Simit implementation for common stencils.

We identify four key elements of our design which allow us to demonstrate these strong results:

1. A stencil assembly construct for lattice graphs.
2. Use of Halide for quick prototyping of generated index-less stencil algorithms.
3. Exposing scheduling options based on lattice indices.
4. A focus on stencil-type matrices with small inner blocks.

The stencil assembly construct is the core of this work: it provides the user a means to define a regular stencil form of matrices which can be operated on efficiently. Specifically, the stencil structure allows the compiler to generate index-less representations of linear algebra which eliminate memory indirection. The result is code that accesses memory coherently and is amenable to vectorization and parallelization. We find this stencil-based matrix form used in a variety of physical simulation and image processing algorithms, and as such believe this work has broad applicability.

In our development of the prototype compiler, we focused on quick evaluation of our stencil description and index-less methods. Using Halide as a backend for our compiler allowed us to generate promising results in cases where the inner blocking was small. We however encountered inefficiencies in cases with large inner blocks, as the lack of high-dimensional Halide buffers forced us to unroll these block dimensions. We believe focusing on a future iteration of the compiler which supports inner-scheduled blocks will generate significantly better results in these cases, and recommend this as a future direction for this work.

Using the Halide backend, we were able to expose scheduling options to the user that allowed tuning of the generated code for performance on a specific machine. Key to providing scheduling options for linear algebra on lattices was an index expression representation. In our work, we focused on scheduling indices defined over lattice domains, and did not consider scheduling options over the full indexing structure. We feel this is another natural extension of this work, and expect to see additional improvements to the results from exposing this choice fully and making use of it in an optimized schedule.

There are several additional developments which we believe would broaden the applicability of this work:

- Adding a lattice subset feature. The even-odd preconditioner on lattice-type matrices requires operating on chessboard subsets of the lattice. We believe the correct approach to incorporating this feature would be develop it alongside a graph subsetting feature within Simit.

- Allowing non-toroidal boundary conditions as a structural choice for lattices. For applications outside of the demonstrated stencils and the Lattice QCD example, it may be helpful to apply a constant exterior or mirrored boundary condition instead of the default toroidal condition.

- Developing semantics for interaction between Lattice edge sets and Unstructured edge sets. As a motivating example, physical applications involving grid-based fluid interacting with irregular mesh boundaries would be benefited by

a unified representation of the physics and avoiding copy costs at the interface of the two systems. Our lattice extension to Simit sits in an ideal position to address this type of future challenge.

We see this work as having impact on both the scientific computing community and the compiler community.

Linear algebra on lattices manifests itself in several physics applications. We were motivated by a case study of Lattice QCD codes, in which the calculations are very costly while the algorithms are grounded fundamentally in blocked linear algebra. We hope that this initial work will be adopted by the Lattice QCD community to more quickly develop efficient code for future physical exploration. However, we also see this having applications to further areas of physics grounded in linear algebra on regular grids. In astrophysics, one approach to hydrodynamics simulations is to use a grid-based method which fits neatly into our model [53]. In Section 7.1.2, we demonstrated a particular discrete differencing kernel with applications to seismic simulations [39, 34]. Finally, weather simulations can also be described as stencils over lattices, and are often constrained by efficiency on supercomputers [49, 41]. By describing these problems using global linear algebra combined with local stencil patterns, we hope to enable efficient, flexible future development.

Beyond physical applications, this work is well-suited to describe the types of challenges faced in Markov Random Fields on grids. A significant application of this method is in low-level image inference [52]. This application centers around defining sparse matrices using stencil patterns and applying iterative solvers to generate inferences. Our language provides a natural description of this process in terms of linear algebra without sacrificing efficiency. In these applications, it is also often useful to schedule computation in tiled and parallelized ways, and we have demonstrated that our language is well-suited for descriptions of schedules separate from the core algorithm.

In our evaluations of linear algebra on lattices, we have thus far focused on the Conjugate Gradient method as a particular iterative solver. This method is described entirely in terms of linear algebra on the lattice in question, and thus fits into our

language naturally. In certain applications, the Conjugate Gradient method may be less well-suited for the problem, and other iterative solvers may be used. Multigrid methods are one particular class of alternative iterative solvers that involve smoothing of the data translated to coarser grids. We believe an interesting extension of this work would be focus on developing support for multiple lattice sizes, with application to the multigrid method in cases where data-parallel smoothers are used, such as in polynomial-smoothed multigrid [1].

In the compiler community, Simit has been successful at demonstrating a linear algebra DSL on arbitrary graphs. Our work extends this impact by exploiting specific structure of lattice graphs to make performance gains over Simit. Beyond this, we believe our work demonstrates a general method by which linear algebra scheduling can be discussed: in terms of the index expression representation produced by Simit, using the techniques of index-based scheduling similar to those employed in Halide.

Altogether, this work provides a launching point for future investigations into efficient computation on the specific class of linear algebra on lattices. This class of applications includes many forms of physical simulation as well as certain machine learning and solver techniques. We see continued development of efficient methods for these applications as a way to open doors in these various fields and enable both faster computation and development of new algorithms.

Appendix A

Quantum Field Theories

A.1 (Lagrangian) Theories

Lagrangian mechanics provides a convenient language with which to describe a physical system while treating time and space dimensions on an equal footing. The Standard Model describes physics on a relativistic spacetime, in which physics does not change under specific rotations between space and time coordinates. This relativistic invariance is a cornerstone of our current understanding of particle physics.

We begin with a description of a free particle, moving under Newton's laws. Recall that in this case, we expect the particle to move at a constant velocity forever. The Lagrangian specification for this problem requires us to write down an *action functional*, a description of the integrated kinetic energy of any path minus the integrated potential energy of the same path. Calling this action S, a functional of path and velocity functions, we have:

$$S(x(t), v(t)) = \int dt \mathcal{L}(x, v) = \int dt (KE - PE) \overset{\text{free particle}}{=} \int dt \frac{1}{2} m v^2$$

Now how does this action give rise to physics? In a classical Lagrangian theory, we additionally demand the *Principle of Stationary Action*. This is simply the requirement that any physical path must sit at a local minimum (or maximum) of the action functional, S. This demand gives rise to constraints that must be met for a

path to be considered physical. We term these constraints the *equations of motion*. A general expression for these equations of motions was derived by Euler and Lagrange [43, Sec 2.3]. For brevity, we skip the derivation, to arrive at the following constraint for any physical path of an arbitrary Lagrangian $\mathcal{L}$:

$$\frac{\delta \mathcal{L}}{\delta x} - \partial_t \frac{\delta \mathcal{L}}{\delta \partial_t x} = 0$$

For our free particle case, applying the functional derivatives (note that we treat v as independent of x in the functional) gives us:

$$-\partial_t(mv) = 0 \rightarrow mv = \text{const}$$

In other words, writing down the free particle Lagrangian, combined with the Principle of Stationary Action, tells us that the free particle moves with constant momentum (equivalently constant velocity), as expected.

A few important notes:

1. This solution is clearly incorrect in the relativistic limit, because there is no mention of a maximum velocity.

2. We explicitly integrated our Lagrangian over *time* specifically to give us the action. We wanted to specify physics without picking out time, but in this case the form of our path was a function of time only, forcing our hand. When we extend to fields which take values over all of time *and* space, we can formuate physics in a way that allows us to avoid singling out time.

A.2 (Lagrangian) Field Theories

The extension to physics of fields over relativistic spacetime requires an understanding first of the nature of that spacetime and second of the extension of Lagrangian physics to fields.

Minkowski spacetime, the spacetime of special relativity, is a variation on Euclidean 4-dimensional space. In 4D Euclidean space, vectors in the space may be specified by 4 cartesian coordinates, say (t, x, y, z). To take an inner product between two vectors, we multiply the corresponding coordinates and add them: $A \cdot_E B = A_t B_t + A_x B_x + A_y B_y + A_z B_z$. In Minkowski spacetime, we specify that the inner product instead incorporates the time dimension with the opposite sign: $A \cdot_M B = -A_t B_t + A_x B_x + A_y B_y + A_z B_z$. The overall sign is unimportant, but for consistency we choose $(-+++)$ throughout. Writing Minkowski vectors with greek indices that run over t, x, y, z, and using Einstein notation with summation implied, we have the following notation for the inner product of two Minkowski vectors $A \cdot_M B = A_\mu B^\mu$.

Using this compact notation, we are able to write down a Minkowski-space, field-based Lagrangian. Taking as an example a real scalar field, $\phi(x)$, where x is a point in Minkowski space, we can write down an analogy to the free particle above:

$$S = \int d^4x \mathcal{L}(\phi(x), \partial_\mu \phi(x)) = \int d^4x \left[-\frac{1}{2} (\partial_\mu \phi)(\partial^\mu \phi) \right]$$

Here we chose a minus sign on the kinetic term to have an overall positive sign on the time derivative component, in analogy to our free particle (this sign is a consequence of our overall sign choice in the Minkowski inner product). An important consequence of bundling all of our derivatives together into a Minkowski vector is that any transformations which leave the Minkowski inner product and integral measure invariant will not affect our physics. This is exactly the desired bundling of time and space into one relativistic object that we hoped for.

Taking a look at the equations of motion for this scalar field, we again find a constraint for every point on our "path", in this case the values of $\phi(x)$ for all x:

$$0 - \partial_\mu(-\partial^\mu \phi(x)) = 0 \rightarrow \partial_\mu \partial^\mu \phi(x) = 0$$

This is the Klein-Gordon equation of motion for the case of a massless field (a mass term could be further introduced in the Lagrangian as a potential energy, which would modify this equation) [55, Sec 1.1]. In this case, we find the real scalar solutions to

be $C\sin(x_\mu p^\mu)$, with $p_\mu p^\mu = 0$. Writing, without loss of generality, $p_\mu = (p_t, 0, 0, p_z)$, we find that the wave velocity $\frac{p_z}{p_t}$ is given by $-p_t^2 + p_z^2 = 0 \rightarrow \frac{|p_z|}{p_t} = 1$ in natural units. Reintroducing the speed of light, c, this is $\frac{|p_z|}{p_t} = c$, telling us that the physical configurations of this field are waves travelling at the speed of light. This matches our expectation of a massless object!

A.3 (Lagrangian) Quantum Field Theories

Finally, we introduce the last piece of framework needed to access the Standard Model: applying quantum mechanics to our Lagrangian field theory. One complete description of quantum mechanics follows from defining a Hilbert space over complex vectors, a Hamiltonian operator, the Schrödinger Equation, and sorting through the fallout [46, Chap 4]. While this description suits certain problems very well, Feynman's later *path integral* formalism corresponds much more closely to the computational methods of Lattice QCD. So far, we have been arriving at *classical* solutions to our physical systems, by means of the Euler-Lagrange equations of motion, all of which derived from demanding the Principle of Stationary Action. Feynman's path integral formalism states that this is only an approximation of the true *quantum* solutions [17]. These are instead given by integrating all configurations of our fields (*not* just the physical ones) weighted by the complex phase e^{iS}.

In other words, rather than picking out the stationary points of our action as physical, we apply our action as a complex phase to all field configurations. For configurations where the action S changes rapidly, small variations of the configuration mostly cancel with each other, whereas for configurations where S is relatively stable, small variations of the configuration sum mostly coherently. The result of this is sharp peaks around classical solutions with amplitudes smeared out to nearby solutions. We can write this all down in a simple equation for computing the "vacuum expectation value" for any "time ordered" quantum operator (a combination of fields and derivatives at various spacetime points):

$$\langle 0|T(\mathcal{O})|0\rangle = \frac{\int \mathcal{D}\phi \mathcal{O} e^{iS[\phi]}}{\int \mathcal{D}\phi e^{iS[\phi]}} = \frac{1}{Z}\int \mathcal{D}\phi \mathcal{O} e^{iS[\phi]}$$

Above, $Z = \int \mathcal{D}\phi e^{iS[\phi]}$ is the normalization of the vacuum in the absence of any operators. $\mathcal{D}\phi$ is a functional integral measure over configurations $\phi(x)$. The time ordering specifies that all products of fields and derivatives within $\mathcal{O}$ apply in order with the latest time leftmost to earliest time rightmost. This is relevant for an understanding of the physical meaning of the expectation value, as written on the left, but does not affect a calculation of this expectation value using the path integral form on the right.

For simple Lagrangians, we can in fact explicitly compute the path integral and find a closed form for our answer. It is illuminating to perform a simple calculation to get a feel for the path integral itself, and the type of results we are looking for. Let us compute the two-point correlator for the massless scalar free field that we wrote classically above:

$$C(x,y) = \langle 0|T(\phi(x)\phi(y))|0\rangle = \frac{1}{Z}\int \mathcal{D}\phi\phi(x)\phi(y)e^{iS[\phi]}$$
$$= \frac{1}{Z}\int \mathcal{D}\phi\phi(x)\phi(y)\exp\left(i*\int d^4x\left[-\frac{1}{2}(\partial_\mu\phi)(\partial^\mu\phi)\right]\right)$$

To perform this calculation, it is convenient to introduce a "source" term, $J(x)$, which has the same form as $\phi(x)$ (i.e. is a real scalar field) and is coupled to $\phi(x)$ in the Lagrangian. Using this source term, we can manipulate our integral, and finally set $J(x) = 0$ at the end. For conciseness, we define $\int_x \equiv \int d^4x$ and $\int_k \equiv \int \frac{d^4k}{(2\pi)^4}$, where k is the Fourier transform dual of x and our Fourier transform convention is to shove all of the 2πs into the k integral. Including our source manipulation, we have:

$$C(x,y) = \frac{1}{Z}\int \mathcal{D}\phi\phi(x)\phi(y)\exp\left(i*\int_x\left[-\frac{1}{2}(\partial_\mu\phi)(\partial^\mu\phi)+J(x)\phi(x)\right]\right)\Bigg|_{J=0}$$

Adding the source term allows us to rewrite the operator as derivatives in J:

$$= \frac{1}{Z}\int \mathcal{D}\phi \frac{-i\delta}{\delta J(x)}\frac{-i\delta}{\delta J(y)} \exp\left(i * \int_x \left[-\frac{1}{2}(\partial_\mu\phi)(\partial^\mu\phi) + J(x)\phi(x)\right]\right)\Big|_{J=0}$$

We can then complete the square in the exponent to remove cross-terms between ϕ and J, finally using invariance of the integral under shifts to find a neat form:

$$\begin{aligned}
&= (...) \exp\left(i * \int_x \left[\frac{1}{2}\phi(\partial_\mu\partial^\mu\phi) + J(x)\phi(x)\right]\right)\Big|_{J=0} \\
&= (...) \exp\left(i * \int_k \left[-\frac{1}{2}(\tilde{\phi}k^2\tilde{\phi}) + \tilde{J}(k)\tilde{\phi}(k)\right]\right)\Big|_{J=0} \\
&= (...) \exp\left(i * \int_k \left[-\frac{1}{2}(\tilde{\phi} - \frac{1}{k^2}\tilde{J})(k^2)(\tilde{\phi} - \frac{1}{k^2}\tilde{J}) + \frac{1}{2}\tilde{J}\frac{1}{k^2}\tilde{J}\right]\right)\Big|_{J=0} \\
&= \frac{1}{Z}\int \mathcal{D}\phi \frac{-i\delta}{\delta J(x)}\frac{-i\delta}{\delta J(y)} \exp\left(i * \int_k \left[-\frac{1}{2}\tilde{\phi}'(k^2)\tilde{\phi}' + \frac{1}{2}\tilde{J}\frac{1}{k^2}\tilde{J}\right]\right)\Big|_{J=0}
\end{aligned}$$

Performing the derivatives in J now gives us a value indepedent of ϕ, which we can remove from the integral and simplify completely:

$$\begin{aligned}
&= \frac{1}{Z}\int \mathcal{D}\phi \left(\int_k \frac{-ie^{ik(x-y)}}{k^2}\right) \exp\left(i * \int_k \left[-\frac{1}{2}\tilde{\phi}'(k^2)\tilde{\phi}' + \frac{1}{2}\tilde{J}\frac{1}{k^2}\tilde{J}\right]\right)\Big|_{J=0} \\
&= \left(\int_k \frac{-ie^{ik(x-y)}}{k^2}\right)\frac{Z}{Z} = \left(\int_k \frac{-ie^{ik(x-y)}}{k^2}\right)
\end{aligned}$$

This correlator tells us something similar to what we found classically. In the classical situation our plane wave solutions propagated with momenta constrained by $k^2 = 0$. Here we find a pole at $k^2 = 0$, but also non-zero correlations away from this classical solution. These are the quantum effects of incorporating the path integral playing a role. In more complex theories, it is of utmost importance to include the full quantum effects to calculate physical values. Yet, in more complex theories it becomes intractable to calculate the path integral explicitly.

A.4 Perturbation Theory

We have so far shown an example of a direct evaluation of a path integral. Reflecting back on this computation, it is clear that additional complexity in the Lagrangian can result in cases where we cannot perform the complete-the-square and integral shift steps to arrive at a Lagrangian form that isolates the source and field terms. As an example, we could imagine adding a $\phi(x)^4$ term to our scalar Lagrangian from earlier:

$$\mathcal{L}_\phi = \left[-\frac{1}{2}(\partial_\mu\phi)(\partial^\mu\phi)\right] - \left[\lambda\phi^4\right]$$

After performing the shift, we would be left with:

$$\frac{1}{Z}\int \mathcal{D}\phi \frac{-i\delta}{\delta J(x)}\frac{-i\delta}{\delta J(y)} \exp\left(i * \int_k \left[-\frac{1}{2}\tilde{\phi}'(k^2)\tilde{\phi}' + \frac{1}{2}\tilde{J}\frac{1}{k^2}\tilde{J} - \lambda(\tilde{\phi}' + \frac{1}{k^2}\tilde{J})^4\right]\right)\Bigg|_{J=0}$$

The ϕ^4 term results in remaining cross-terms between J and ϕ, and we cannot simply evaluate the J derivatives to produce a ϕ-independent value for the operator. In this case, we can instead take a different route, resulting in a *perturbative* expansion of our answer. If we pull the ϕ^4 bit out as an exponential of J derivatives prior to shifting, we have some breathing room:

$$\frac{1}{Z}\int \mathcal{D}\phi \frac{-i\delta}{\delta J(x)}\frac{-i\delta}{\delta J(y)} \exp\left(\int_x -\lambda\left[\frac{-i\delta}{\delta J(x)}\right]^4\right) \exp\left(i * \int_k \left[-\frac{1}{2}\tilde{\phi}'(k^2)\tilde{\phi}' + \frac{1}{2}\tilde{J}\frac{1}{k^2}\tilde{J}\right]\right)\Bigg|_{J=0}$$

Taylor expanding the exponential of J derivatives gives us something that looks like a complicated sum of operators evaluated in the free theory. We know how to evaluate any operator in the free theory, so this is tractable, *as long as our perturbative series converges*. This type of analysis can be extended to multiple fields, and many types of terms that may be added to the Lagrangian [45]. The end result is a set of rules allowing us to compute an operator in the full theory by an infinite sequence of terms in the free theory. The key point here is that we can achieve reasonable approximations of a physical value by computing only a few terms, if the perturbative series converges. Noting that at each subsequent order of the perturbative sequence

we pick up one more copy of λ, it is sufficient to have $\lambda \ll 1$.

Appendix B

Details of SU(3) Group and Algebra

The SU(3) group is at the heart of 3-color QCD physics. We briefly outline the concept of a *representation* of the SU(3) group, and specifically give concrete examples of bases of the adjoint and fundamental representations. In the case of QCD, these representations correspond to the gluon and quark fields respectively, however keep in mind that in Wilson's *Lattice* QCD formulation, the gluon field is exponentiated to give *fundamental* representation values on the links. Thus for the purposes of computation, we generally focus on entirely fundamental representation objects, though we can arrive at link values through exponentiation of a particular continuum gluon field if this is convenient.

B.1 SU(3) Group Definition

Mathematical groups are a set of elements, $G = \{g\}$, related by the following properties [50]:

1. An associative product operator, $\cdot : G \times G \to G$.

2. An identity element, e, which maps each element to itself for both left and right multiplication: $e \cdot g = g$ and $g \cdot e = g$.

3. An inverse for every element, such that $g \cdot g^{-1} = e$.

Importantly, the *structure* of the group elements under the product map defines the group, rather than any particular values for the elements. In practice, it is convenient to pin the structure of a group to a particular form of the elements, and in the case of SU(3) this is exactly how we proceed.

We define the SU(3) group by all complex 3×3 matrices that satisfy unitarity, $gg^\dagger = g^\dagger g = \mathbb{1}_3$, and unit determinant $\det g = 1$. In this definition, the group product is simply matrix products, $\det g = 1$ guarantees that an inverse exists, and the identity element is the identity matrix, $\mathbb{1}_3$.

One significant property of the SU(3) group is the fact that the action of an element of SU(3) on a complex 3-vector preserves the Hermitian inner product. For $\langle u, v\rangle = u_1^* v_1 + u_2^* v_2 + u_3^* v_3$, we have:

$$\langle g \cdot u, g \cdot v\rangle = \langle u, g^\dagger g \cdot v\rangle = \langle u, v\rangle$$

B.2 Representations of SU(3) and Particles

A representation of a group is defined as a vector space V, with associated linear operators for each group element $g \to U(g)$, such that $U(e) = \mathbb{1}$ and $U(g \cdot g') = U(g)U(g')$. One may hear this mapping to linear operators called the representation, or the vector space itself called the representation. Fundamentally, the mapping of group elements defines the representation, however in physics we are often interested in the vectors living in V as well, thus the confusing terminology.

We have already seen a representation of SU(3). The 3×3 matrix definition given above is the *fundamental* representation of SU(3). With this definition in mind, we can tackle what we mean when we say quarks live in the fundamental representation. In this case, we mean that quarks take 3-vector values, and should be acted on via matrix multiplication by group objects. Recalling that our quark term in the QCD Lagrangian looks like the following, we see that those group objects must be the gluons:

$$\mathcal{L}_{\text{quark}} = \bar{\psi}(i\gamma^\mu(\partial_\mu - igA_\mu) + m)\psi$$

In fact, we mentioned that gluons live in the *adjoint* representation, not the fundamental representation, so in reality the Lagrangian written above is shorthand for some sort of translation between the adjoint and fundamental representations. Let's first discuss what the adjoint representation means, and then return to how an adjoint representation object can act on the vector space of the fundamental representation.

The *adjoint* representation of SU(3) can be thought of intuitively as a tangent space to the group around the identity. A full description of the adjoint representation requires delving into Lie Algebras, but for the sake of brevity, we refer the reader to a more detailed introduction to groups and algebras [20], and move on to a more concrete description of the properties of the adjoint representation of SU(3). We will simply state that the adjoint representation operates on a vector space of dimension 8. With a little investigation, one can see that this is in fact the number of free parameters permitted by the matrix definition of the SU(3) group. Our gluons live in this 8-dimensional vector space, and can be written in index notation with a latin index $a \in [0, 7]$: $A_\mu \to A^a_{\text{adj},\mu}$, where previously the group structure was implied.

We are now in a position to discuss how an 8-dimensional vector value in this adjoint representation translates to a matrix action on the 3-vector quarks. Specifically, we write down 8 basis matrices which define the map. Harking back to the idea that the structure of a group is more important than particular values, in fact the structure of the Lie Algebra of SU(3) is intrinsically related to how these matrices commute, and this property is more important than the particular matrices. For our uses, however, we simply write down one example basis, the Gell-Mann basis [44, Chap 12]:

$$\lambda_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \lambda_2 = \begin{pmatrix} 0 & -i & 0 \\ i & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \lambda_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\lambda_4 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \qquad \lambda_5 = \begin{pmatrix} 0 & 0 & -i \\ 0 & 0 & 0 \\ i & 0 & 0 \end{pmatrix} \qquad \lambda_6 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\lambda_7 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -i \\ 0 & i & 0 \end{pmatrix} \qquad \lambda_8 = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -2 \end{pmatrix}$$

Our quark portion of the Lagrangian can now be written in a more explicit form:

$$\mathcal{L}_{\text{quark}} = \bar{\psi}(i\gamma^\mu(\partial_\mu \mathbb{1}_3 - igA^a_{\text{adj},\mu}\lambda_a) - m\mathbb{1}_3)\psi$$

In this notation, each component A^a acts as a coefficient to a 3×3 matrix λ_a (keep in mind the implied summation), and the result in parentheses is a matrix that acts between the row 3-vector $\bar{\psi}$ and the column 3-vector ψ, resulting in a scalar value.

In the Wilson formulation of Lattice QCD, ψ continues to be a color 3-vector in the fundamental representation, however we choose U_μ, instead of A_μ, and define it to be a fundamental representation 3×3 matrix. Thus in Lattice QCD computations, the values we are interested in are generally the matrices and vectors of the SU(3) fundamental representation.

Appendix C

Typedef Preprocessor Listing

The full code listing of the Typedef Preprocessor is given below. The preprocessor is designed to accept a `.sim` file as the first argument, and print the rewritten code to `stdout`. This is intended only as a prototype, to allow full handling of the example Simit programs, as listed in Appendix D.

```
### Simit preprocessor
### Substitute in typedefs as macros, in a text-for-text pattern matched
### manner. This is a standin for an actual typedef system.

import fileinput
import re
import string
import sys

typedefs = {}

def replace_tds(s):
    for td in typedefs:
        s = re.sub(r'(\W|^)%s(?=\W|$)' % (td),
                   r'\g<1>%s' % (typedefs[td]), s)
```

```
    return s


for line in fileinput.input():
    tokens = line.strip().split(" ")
    if len(tokens) == 3 and tokens[0] == "typedef":
        # Clean the semicolon
        ts = tokens[2].split(";")
        assert len(ts) == 2
        tokens[2] = ts[0]


        typedefs[tokens[2]] = replace_tds(tokens[1])
        sys.stderr.write("Found typedef %s <- %s\n"
                         % (tokens[2], tokens[1]))
    else:
        bits = line.split("%")
        if len(bits) > 1:
            # Sub typedefs in actual code
            bits[0] = replace_tds(bits[0])
            out = "%".join(bits)
        else:
            out = replace_tds(line)
        sys.stderr.write("Replaced:\n%s%s\n" % (line, out))
        print out,
}
```

Appendix D

Simit Lattice QCD Listing

We show the description of the Wilson action inversion application implemented in our language in Listing D.1. Because our prototype compiler does not support blocking, for evaluation we manually implemented a Halide program representing the form of code that would be emitted from a future iteration of our compiler for this application. We also demonstrate the application implemented in Simit in Listing D.2.

Listing D.1: Description of the representative Lattice QCD application in our language.

```
1  typedef vector[4](complex) spinor;
2  typedef matrix[4,4](complex) gamma;
3  % Abusing the typedef preprocessor
4  typedef <0.0,0.0> z;
5  % Nct externally defined based on the experiment in question
6
7  element Site
8    idx : vector[4](int); % Index label (t,x,y,z)
9  end
10
11 element Link
12   U : matrix[Nct,Nct](complex);
```

```
13    mu : int; % Directional index
14  end
15
16  extern fermions : set{Site};
17  extern gauges : lattice[4]{Link}(fermions);
18
19  % hopping param
20  const kappa : float = 0.1;
21  const gamma_ident : gamma = [<1.0,0.0>, z, z, z;
22                               z, <1.0,0.0>, z, z;
23                               z, z, <1.0,0.0>, z;
24                               z, z, z, <1.0,0.0>];
25  % QDP convenction gamma matrices:
26  % http://usqcd.jlab.org/usqcd-docs/qdp++/manual/node83.html
27  const gamma_0 : gamma = [z, z, z, <0.0,1.0>;
28                           z, z, <0.0,1.0>, z;
29                           z, <0.0,-1.0>, z, z;
30                           <0.0,-1.0>, z, z, z];
31  const gamma_1 : gamma = [z, z, z, <-1.0,0.0>;
32                           z, z, <1.0,0.0>, z;
33                           z, <1.0,0.0>, z, z;
34                           <-1.0,0.0>, z, z, z];
35  const gamma_2 : gamma = [z, z, <0.0,1.0>, z;
36                           z, z, z, <0.0,-1.0>;
37                           <0.0,-1.0>, z, z, z;
38                           z, <0.0,1.0>, z, z];
39  const gamma_3 : gamma = [z, z, <1.0,0.0>, z;
40                           z, z, z, <1.0,0.0>;
41                           <1.0,0.0>, z, z, z;
42                           z, <1.0,0.0>, z, z];
```

```
43
44 func build_mass_ident(m : complex) -> (MI : tensor[Nct,Nct](gamma))
45   MI = <0.0,0.0>;
46   for ii in 0:Nct
47     MI(ii,ii) = gamma_ident * m;
48   end
49 end
50
51 % Include hopping param at the promotion stage
52 func promote_gauge_spinor(sp : matrix[Nct,Nct](complex), gm : gamma)
53     -> (sp_gauge : matrix[Nct,Nct](gamma))
54   for ii in 0:Nct
55     for jj in 0:Nct
56       sp_gauge(ii,jj) = gm*sp(ii,jj)*createComplex(kappa,0.0);
57     end
58   end
59 end
60
61 func computeDirac(sign : complex,
62                   links : lattice[4]{Link}(sites),
63                   sites : set{Site}(sites))
64         -> (M : matrix[fermions,fermions](matrix[Nct,Nct](gamma)))
65     % In a future iteration of our language, this should be written
66     % a loop over direction mu.
67
68     % Get gamma projectors
69     var projForward0 : gamma = gamma_ident - sign*gamma_0;
70     var projBackward0 : gamma = gamma_ident + sign*gamma_0;
71     var projForward1 : gamma = gamma_ident - sign*gamma_1;
72     var projBackward1 : gamma = gamma_ident + sign*gamma_1;
```

```
73      var projForward2 : gamma = gamma_ident - sign*gamma_2;
74      var projBackward2 : gamma = gamma_ident + sign*gamma_2;
75      var projForward3 : gamma = gamma_ident - sign*gamma_3;
76      var projBackward3 : gamma = gamma_ident + sign*gamma_3;
77
78      % Mass term
79      M(sites[0,0,0,0],sites[0,0,0,0]) = build_mass_ident(<1.0,0.0>);
80
81      % Wilson derivative
82      % (1-gamma_mu) * U_mu
83      M(sites[0,0,0,0],sites[1,0,0,0])
84          = promote_gauge_spinor(link[0,0,0,0;1,0,0,0].U, projForward0);
85      M(sites[0,0,0,0],sites[0,1,0,0])
86          = promote_gauge_spinor(link[0,0,0,0;0,1,0,0].U, projForward1);
87      M(sites[0,0,0,0],sites[0,0,1,0])
88          = promote_gauge_spinor(link[0,0,0,0;0,0,1,0].U, projForward2);
89      M(sites[0,0,0,0],sites[0,0,0,1])
90          = promote_gauge_spinor(link[0,0,0,0;0,0,0,1].U, projForward3);
91      % (1+gamma_mu) * U_mu^dagger
92      M(sites[0,0,0,0],sites[-1,0,0,0])
93          = promote_gauge_spinor(
94              gauge_dagger(link[0,0,0,0;-1,0,0,0].U), projBackward0);
95      M(sites[0,0,0,0],sites[0,-1,0,0])
96          = promote_gauge_spinor(
97              gauge_dagger(link[0,0,0,0;0,-1,0,0].U), projBackward1);
98      M(sites[0,0,0,0],sites[0,0,-1,0])
99          = promote_gauge_spinor(
100             gauge_dagger(link[0,0,0,0;0,0,-1,0].U), projBackward2);
101     M(sites[0,0,0,0],sites[0,0,0,-1])
102         = promote_gauge_spinor(
```

```
103             gauge_dagger(link[0,0,0,0;0,0,0,-1].U), projBackward3);
104 end
105
106 % Build a point source at the origin
107 func set_origin_src(p : Site)
108     -> (src : vector[fermions](vector[Nct](spinor)))
109   if (p.idx(0) == 0 and p.idx(1) == 0 and
110       p.idx(2) == 0 and p.idx(3) == 0)
111     for ii in 0:Nct
112       for jj in 0:4
113         src(p)(ii)(jj) = <1.0,0.0>;
114       end
115     end
116   else
117     for ii in 0:Nct
118       for jj in 0:4
119         src(p)(ii)(jj) = <0.0,0.0>;
120       end
121     end
122   end
123 end
124
125 proc main
126   var src : vector[fermions](vector[Nct](spinor));
127   src = map set_origin_src to fermions;
128
129   % Wilson action Dirac matrix
130   M_pos = map computeDirac(<1.0,0.0>) to gauges;
131   M_neg = map computeDirac(<-1.0,0.0>) to gauges;
132
```

```
133 % BEGIN CG SOLVE
134   const maxiters = 100;
135   var x = <1.0,0.0> * src;
136   var r = src - M_neg*(M_pos*x);
137   var p = r;
138   var iter = 0;
139
140   var tmpNRS = complexDot(r,r);
141   var rsq = complexNorm(tmpNRS);
142   var oldrsq = rsq;
143   while (iter < maxiters)
144     var beta = rsq/oldrsq;
145     oldrsq = rsq;
146     p = r + createComplex(beta,0.0)*p;
147
148     var Mp = M_neg*(M_pos*p);
149     var denom = complexDot(p,Mp); % p^{dag} M p
150     var denomReal = complexNorm(denom);
151     var alpha = rsq / denomReal;
152
153     x = x + createComplex(alpha,0.0)*p;
154     r = r - createComplex(alpha,0.0)*Mp;
155     tmpNRS = complexDot(r,r);
156     rsq = complexNorm(tmpNRS);
157     iter = iter + 1;
158   end
159 % END CG SOLVE
160 end
```

Listing D.2: Implementation of the representative Lattice QCD application in Simit.

```
1  typedef vector[4](complex) spinor;
2  typedef matrix[4,4](complex) gamma;
3  % Abusing the typedef preprocessor
4  typedef <0.0,0.0> z;
5  % Nct externally defined based on the experiment in question
6
7  element Site
8    idx : vector[4](int); % Index label (t,x,y,z)
9  end
10
11 element Link
12   U : matrix[Nct,Nct](complex);
13   mu : int; % Directional index
14 end
15
16 extern fermions : set{Site};
17 extern gauges : set{Link}(fermions, fermions);
18
19 % hopping param
20 const kappa : float = 0.1;
21 const gamma_ident : gamma = [<1.0,0.0>, z, z, z;
22                              z, <1.0,0.0>, z, z;
23                              z, z, <1.0,0.0>, z;
24                              z, z, z, <1.0,0.0>];
25 % QDP convenction gamma matrices:
26 % http://usqcd.jlab.org/usqcd-docs/qdp++/manual/node83.html
27 const gamma_0 : gamma = [z, z, z, <0.0,1.0>;
28                          z, z, <0.0,1.0>, z;
29                          z, <0.0,-1.0>, z, z;
30                          <0.0,-1.0>, z, z, z];
```

```
31 const gamma_1 : gamma = [z, z, z, <−1.0,0.0>;
32                         z, z, <1.0,0.0>, z;
33                         z, <1.0,0.0>, z, z;
34                         <−1.0,0.0>, z, z, z];
35 const gamma_2 : gamma = [z, z, <0.0,1.0>, z;
36                         z, z, z, <0.0,−1.0>;
37                         <0.0,−1.0>, z, z, z;
38                         z, <0.0,1.0>, z, z];
39 const gamma_3 : gamma = [z, z, <1.0,0.0>, z;
40                         z, z, z, <1.0,0.0>;
41                         <1.0,0.0>, z, z, z;
42                         z, <1.0,0.0>, z, z];
43
44 func build_mass_ident(m : complex) -> (MI : matrix[Nct,Nct](gamma))
45   MI = <0.0,0.0>;
46   for ii in 0:Nct
47     MI(ii,ii) = gamma_ident * m;
48   end
49 end
50
51 func compute_mass_term(site : Site)
52     -> (M_mass : matrix[fermions,fermions](matrix[Nct,Nct](gamma)))
53   % Unit mass, using hopping form
54   M_mass(site, site) = build_mass_ident(<1.0,0.0>);
55 end
56
57 func gamma_dagger(g : gamma) -> (g_dag : gamma)
58   for ii in 0:4
59     for jj in 0:4
60       % Transpose conjugate
```

```
61        g_dag(jj,ii) = complexConj(g(ii,jj));
62      end
63    end
64  end
65
66  func gauge_dagger(U : matrix[Nct,Nct](complex))
67      -> (U_dag : matrix[Nct,Nct](complex))
68    for ii in 0:Nct
69      for jj in 0:Nct
70        U_dag(jj,ii) = complexConj(U(ii,jj));
71      end
72    end
73  end
74
75  % Include hopping param at the promotion stage
76  func promote_gauge_spinor(sp : matrix[Nct,Nct](complex), gm : gamma)
77      -> (sp_gauge : matrix[Nct,Nct](gamma))
78    for ii in 0:Nct
79      for jj in 0:Nct
80        sp_gauge(ii,jj) = gm*sp(ii,jj)*createComplex(kappa,0.0);
81      end
82    end
83  end
84
85  func compute_deriv_term(sign : complex, link : Link, sites : (Site*2))
86          -> (M_deriv : matrix[fermions,fermions](matrix[Nct,Nct](gamma)))
87      % Get gamma projectors
88      % No good switch structure, so fold gammas into a single vector
89      var gamma_mu : gamma;
90      if (link.mu == 0)
```

```
91        gamma_mu = gamma_0;
92      else if (link.mu == 1)
93        gamma_mu = gamma_1;
94      else if (link.mu == 2)
95        gamma_mu = gamma_2;
96      else if (link.mu == 3)
97        gamma_mu = gamma_3;
98      end
99      end
100     end
101     end
102     var projForward : gamma = gamma_ident - sign*gamma_mu; % 1 - gamma_mu
103     var projBackward : gamma = gamma_ident + sign*gamma_mu; % 1 + gamma_mu
104
105     % Wilson derivative
106     % (1-gamma_mu) * U_mu
107     M_deriv(sites(0), sites(1))
108       = promote_gauge_spinor(link.U, projForward);
109     % (1+gamma_mu) * U_mu^dagger
110     M_deriv(sites(1), sites(0))
111       = promote_gauge_spinor(gauge_dagger(link.U), projBackward);
112 end
113
114 % Build a point source at the origin
115 func set_origin_src(p : Site)
116     -> (src : vector[fermions](vector[Nct](spinor)))
117   if (p.idx(0) == 0 and p.idx(1) == 0 and
118       p.idx(2) == 0 and p.idx(3) == 0)
119     for ii in 0:Nct
120       for jj in 0:4
```

```
121           src(p)(ii)(jj) = <1.0,0.0>;
122         end
123       end
124     else
125       for ii in 0:Nct
126         for jj in 0:4
127           src(p)(ii)(jj) = <0.0,0.0>;
128         end
129       end
130     end
131 end
132
133 proc main
134   var src : vector[fermions](vector[Nct](spinor));
135   src = map set_origin_src to fermions;
136
137   % Build Dirac matrix for our gauge config
138   var M_mass : matrix[fermions,fermions](matrix[Nct,Nct](gamma));
139   M_mass = map compute_mass_term to fermions reduce +;
140   var M_deriv_pos : matrix[fermions,fermions](matrix[Nct,Nct](gamma));
141   M_deriv_pos = map compute_deriv_term(<1.0,0.0>) to gauges reduce +;
142   var M_deriv_neg : matrix[fermions,fermions](matrix[Nct,Nct](gamma));
143   M_deriv_neg = map compute_deriv_term(<-1.0,0.0>) to gauges reduce +;
144
145   % Wilson action
146   M_pos = M_mass - M_deriv_pos;
147   M_neg = M_mass - M_deriv_neg;
148
149 % BEGIN CG SOLVE
150   const maxiters = 100;
```

```
151   var x = <1.0,0.0> * src;
152   var r = src - M_neg*(M_pos*x);
153   var p = r;
154   var iter = 0;
155
156   var tmpNRS = complexDot(r,r);
157   var rsq = complexNorm(tmpNRS);
158   var oldrsq = rsq;
159   while (iter < maxiters)
160     var beta = rsq/oldrsq;
161     oldrsq = rsq;
162     p = r + createComplex(beta,0.0)*p;
163
164     var Mp = M_neg*(M_pos*p);
165     var denom = complexDot(p,Mp); % p^{dag} M p
166     var denomReal = complexNorm(denom);
167     var alpha = rsq / denomReal;
168
169     x = x + createComplex(alpha,0.0)*p;
170     r = r - createComplex(alpha,0.0)*Mp;
171     tmpNRS = complexDot(r,r);
172     rsq = complexNorm(tmpNRS);
173     iter = iter + 1;
174   end
175 % END CG SOLVE
176 end
```

Appendix E

Lattice QCD Raw Data

Tables E.1 and E.2 demonstrate the raw data collected to show a comparison of an optimized USQCD implementation, provided in the QOPQDP module, a naive Simit implementation with no handling of the regular grid structure, and a manual Halide implementation. The comparison is split up by the number of colors, ranging from $N_c = 1$ to $N_c = 4$. All runtimes as in milliseconds. All performance comparisons were executed on one node of a 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster, with two sockets, 12 cores each, and 128GB of memory.

Implementation	Size	Runtime
Simit	8^4	414
	16^4	8230
	32^4	151838
Halide (no sched.)	2^4	3
	4^4	10
	6^4	35
	8^4	151
	16^4	3126
	32^4	65373
Halide (vectorized t)	2^4	4
	4^4	10
	6^4	38
	8^4	172
	16^4	2992
	32^4	67307
QOPQDP (no sched.)	2^4	2
	4^4	19
	6^4	95
	8^4	236
	16^4	4006
	32^4	69120
QOPQDP (SSE, blocked 4)	8^4	158
	16^4	2720
QOPQDP (SSE, best blocking)	2^4	1
	4^4	10
	6^4	37
	8^4	127
	16^4	2314

(a) $N_c = 1$

Implementation	Size	Runtime
Simit	8^4	1625
	16^4	29207
	32^4	470323
Halide (no sched.)	2^4	3
	4^4	19
	6^4	102
	8^4	392
	16^4	9243
	32^4	177981
Halide (vectorized t)	2^4	6
	4^4	20
	6^4	118
	8^4	405
	16^4	9339
	32^4	181889
QOPQDP (no sched.)	2^4	2
	4^4	29
	6^4	133
	8^4	366
	16^4	7001
	32^4	117928
QOPQDP (SSE, blocked 4)	8^4	298
	16^4	5298
QOPQDP (SSE, best blocked)	2^4	2
	4^4	15
	6^4	77
	8^4	267
	16^4	5021

(b) $N_c = 2$

Table E.1: $N_c = 1, 2$ demonstrations of performance of naive Simit, a manual Halide code, and the QOPQDP library module.

Implementation	Size	Runtime
Simit	8^4	3472
	16^4	57114
	32^4	OOM
Halide (no sched.)	2^4	6
	4^4	46
	6^4	194
	8^4	761
	16^4	18856
	32^4	404605
Halide (vectorized t)	2^4	8
	4^4	52
	6^4	248
	8^4	756
	16^4	19282
	32^4	344788
QOPQDP (no sched.)	2^4	3
	4^4	43
	6^4	176
	8^4	575
	16^4	9806
	32^4	172010
QOPQDP (SSE, blocked 4)	8^4	492
	16^4	10009
QOPQDP (SSE, best blccked)	2^4	2
	4^4	25
	6^4	141
	8^4	454
	16^4	8269

(a) $N_c = 3$

Implementation	Size	Runtime
Simit	8^4	5560
	16^4	91426
	32^4	OOM
Halide (no sched.)	2^4	11
	4^4	112
	6^4	479
	8^4	1742
	16^4	47254
	32^4	814823
Halide (vectorized t)	2^4	11
	4^4	133
	6^4	574
	8^4	1272
	16^4	32760
	32^4	556471
QOPQDP (no sched.)	2^4	4
	4^4	66
	6^4	265
	8^4	876
	16^4	17065
	32^4	272340
QOPQDP (SSE, blocked 4)	8^4	832
QOPQDP (SSE, best blocked)	2^4	4
	4^4	42
	6^4	231
	8^4	772
	16^4	15378

(b) $N_c = 4$

Table E.2: $N_c = 3, 4$ demonstrations of performance of naive Simit, a manual Halide code, and the QOPQDP library module.

Bibliography

[1] M. Adams, M. Brezina, J. Hu, and R. Tuminaro. "Parallel multigrid smoothing: polynomial versus Gauss-Seidel". *Journal of Computational Physics*, 188(2):593 – 610 (2003). doi:10.1016/S0021-9991(03)00194-3.

[2] C. Allton. "Recent lattice QCD results from the UKQCD collaboration". *Comput. Phys. Commun.*, 142:168–171 (2001). doi:10.1016/S0010-4655(01)00317-4.

[3] C. R. Allton, W. Armour, D. B. Leinweber, A. W. Thomas, and R. D. Young. "Chiral and continuum extrapolation of partially-quenched lattice results". *Phys. Lett.*, B628:125–130 (2005). doi:10.1016/j.physletb.2005.09.020.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. "LAPACK Users' Guide", chapter 1, 3–8. doi:10.1137/1.9780898719604.ch1.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, D. Kaushik, M. G. Knepley, L. C. McInnes, W. D. Gropp, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. "PETSc Users Manual". Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory (2016).

[6] A. Bazavov et al. "Update on the 2+1+1 flavor QCD equation of state with HISQ". *PoS*, LATTICE2013:154 (2014).

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. "ScaLAPACK User's Guide" (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997).

[8] R. Brower, N. Christ, F. Karsch, J. Kuti, P. Mackenzie, J. Negele, D. Richards, M. Savage, and R. Sugar. "Computational Resources for Lattice QCD: 2015-2019". Technical report (2013).

[9] G. Brussino and V. Sonnad. "A comparison of direct and preconditioned iterative techniques for sparse, unsymmetric systems of linear equations". *International Journal for Numerical Methods in Engineering*, 28(4):801–815 (1989). doi:10.1002/nme.1620280406.

[10] N. Christ, M. Creutz, P. Mackenzie, J. Negele, C. Rebbi, S. Sharpe, R. Sugar, and W. W. III. "National Computational Infrastructure for Lattice Gauge Theory". Technical report (2005).

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Trans. Program. Lang. Syst.*, 13(4):451–490 (1991). doi: 10.1145/115372.115320.

[12] C. T. Davies, E. Follana, A. Gray, G. Lepage, Q. Mason, M. Nobes, J. Shigemitsu, H. Trottier, M. Wingate, C. Aubin, et al. "High-precision lattice QCD confronts experiment". *Physical Review Letters*, 92(2):022001 (2004).

[13] J. Dongarra, P. Koev, X. Li, J. Demmel, and H. van der Vorst. "10. Common Issues", chapter 10, 315–336. doi:10.1137/1.9780898719581.ch10.

[14] S. Duane, A. Kennedy, B. J. Pendleton, and D. Roweth. "Hybrid Monte Carlo". *Physics Letters B*, 195(2):216 – 222 (1987). doi:10.1016/0370-2693(87)91197-X.

[15] R. G. Edwards. "QDP++ Data Parallel Interface for QCD". SciDAC Software Coordinating Committee.

[16] A. Einstein. "Die Grundlagen der Allgemeinene Relativitätstheorie. (German) [The Foundations of the Theory of General Relativity]". 354(7):769–822 (1916). doi:10.1002/andp.19163540702.

[17] R. P. Feynman. "Space-Time Approach to Non-Relativistic Quantum Mechanics". *Rev. Mod. Phys.*, 20:367–387 (1948). doi:10.1103/RevModPhys.20.367.

[18] Z. Fodor and C. Hoelbling. "Light Hadron Masses from Lattice QCD". *Rev. Mod. Phys.*, 84:449 (2012). doi:10.1103/RevModPhys.84.449.

[19] M. P. Forum. "MPI: A Message-Passing Interface Standard". Technical report, Knoxville, TN, USA (1994).

[20] B. C. Hall. "An Elementary Introduction to Groups and Representations". *ArXiv Mathematical Physics e-prints* (2000).

[21] W. K. Hastings. "Monte Carlo sampling methods using Markov chains and their applications". *Biometrika*, 57(1):97–109 (1970). doi:10.1093/biomet/57.1.97.

[22] B. Joó, R. G. Edwards, and M. Peardon. "An anisotropic preconditioning for the Wilson fermion matrix on the lattice". *Computational Science & Discovery*, 3(1):015001 (2010).

[23] S. F. King. "Neutrino mass models". *Rept. Prog. Phys.*, 67:107–158 (2004). doi:10.1088/0034-4885/67/2/R01.

[24] F. Kjolstad. "Simit Language" (2015). `http://simit-lang.org`.

[25] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D. I. Levin, S. Sueda, D. Chen, E. Vouga, D. M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe. "Simit: A Language for Physical Simulation". Technical report (2015).

[26] F. B. Kjolstad and M. Snir. "Ghost cell pattern". In "Proceedings of the 2010 Workshop on Parallel Programming Patterns", 4 (ACM, 2010).

[27] M. D. Lam, E. E. Rothberg, and M. E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms". *SIGPLAN Not.*, 26(4):63–74 (1991). doi:10.1145/106973.106981.

[28] D. B. Leinweber, A. W. Thomas, and R. D. Young. "Physical Nucleon Properties from Lattice QCD". *Phys. Rev. Lett.*, 92:242002 (2004). doi:10.1103/PhysRevLett.92.242002.

[29] X. Luo and E. Gregory. "Non-Perturbative Methods and Lattice QCD" (2001).

[30] M. Luscher. "Advanced lattice QCD". In "Probing the standard model of particle interactions. Proceedings, Summer School in Theoretical Physics, NATO Advanced Study Institute, 68th session, Les Houches, France, July 28-September 5, 1997. Pt. 1, 2", 229–280 (1998).

[31] M. Luscher. "Computational Strategies in Lattice QCD". In "Modern perspectives in lattice QCD: Quantum field theory and high performance computing. Proceedings, International School, 93rd Session, Les Houches, France, August 3-28, 2009", 331–399 (2010).

[32] H. Matsufuru. "Introduction to lattice QCD simulations" (2007).

[33] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. "Equation of State Calculations by Fast Computing Machines". *The Journal of Chemical Physics*, 21(6):1087–1092 (1953). doi:10.1063/1.1699114.

[34] P. Micikevicius. "3D finite difference computation on GPUs using CUDA". In "Proceedings of 2nd workshop on general purpose processing on graphics processing units", 79–84 (ACM, 2009).

[35] K. Nakamura, K. Hagiwara, K. Hikasa, H. Murayama, M. Tanabashi, T. Watari, C. Amsler, M. Antonelli, D. M. Asner, H. Baer, H. R. Band, R. M. Barnett, T. Basaglia, E. Bergren, J. Beringer, G. Bernardi, W. Bertl, H. Bichsel, O. Biebel, E. Blucher, S. Blusk, R. N. Cahn, M. Carena, A. Ceccucci, D. Chakraborty, M. C. Chen, R. S. Chivukula, G. Cowan, O. Dahl, G. D'Ambrosio, T. Damour, D. de Florian, A. de Gouvea, T. DeGrand, G. Dissertori, B. Dobrescu, M. Doser, M. Drees, D. A. Edwards, S. Eidelman, J. Erler, V. V. Ezhela, W. Fetscher, B. D. Fields, B. Foster, T. K. Gaisser, L. Garren, H. J. Gerber, G. Gerbier, T. Gherghetta, C. F. Giudice, S. Golwala, M. Goodman, C. Grab, A. V. Gritsan, J. F. Grivaz, D. E. Groom, M. Grunewald, A. Gurtu, T. Gutsche, H. E.

Haber, C. Hagmann, K. G. Hayes, M. Heffner, B. Heltsley, J. J. Hernandez-Rey, A. Hoecker, J. Holder, J. Huston, J. D. Jackson, K. F. Johnson, T. Junk, A. Karle, D. Karlen, B. Kayser, D. Kirkby, S. R. Klein, C. Kolda, R. V. Kowalewski, B. Krusche, Y. V. Kuyanov, Y. Kwon, O. Lahav, P. Langacker, A. Liddle, Z. Ligeti, C. J. Lin, T. M. Liss, L. Littenberg, K. S. Lugovsky, S. B. Lugovsky, J. Lys, H. Mahlke, T. Mannel, A. V. Manohar, W. J. Marciano, A. D. Martin, A. Masoni, D. Milstead, R. Miquel, K. Moenig, M. Narain, P. Nason, S. Navas, P. Nevski, Y. Nir, K. A. Olive, L. Pape, C. Patrignani, J. A. Peacock, S. T. Petcov, A. Piepke, G. Punzi, A. Quadt, S. Raby, G. Raffelt, B. N. Ratcliff, P. Richardson, S. Roesler, S. Rolli, A. Romaniouk, L. J. Rosenberg, J. L. Rosner, C. T. Sachrajda, Y. Sakai, G. P. Salam, S. Sarkar, F. Sauli, O. Schneider, K. Scholberg, D. Scott, W. G. Seligman, M. H. Shaevitz, M. Silari, T. SjÃűstrand, J. G. Smith, G. F. Smoot, S. Spanier, H. Spieler, A. Stahl, T. Stanev, S. L. Stone, T. Sumiyoshi, M. J. Syphers, J. Terning, M. Titov, N. P. Tkachenko, N. A. Tornqvist, D. Tovey, T. G. Trippe, G. Valencia, K. van Bibber, G. Venanzoni, M. G. Vincter, P. Vogel, A. Vogt, W. Walkowiak, C. W. Walter, D. R. Ward, B. R. Webber, G. Weiglein, E. J. Weinberg, J. D. Wells, A. Wheeler, L. R. . Wiencke, C. G. Wohl, L. Wolfenstein, J. Womersley, C. L. Woody, R. L. Workman, A. Yamamoto, W. M. Yao, O. V. Zenin, J. Zhang, R. Y. Zhu, P. A. Zyla, G. Harper, V. S. Lugovsky, and P. Schaffner. "Review Of Particle Physics". 37(7A):1–1422 (2010).

[36] R. M. Neal. "MCMC using Hamiltonian dynamics". *ArXiv e-prints* (2012).

[37] S. U. Nicholas Metropolis. "The Monte Carlo Method". *Journal of the American Statistical Association*, 44(247):335–341 (1949).

[38] K. A. Olive et al. "Review of Particle Physics". *Chin. Phys.*, C38:090001 (2014). doi:10.1088/1674-1137/38/9/090001.

[39] F. Ortigosa, M. A. Polo, F. Rubio, J. Cela, R. de la Cruz, M. Hanzich, et al. "Evaluation of 3d rtm on hpc platforms". In "2008 SEG Annual Meeting", (Society of Exploration Geophysicists, 2008).

[40] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". *SIGPLAN Not.*, 48(6):519–530 (2013). doi:10.1145/2499370.2462176.

[41] B. Rodriguez, L. Hart, and T. Henderson. "Programming Regular Grid-Based Weather Simulation Models for Portable and Fast Execution". In "Proceedings of the International Conference on Parallel Processing", III–51 (CRC PRESS, 1995).

[42] Y. Saad. "Iterative Methods for Sparse Linear Systems" (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003), 2nd edition.

[43] H. Sagan. "Introduction to the Calculus of Variations" (Courier Corporation, 2012).

[44] K. Schulten. "Notes on Quantum Mechanics". *Department of Physics and Beckman Institute University of Illinois at Urbana, 2000, 390 ÑĄ* (2000).

[45] M. Schwartz. "Lecture I-7: Feynman Rules" (2012).

[46] R. Shankar. "Principles of Quantum Mechanics" (Springer US, 2012).

[47] B. Sheikholeslami and R. Wohlert. "Improved continuum limit lattice action for QCD with wilson fermions". *Nuclear Physics B*, 259(4):572 – 596 (1985). doi:10.1016/0550-3213(85)90002-1.

[48] J. R. Shewchuk. "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain". Technical report, Pittsburgh, PA, USA (1994).

[49] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, M. G. Duda, X.-Y. Huang, W. Wang, and J. G. Powers. "A Description of the Advanced Research WRF Version 3". Technical report, DTIC Document (2008). doi: 10.5065/D68S4MVH.

[50] F. Stancu. "Group theory in subnuclear physics" (Clarendon Press Oxford, 1996).

[51] G. Sterman, J. Smith, J. C. Collins, J. Whitmore, R. Brock, J. Huston, J. Pumplin, W.-K. Tung, H. Weerts, C.-P. Yuan, S. Kuhlmann, S. Mishra, J. G. Morfín, F. Olness, J. Owens, J. Qiu, and D. E. Soper. "Handbook of perturbative QCD". *Rev. Mod. Phys.*, 67:157–248 (1995). doi:10.1103/RevModPhys.67.157.

[52] R. Szeliski. "Bayesian modeling of uncertainty in low-level vision". *International Journal of Computer Vision*, 5(3):271–301 (1990).

[53] R. Teyssier. "Grid-Based Hydrodynamics in Astrophysical Fluid Flows". *Annual Review of Astronomy and Astrophysics*, 53:325–364 (2015).

[54] A. Ukawa. "Kenneth Wilson and lattice QCD". *J. Statist. Phys.*, 160:1081 (2015). doi:10.1007/s10955-015-1197-x.

[55] S. Weinberg. "The Quantum Theory of Fields". Number v. 1 in The Quantum Theory of Fields 3 Volume Hardback Set (Cambridge University Press, 1995).

[56] S. Weinberg. "The Quantum Theory of Fields". Number v. 2 in The Quantum Theory of Fields 3 Volume Hardback Set (Cambridge University Press, 1996).

[57] G. C. Wick. "Properties of Bethe-Salpeter Wave Functions". *Phys. Rev.*, 96:1124–1134 (1954). doi:10.1103/PhysRev.96.1124.

[58] K. G. Wilson. "Confinement of quarks". *Phys. Rev. D*, 10:2445–2459 (1974). doi:10.1103/PhysRevD.10.2445.

[59] L. Yang and M. Guo. "High-performance computing: paradigm and infrastructure". Wiley series on parallel and distributed computing (Wiley-Interscience, 2006).