View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Carolina Digital Repository

AUTOMATING DATA-LAYOUT DECISIONS IN DOMAIN-SPECIFIC LANGUAGES

Diptorup Deb

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2019

Approved by:

Robert J. Fowler

Allan Porterfield

Jan F. Prins

Donald E. Porter

Mary Hall

©2019
Diptorup Deb
ALL RIGHTS RESERVED

ABSTRACT

DIPTORUP DEB: Automating Data-Layout Decisions In Domain-Specific Languages
(Under the direction of Robert J. Fowler)

A long-standing challenge in High-Performance Computing (HPC) is the simultaneous achievement of programmer productivity and hardware computational efficiency. The challenge has been exacerbated by the onset of multi- and many-core CPUs and accelerators. Only a few expert programmers have been able to hand-code domain-specific data transformations and vectorization schemes needed to extract the best possible performance on such architectures. In this research, we examined the possibility of automating these methods by developing a Domain-Specific Language (DSL) framework. Our DSL approach extends C++14 by embedding into it a high-level data-parallel array language, and by using a domain-specific compiler to compile to hybrid-parallel code. We also implemented an array index-space transformation algebra within this high-level array language to manipulate array data-layouts and data-distributions. The compiler introduces a novel method for SIMD auto-vectorization based on array data-layouts. Our new auto-vectorization technique is shown to outperform the default auto-vectorization strategy by up to 40% for stencil computations. The compiler also automates distributed data movement with overlapping of local compute with remote data movement using polyhedral integer set analysis. Along with these main innovations, we developed a new technique using C++ template metaprogramming for developing embedded DSLs using C++. We also proposed a domain-specific compiler intermediate representation that simplifies data flow analysis of abstract DSL constructs.

We evaluated our framework by constructing a DSL for the HPC grand-challenge domain of lattice quantum chromodynamics. Our DSL yielded performance gains of up to twice the flop rate over existing production C code for selected kernels. This gain in performance was obtained while using less than one-tenth the lines of code. The performance of this DSL was also competitive with the best hand-optimized and hand-vectorized code, and is an order of magnitude better than existing production DSLs.

To my wife and my parents.

ACKNOWLEDGMENTS

My journey as a graduate student would not have been possible without the unwavering support and encouragement of a lot of people. These acknowledgments are a token of my appreciation to the people without whom this dissertation would not have been possible.

I thank Rob Fowler, my adviser, for teaching me how to be a researcher, and supporting me to the very end. I would not have reached this point without Rob believing in me, even more than I did at times. I would also thank Allan Porterfield for the endless hours he devoted to discussing and developing my ideas, and seed new ones. I am especially grateful that even after leaving Renci Allan continued visiting and was always available for discussions.

Thanks to Balint Joó at Thomas Jefferson National Accelerator Facility for teaching me enough about Lattice Quantum Chromodynamics (LQCD) to be able to build a domain-specific language for LQCD. Balint also helped me understand an advanced SIMD vectorization method, automating which ended up becoming a large part of my thesis. I am grateful to Robert Edwards and Frank Winter from Thomas Jefferson National Accelerator Facility and Dheeraj Kalamkar at Intel Parallel Labs for their research insights and ideas.

Jan Prins taught me parallel programming and high-performance computing. Don Porter helped me become better at presenting my research in words. Thanks also to Mary Hall for graciously agreeing to be on my committee, and providing insights into my work.

Thanks to Niki Fowler for painstakingly proofreading and editing my dissertation, and all my papers.

Thanks to Xipeng Shen at NCSU for providing a graduate-level compiler optimization course when no such course was offered at UNC.

Diane Pozefsky first planted the idea of staying back after my M.S. for a Ph.D. I am thankful that I took her advice and decided to take the qualifier examination. Victor Eijkhout at TACC guided me during a wonderful internship I did under him. The initial ideas around focusing on data-layouts and data-placement in HPC applications started germinating during my work under Victor.

I thank Anirban Mandal for his support and guidance all thought my work at Renci. I also acknowledge the wonderful friends I had during graduate school, including Duo Zhao, Sridutt Bhalachandra, Yue Gao, Fan Ziang, and Dheeraj Shetty.

I thank my parents for instilling the values in me that shape the person I am today. Thanks also to my sister, my uncle and my aunt for their constant encouragement and well wishes.

I thank my wife, Shreoshi, for encouraging me to go back to graduate school. Thanks to her unending love and support I could maintain focus and sanity during the bleakest of time.

I acknowledge support from the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of Nuclear Physics, Scientific Discovery through Advanced Computing (SciDAC) program under grants DE-FG02-11ER26050/DE-SC0006925, DE-SC0008706, and a sub-contract of grant DE-AC05-06OR23177 from the Thomas Jefferson National Accelerator Facility.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACT	Abstraction Characterization Template
AOS	Array-of-Structs
AOSOA	Array-of-Structs-of-Arrays
ATL	Array Transformation Language
C++ ET	C++ Expression Template
DSL	Domain-Specific Language
EDSL	Embedded Domain-Specific Language
FLOP	Floating-point Operations per Second
GPGPU	General-purpose Graphical Processing Unit
HPC	High-Performance Computing
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISL	Integer Set Library
JIT	Just-in-Time
LHS	Left-Hand Side
LQCD	Lattice Quantum Chromodynamics
METAL	Minimal Expression Template Array Language
MPI	Message Passing Interface
MSP	Multi-Staged Programming
QCD	Quantum Chromodynamics
QOPT	QUARC's Optimizer
QUARC	QCD's Array-based Rapid-prototyping Compiler
QUARC-RT	QUARC's Runtime
QUICQ	QUICQ Internally Calls QUARC
RHS	Right-Hand Side
SIMD	Single Instruction Multiple Data
SOA	Structs-of-Arrays
TMP	Template Metaprogramming

CHAPTER 1: INTRODUCTION

This dissertation targets a long-standing challenge in High-Performance Computing (HPC), *i.e.* simultaneous achievement of programmer productivity and hardware computational efficiency. Our approach is to develop a compiler for an Embedded Domain-Specific Language (EDSL) using a production compiler infrastructure (The LLVM Foundation, 2018). A unique innovation of this compiler is a framework for data-layout transformations to generate code competitive with the very best hand-optimizations.

The breakdown of Dennard scaling (Dennard *et al.*, 1974) in the early 2000's capped air-cooled uniprocessor clock speeds at around 3GHz and marked the onset of chip multiprocessing. Consequently, shifts occurred in HPC hardware architectures and in parallel-programming models. A single node of a current-generation HPC cluster has complementary features, such as deeply nested cache hierarchies, multiple cores, simultaneous multithreading, short-vector Single Instruction Multiple Data (SIMD) units, and accelerators. The software stack consists of frameworks supporting various programming models, such as message-passing, shared-memory threads, and accelerator-based data parallelism. Such diversity offers more choices to application developers, but it greatly complicates extraction of high levels of portable performance from modern HPC systems. Application programming on today's HPC systems requires programmers to mesh these disparate hardware and software components to boost performance. In addition, significant architecture-aware programming skills and domain-specific knowledge are both necessary. The deepening programming crisis calls for newer approaches in programming language design and in compiler technology. Development of such newer approaches is an increasingly important goal for the HPC research community (Department of Defense, 2001), especially given the current rush towards exascale computing.

Legacy applications in C/C++ or FORTRAN are unable to utilize the peak machine Floating-point Operations per Seconds (sFLOPs) rate and the maximum available memory bandwidth without architecture-specific code-tuning. Certain types of newer architectures such as NVIDIA's General-purpose Graphical Processing Units (sGPGPUs) require large-scale re-implementation and rewrite of

legacy applications. The issue of code-tuning and rewriting also applies to non-legacy applications. Converting a prototype of an application to a production-grade HPC version generally involves an order of magnitude of growth in the lines of code. Thus, the net result is a high entry barrier to HPC programming that limits true accessibility of HPC platforms to a small niche of experts.

Performance portability is a set of closely related problems linked to the overall problem of code rewrite and re-implementation. For the very best performance, performance-tuning of an HPC application kernel by hand often requires architecture-specific optimizations. The specificity of such optimizations necessitate repeating the exercise on different architectures or generations of an architecture family. A notable example is hand-vectorization of code for different generations of x86 processors. Each generation of x86 has a different Instruction Set Architecture (ISA) extension to support SIMD vectorization that requires a rewrite of hand-vectorized codes before the code can make effective use of a newer ISA vector extension. Another aspect of performance portability is that hand optimizations are not directly portable across applications, or even across kernels within an application stack. Solving the issue of performance portability requires addressing all these related issues. A solution to the issue of performance portability is still missing in existing programming languages and frameworks. Often, the only recourse to high performance on different architectures is to write and maintain different versions of the application. The resulting code complexity means that many HPC applications bear little resemblance to the original abstract algorithms and are difficult, if not impossible, to understand for those domain experts who are not expert programmers.

Domain-specific Languages (DSLs) offer a potential solution to the programming crisis in HPC. The history of DSLs is traceable to the very first commercially available language, FORTRAN (Backus, 1978). Although not called a DSL, FORTRAN was conceived as a high-level language specifically for scientific and mathematical computation. Since then, several DSLs such as Matlab (MATLAB, 2010), R (The R Foundation, 2018), NumPy (van der Walt *et al.*, 2011) have gained popularity in HPC. Code written in such DSLs is not always the most performant, and often the computationally intensive tasks require implementation within libraries written in C/C++ or FORTRAN. The strategy works if a highly tuned library implementation is available, but this is not always the case. The past two decades have seen much effort towards improving the performance of DSLs. One possible solution is to create DSLs embedded inside a high-level language. Such DSLs are called EDSLs. EDSLs and the closely related concept of active libraries (Veldhuizen and Gannon, 1998) rely on generative programming features of an

existing high-level language to optimize, tune, and generate domain-specific code for a DSL embedded within that high-level language. C++ template metaprogramming is perhaps one of the most successfully adopted techniques for building EDSLs. Apart from C++ templates, other recent approaches have used Multi-Staged Programming (MSP) (Taha and Sheard, 1997) to build EDSLs on top of languages such as Scala (Odersky *et al.*, 2008) and Lua (LabLua, 2015). Chapter 2 explains these methods in detail.

EDSL designs, especially those based on purely generative methods, are not perfect. A significant limitation is the lack of important types of optimizations that require deeper compiler analysis than is possible inside the EDSL-layer. Analyses range from data-flow-based redundancy elimination optimization to much more intricate data-dependence-based loop optimizations. Code generated by some EDSL can also introduce obfuscations that impede low-level compiler optimizations. These obfuscations, such as address and loop linearization, can make it very hard, if not impossible, for any compiler to infer the programmer's intent, inhibiting subsequent compiler-driven analysis and optimization.

Overcoming the limitations of EDSLs requires a new design approach. Such a design should extend the underlying host-language's compiler be aware of EDSL domain-specific abstractions and constructs. With such a design the possibility exists of optimizing EDSL expressions using a wider range of standard compiler optimizations. A close coupling between the EDSL and the host-language compiler simplifies designing newer domain-specific optimization and code-generation techniques by reusing existing compiler infrastructure. Chapters 5 and 6 present such a new EDSL-design approach.

Data-placement is another area that is increasingly important for both general-purpose parallel programming languages and EDSLs. The overall data-placement of an application controls several aspects of data movement across the computation domain. In the context of HPC, data movement refers to communication over a network interconnect, across the non-uniform memory-access shared-memory hierarchy, or even among registers. Several complementary methods exist to optimize applications for these scenarios. Such methods include customizing shared-memory data-layouts, and customizing data-distributions for distributed-memory clusters. Automating data-placement *via* compiler analysis, and even providing an interface to define data-placement, remain key challenges for programming systems. Optimizing data-placement by hand is one of the main areas that requires expert programming skills in HPC.

Alleviating the programming crisis by addressing limitations in EDSL code-generation and designing an easy-to-use data-placement abstraction technique form the core of this dissertation. We introduce

a new technique for EDSL code-generation that uses C++ template metaprogramming to encode a domain-specific Intermediate Representation (IR) inside a general-purpose compiler IR. The new metaprogramming technique simplifies the engineering of new domain-specific optimizations and code transformations on top of existing general-purpose compiler technology. We also introduce a data-placement abstraction technique that completely decouples architecture-specific data-distributions and data-layouts from application-level algorithms. Using these two innovations, we introduce a new SIMD auto-vectorization based on a data-layout transformation technique. For selected kernels form a large scale HPC application, the new auto-vectorization offered a factor of two performance improvement, while taking less than one-tenth the lines of code. Other innovations explained in Section 1.3 include automated code-generation from the same abstract high-level program for both multi-core and multi-node cluster, and a design for high-level optimization of EDSL constructs.

1.1 A Historical Perspective

Kennedy *et al*. (Kennedy *et al.*, 2004) quantified computational productivity as a function of the amortized cost of preparing the program, the cost of running it, and the net present value of the results. The computational resources needed by some problems are minimal and the time to solution is not large. For such problems, it is ideal to minimize the programming costs by using an expressive scripting tool such as Python, Matlab, or R. At the other end of the spectrum, large computational campaigns may use many millions of CPU hours and the opportunity cost of waiting for an answer may be very large. In extreme cases, *e.g.*, disaster forecasting or even responses to interactive queries, late answers may be useless. Such cases require uncompromised computational efficiency. There are even cases in which there is a need to program a new algorithm quickly for a large time-critical computation. Most large HPC applications lie somewhere between these extremes.

In 1978, Glenford Myers (Myers, 1979) first published a book that focused on the "semantic gap", the incompatibility between the abstractions of high-level languages and the low-level machine instructions of the computers of that era. Myers advocated closing the gap by raising the semantic level of the hardware. Instead, at about the same time, the RISC revolution increased the gap by further lowering the semantics of the machines, thus leaving a larger gap that needed to be bridged by software. Subsequent complexities, such as multi-issue CPUs with deep pipelines, vector pipelines on commodity processor

chips, multi-core/multi-threaded chips, and deeper, more complex coherent memory hierarchies, have exacerbated the problem.

Myers and Kennedy *et al*. addressed the same problem from different perspectives. Closing the semantic or productivity gap entails increasing the expressive power of the programming environment while simultaneously increasing the computational efficiency of programs.

Striking the right balance between expressiveness and computational efficiency has been a challenge since the first generation of computers. To incorporate expressiveness, programming systems have added layers of abstractions. In recent decades, innovations such as object-oriented and functional programming have greatly improved the tools available for abstraction. Unfortunately, multiple layers of abstraction and the resulting deep call chains, sometimes involving dynamic bindings, may decrease computational efficiency as measured in terms of the peak achievable architectural FLOPs rate. Generic and generative programming techniques, *e.g.*, C++ template metaprogramming (Eisenecker, 1997; Bischof *et al.*, 2004) and Template Haskell (Sheard and Peyton Jones, 2002) macros, mitigate aspects of this problem. MSP (Taha and Sheard, 1997), or staging, also has shown promise. From a code-generation perspective, Just-in-Time (JIT) compilation has been by far the most popular option. Many expressive high-level languages that support type introspection use JIT to enhance computational efficiency. Full-blown parallel programming languages such as the ones developed as part of United States Department of Defense's Defense Advanced Research Projects Agency's (DARPA) High Productivity Computing System (HPCS) project (Dongarra *et al.*, 2008) are another alternative. These languages focus primarily on abstracting architecture-specific parallelization, making it easier to write parallel programs.

1.2 Thesis

Existing EDSLs and other high-productivity programming systems for HPC look to combine high productivity with high performance *via* abstract parallel patterns. Such patterns typically are specialized for multiple architectures and provide some level of performance portability. However, no EDSLs or high-productivity programming system has completely addressed the issue of abstractions for data-placement. Programming systems that support such abstractions often do so for only a single architectural layer. Many systems either support a set of abstractions for data-layouts and ignore data-distributions, or the *vice versa*. Failing to address both aspects of data-placement leads to performance loss and limits the

applicability of the programming system. Close coupling of abstractions within the main programming interface is another issue, which leads to source code that has data-placement logic intertwined with application logic. Such entanglement impairs readability and overall portability of the code.

To mitigate these issues, our thesis is as follows.

A system of high-level data-placement abstraction based on a well-defined algebra can describe multiple layers of data-placement in relation to each other. Combining the data-placement mechanism with domain-specific code optimization and generation within a compiler can improve programmer productivity without loss in computational efficiency, reducing the semantic gap.

1.3 Contributions

In support of the thesis statement presented in Section 1.2 our dissertation makes the following main contributions.

- We implemented a C++-14 and LLVM-based EDSL compilation framework called QCD's Array-based Rapid-prototyping Compiler (QUARC) that serves as a proof-by-example of our thesis.
  - To demonstrate QUARC's capability, we implemented a prototype EDSL for the HPC domain of Lattice Quantum Chromodynamics (LQCD). The performance of selected kernels implemented in this EDSL, QUICQ Internally Calls QUARC (QUICQ), was up to two times better than that of an existing production application called MILC (MILC collaboration, 1992). The kernels implemented in QUICQ also took less than one-tenth the lines of code when compared to MILC. QUICQ's performance was up to 10 times better than that of an existing production DSL in LQCD, QDP++ (Edwards and Joó, 2005). QUICQ also was competitive with the best hand-optimized kernels for the evaluated set of examples.
- We implemented an array index-space transformation algebra to define data-placement abstractions. Using the algebra, it is possible to define abstractions both for data-layouts and for data-distributions.
  - This design separates data-placement specifications from the rest of the programming layer. Application developers can customize data-placement specifications at runtime to tune program execution.

  - The same high-level code executes in parallel on a single multi-core server or on a multi-node cluster by changing runtime data-placement specifications.

- We introduce an external policy-driven speculative SIMD auto-vectorizer.

  - Speculative SIMD vectorization frees application programmers from having to make data-layout choices in their application code. External agents, such as an autotuner, or a low-level expert, can make the layout selection at runtime.
  - We introduce a data-layout selection policy for higher-order stencil kernels. A stencil kernel is an iterative computation used in various scientific computations such as partial differential operators and image-processing. The term "stencil" refers to updating an array element according to a fixed computational pattern involving neighboring array elements in the same or in a separate array.

- We introduce a new technique for building C++-based EDSLs.

  - The new technique, Abstraction Characterization Templates (sACTs), uses metaprogramming to generate a domain-specific IR from standard C++ templates.
  - We demonstrate the efficacy of delayed or late *scalarization* of array expressions. In QUARC, scalarization does not happen inside C++ templates, and is done as late as possible in the code-generation process. This facilitates improved compiler analysis and optimization.
  - We introduce a design for using scalar data-flow optimizations to optimize EDSL constructs.

1.4 Dissertation Overview

This rest of the dissertation provides the motivation, design considerations, implementation details, and evaluation of the QUARC framework as follows:

Chapter 2 provides a background on different EDSL designs. The emphasis is on EDSL code-generation methods and on data-placement abstractions provided by stat- of-the-art HPC EDSLs. The end of the chapter includes a motivating example showing the impact of data-layout transformations on SIMD vectorization performance on a modern x86 architecture.

Chapter 3 introduces the QUARC framework and its individual components.

Chapter 4 presents a formalization of the array index-space transformation algebra. The chapter discusses these operators in general, without focusing on any aspect of their use in QUARC. Readers with experience using APL or other array programming language already should be familiar with most of the concepts. They can refer to Section 4.1.2 for the specific differences between our operators and the more general APL counterparts.

Chapter 5 presents QUARC's programming interface (METAL), describing the C++-14-based data-parallel array operators and the overall API to construct EDSLs. Section 5.1 includes a detailed guide to METAL's syntax, and is aimed at programmers. General audiences may skip this chapter. Section 5.2 describes the ACT's design pattern and its use for preserving high-level semantics inside METAL. Section 5.3 describes QUARC's data-placement abstractions (ATL).

Chapter 6 describes in detail our LLVM-based domain-specific compiler's design and implementation. It also describes a small runtime library interface for automating Message Passing Interface (MPI) communication-generation. The chapter explains each stage of the code-generation and optimization pipeline, including the late scalarization approach, in detail.

Chapter 7 presents an empirical performance analysis of the impact of different data-layout choices on the SIMD vectorization performance of stencil computations. The study was conducted for multiple generations of Intel x86-based architectures. We discuss the data-layout characteristics that drive SIMD vectorization performance and the trade-offs required to ensure high performance.

Chapter 8 presents a data-layout selection policy based on the empirical study in Chapter 7. QUARC uses the policy in its speculative SIMD vectorization technique.

Chapter 9 presents the steps involved in developing an EDSL for LQCD. Section 9.1 introduces the domain of LQCD, and then Section 9.2 shows the main excerpts of our EDSL implementation. The goal of this chapter is to evaluate the overall productivity gained in terms of lines of code by using this EDSL, when compared to an existing production application developed in C.

Chapter 10 evaluates the performance of our LQCD DSL by comparing it both to an existing C++-based production DSL and to a legacy application written in C. Wherever available, both sets of evaluation include the very best hand-optimized implementations as the standard of peak performance.

Chapter 11 surveys further related work in various areas of relevance to QUARC.

Chapter 12 concludes this dissertation and lays out a vision for future extensions.

CHAPTER 2: BACKGROUND AND MOTIVATION

Designing a DSL for an HPC domain involves several important design considerations. This chapter reviews the choices, emphasizing code-generation methods and data-placement abstractions for domain-specific abstractions.

2.1 EDSL Code-Generation

Contemporary HPC DSLs fall into two broad categories: standalone DSLs, and EDSLs that are embedded inside a high-level host language. Examples of standalone DSLs are the numerical analysis DSLs Matlab (MATLAB, 2010), Julia (Bezanson *et al.*, 2017), the machine-learning DSL Glow (Rotem *et al.*, 2018). Prominent EDSLs are Halide (Ragan-Kelley *et al.*, 2013) an image-processing DSL, SPIRAL (Püschel *et al.*, 2005) a DSL for signal processing, and QDP++ (Edwards and Joó, 2005), a DSL for LQCD. Our present discussion defines EDSLs as those languages whose syntax does not differ from the host languages, *i.e.*, every EDSL program is a completely legal host language program. Such characterization of an EDSL excludes language extensions such as CUDA C (NVIDIA Corporation, 2010), OpenMP (OpenMP Architecture Review Board, 2015), OpenCL (Stone *et al.*, 2010) and OpenACC (OpenACC.org, 2013). All these language extensions introduce new keywords, data types, and annotations. All require custom parsers and compilers.

Standalone DSLs require more work to develop and to maintain the language front-end and parser, but the interface can be tailored more closely to domain-specific requirements. EDSLs lower the front-end implementation cost, but the interface is restricted to the features of the underlying high-level language. Standalone DSLs provide their own compilers with high-level, domain-specific optimizations and code-generation. Usually, low-level optimizations and machine-code-generation are offloaded to a general-purpose compiler such as LLVM (The LLVM Foundation, 2018). However, the high engineering cost of developing standalone DSLs disqualify this option for many HPC communities. EDSLs offer

quicker design and development turnaround times, and thus present a more viable option. This chapter focuses exclusively on EDSL development methodologies.

There are several methodologies to construct EDSLs. Each methodology uses a different approach to detect EDSL code sections embedded inside a host language program and to generate domain-specific code for those EDSL sections. Metaprogramming within the host language, through custom source-to-source translation, and utilizing staging are some of the most successful EDSL methodologies. The following paragraphs review these methodologies.

Metaprogramming is a programming technique in which programs treat themselves or other programs as data. Using metaprogramming, a program generates code during compilation of itself, and merges the generated code with the rest of its source code. Lisp and its dialects were the earliest exponents of metaprogramming. Lisp (LISt Processor) treats source programs as linked lists of data structures. Using Lisp's macro system, programmers can *introspect* their programs as a list of data structures. Related macro systems have evolved in other programming languages as well, *e.g.*, Template Haskell (Sheard and Peyton Jones, 2002), Scala (Odersky *et al.*, 2008), and Clojure (Rich Hickey, 2007).

Generative programming (GP) is another term closely associated with metaprogramming. Czarnecki and Eisenecker (Czarnecki and Eisenecker, 2000) described GP as an attempt to automate creation of software components by developing programs that synthesize other programs. Template Metaprogramming (TMP) falls into this category. Templates are program constructs that are written without binding them to specific data types. The data types are specified later as a specialization of the template. TMP was pioneered by ML, a language derived from Lisp. Since templates are written with generic types rather than with concrete types, TMP is often also called *generic programming*. The most popular use of TMP is found in C++. C++ TMP is widely used in C++'s standard library. C++ TMP-based methods such as C++ Expression Templates (sC++ ETs) (Veldhuizen, 1995; Vandevoorde and Josuttis, 2002) are also widely used to create EDSLs. C++ ETs use TMP to build *expression objects* that abstract complicated loop structures. Template expansion and specialization are used to synthesize loops for different data types and architectures without direct programmer involvement. C++ ETs has been used to develop HPC DSLs (Edwards and Joó, 2005; Parsons and Quinlan, 1994; Reynders and Cummings, 1998), where C++ ETs automate MPI, OpenMP, or CUDA code-generation.

C++ TMP, and specifically C++ ETs, suffer from several shortcomings. The loop synthesis happens in the template layer and cannot incorporate various advanced loop optimizations. Loop optimizations

that require data dependence-based analysis are hard, if not impossible, to incorporate using C++ TMP. C++ TMP also cannot apply high-level optimizations, such as expression fusion and other redundancy elimination, across multiple template statements. C++ TMP code-generation is limited to a single template recursion chain. For these reasons, C++ TMP may be used to synthesize parallel loops, but the performance often falls short of the level obtained by the very best hand-tuned libraries. Additionally, obfuscations introduced by C++ TMP can also impede subsequent compiler analysis and optimization.

Multistage programming (MSP) (Taha and Sheard, 1997), or staging, is a special case of metaprogramming that involves staging of portions of code for evaluation and compilation at a later phase. Phased, or partial, evaluation is beneficial for scenarios in which information about a program only becomes available at a later phase. A good example is type information in dynamically typed languages. MSP can have both compile-time and runtime stages.

MSP is the basis for recent HPC DSL frameworks like Delite (Sujeeth *et al.*, 2014) and Terra (DeVito *et al.*, 2013). Delite is an EDSL on top of Scala (Odersky *et al.*, 2008), a hybrid object-oriented functional language that runs on the Java Virtual Machine (JVM). Delite uses a modified Scala compiler and MSP to generate a domain-specific IR embedded into Scala's byte code. Delite's compiler optimizes this domain-specific IR, before generating architecture-specific parallel code. Terra is a low-level language specifically designed for MSP in Lua (LabLua, 2015), a dynamically typed functional programming language. Terra's design envisions two-level staging. At first, a program is implemented in Lua for rapid prototyping, and then performance-critical sections are staged as Terra code. The Terra compiler uses dynamic staging to enable runtime feedback-driven optimizations and auto-tuning. Chapter 11 expands our discussion of Delite and Terra.

MSP requires an up-front decision about the portions of a program that are to be staged. MSP-based systems use either explicit staging annotations or custom data types to denote staged code. The staged portion of the code can be adapted to different platforms and architectures by adding new library routines or by adding new compilation targets. The high-level staged code does not need changes. However, the up-front demarcation of staged code limits the interaction between the staged code and the non-staged code. The limited interaction between the staged and the non-staged code is disadvantageous in scenarios in which optimizing the staged code requires knowing the context of the staged code inside a larger non-staged code section. Maintaining type-safety is another issue associated with MSP. An MSP compiler translates high-level staged code into a domain-specific IR, and after optimizations translates

the domain-specific IR back into the regular host language IR or directly to low-level code. It is not trivial to ensure guaranteed type-safety during these code-generation steps, especially when MSP is used inside a dynamically typed language. Addressing the type-safety issue was one of the major emphases of the Terra framework. Terra required static type information to be added to every staged library function call within the dynamically typed Lua language. Beyond maintaining type-safety, extending the domain-specific IR of an MSP compiler requires significant engineering work. Often, that engineering work involves rewriting a lot of the same boilerplate. The recent Forge framework (Sujeeth *et al.*, 2013) targets this issue within the Delite framework.

Split-languages can also fall under the purview of metaprogramming, but we choose to categorize them separately to highlight the use of two separate programming interfaces in split-languages. The two programming interfaces of a split-language separate domain-specific algorithms from architecture-specific code-generation decisions. The domain-specific programming interface is embedded into a host language, and a separate specification language drives architecture-specific code-generation. Notable examples of EDSLs using a split-language design are Halide (Ragan-Kelley *et al.*, 2013), SPIRAL (Püschel *et al.*, 2005), and Sequoia (Fatahalian *et al.*, 2006). All three languages are programmed in C++/C, but use standalone specification languages for code-generation decisions. The standalone code-generation specification interface is a main feature of these EDSLs. Using the separate interface, a set of domain experts can rapidly prototype the algorithmic portions of an application. After that another set of experts can tune the application by building an architecture-specific code-generation specification.

Despite their elegance, split-languages have potential pitfalls. Often, the code-generation specification is too intricate for the average domain expert, and a separate set of experts to write the code-generation specifications are not available. Even when experts to write code-generation specifications are available, the best specification may require exhaustive searching of a large optimization space. Thus, the challenge lies in splitting the two programming layers without over-complicating the code-generation specification layer. The authors of Halide acknowledge the problem in their recent follow-up paper (Ragan-Kelley *et al.*, 2017). While the core Halide language is highly expressive and easy to learn for domain experts, the programming schedules remains hard for most programmers.

JIT compilation is another common EDSL code-generation strategy. JIT refers to the fact that low-level code-generation happens during program execution. Terra supports JIT compilation; there have also been implementations (Winter *et al.*, 2014) of JIT compilers within C++ ETs. Source-to-source

translation is a technique in which the DSL code is translated into a low-level language like C/C++ or FORTRAN. The translated code then is compiled using a standard compiler.

2.2 Data Placement Abstractions

Data placement refers to the organization of a program's data across a memory domain. For modern HPC architectures, this can occur at multiple levels. Data placement across multiple nodes of a cluster is called data-distribution or data partitioning. The organization of the members of a data structure inside a shared-memory domain is called memory data-layout. The various levels of data-placement have to perform well in conjunction with each other. Together, they control data-movement-related costs and significantly impact a program's overall performance.

The importance of data placement is closely tied to the *locality of reference*. Denning and Schwartz (Denning and Schwartz, 1972) first observed that programs repeatedly access same or related storage locations, and called the locality of reference. A significant number of optimizations, both in software and hardware, are designed to exploit locality of reference. Such features include deeply nested multiple levels of caches, branch predictors, and prefetchers. Operating systems tailor their virtual memory sub-system and paging policies to depend on locality of reference. Important compiler optimizations such as loop tiling, loop fusion, and software prefetching work with the hardware layers to exploit locality of reference.

Several programming languages have explicit data-placement options using abstractions both for data-distribution and for data-layout. We next survey examples of both types of abstraction.

Data-Distribution Abstractions

Data-parallel programming languages, most notably High Performance Fortran (HPF) (Loveman, 1993) and ZPL (Chamberlain *et al.*, 1998), focused on abstractions to define data-distribution across multiple nodes of a cluster computer. Partitioned Global Address Space (PGAS) languages first developed in the late 1990's took a different approach to data-distribution. PGAS languages allowed programming with a seemingly shared-memory model, with explicit demarcation between local and shared data. All shared data was partitioned over processors. The initial PGAS languages, Unified Parallel C (UPC) (El-Ghazawi and Smith, 2006), Co-array FORTRAN (Numrich and Reid, 1998), and Titanium (Yelick *et al.*,

1998), all began as extensions to existing sequential languages. The initial implementation did not offer much support for controlling data-distributions. Later enhancements added abstractions that provided support for data-distributions, such as *blocking* of arrays. Closely related to the PGAS languages are those developed as part of DARPA's High-Productivity Computing System (HPCS) project (Dongarra *et al.*, 2008). The three HPCS languages, X10 (Charles *et al.*, 2005), Fortress (Allen *et al.*, 2008), and Chapel (Chamberlain *et al.*, 2007), offered different levels of support for data-distributions. Fortress allows libraries to define custom data-distributions for arrays. X10 supports distributed arrays that can be blocked across multiple processes. Chapel offers the most flexibility for array-data distributions. It supports a set of data-distribution choices, but also allows defining application-specific custom data-distributions.

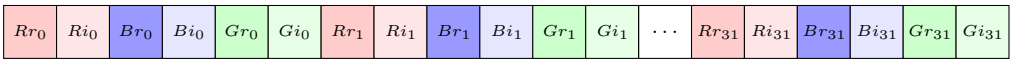
A common limitation of most of these languages is the close coupling of data-placement abstractions with the core language semantics. The close coupling means that data-placement abstractions are inherently intertwined with application-level algorithms. Experimenting is hard with different distribution options without first altering the source code. The code entanglement also limits portability, and restricts evolving an old code to a newer architecture.

A more modern framework, Legion (Bauer *et al.*, 2012), simplifies the problem of separating data placements from the rest of the application logic. Legion's programming model is built around the notion of *logical regions*. A logical region is a logical data partition, and the smallest unit for data-distribution in Legion's programming model. Every logical region defines its access privileges, aliasing, and coherence properties. The properties of a logical region decide the level of concurrent accesses possible on the logical region. Legion programs are composed of *tasks*, each of which accesses logical regions. By performing a task-dependence analysis, Legion's runtime system schedules parallel execution of tasks. Tasks can execute on different nodes of a cluster, a node-attached accelerator, or multiple cores of a node. Programmers retain control of how the data is partitioned and of the mapping of the data partitions to processors. The mapping interface is decoupled from the rest of the application programming interface. Therefore, data-placement logic is separated from the main application logic, and provides flexibility in extending an existing application to a new architecture.

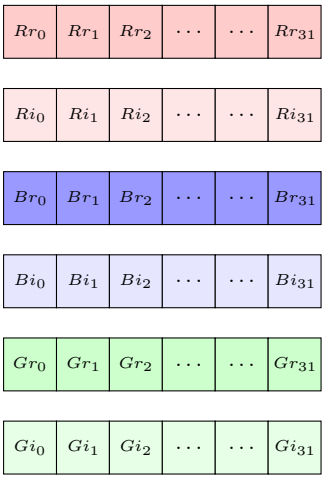
Data-Layout Abstractions

Data-layouts define the in-memory organization of members of a data structure in shared-memory architectures. The data-layout of an application's data structures plays an important role in the appli-

AoS



SoA



AoSoA

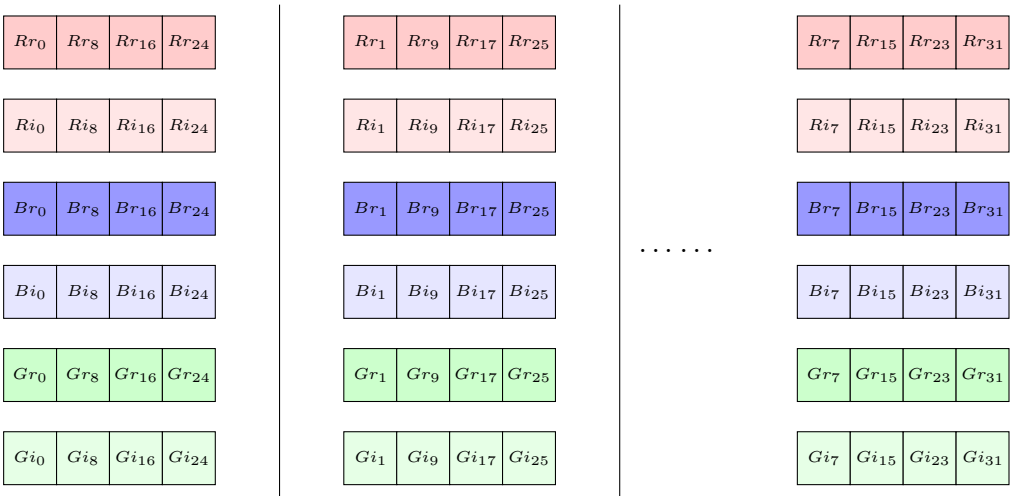


Figure 2.1: This illustrative example shows three different data-layouts for a one-dimensional array of size 32. Each element of this array has six nested elements $\{R_r, R_i, B_r, B_i, G_r, G_i\}$. The data type represents an $SU(3)$ complex vector. Chapter 9 provides further details on this data type and its use. The figure shows an AOS, SOA, and an AOSOA data-layout for the array. Each data-layout leads to a rearrangement both of the array elements and of nested elements at each index position.

cation's performance, and for this reason there has been significant amount of research to optimize data-layouts. Anderson *et al*. (Anderson *et al.*, 1995) used data-layout transformation to address false sharing in cache-coherent shared-memory multiprocessors. Lu *et al*. (Lu *et al.*, 2009) used an approach like Anderson *et al*., but for improving the locality on a prototype non-uniform cache-access architecture. Barua *et al*. (Barua *et al.*, 1999) and So *et al*. (So *et al.*, 2004) proposed data-layout transformations to improve memory-level parallelism on FPGAs. Both methods transform the data-layout of an array to interleave elements across multiple on-chip memory banks. Sung (Sung *et al.*, 2010) targeted structured grid applications on many-core and GPGPU platforms. Henretty (Henretty *et al.*, 2011) looked at data-layout transformations to improve short-vector SIMD vectorization.

Programming systems for modern architectures are increasingly exploring data-layout options to improve application performance. Figure 2.1 illustrates three data-layout options, AOS, SOA, and Array-of-Structs-of-Arrays (AOSOA), for a one-dimensional array. Each choice has its advantage, the SOA data-layout is generally better for SIMD vectorization, the AOS data-layout has better spatial locality, especially if neighboring array elements are accessed together, the AOSOA data-layout strikes a balance between AOS and SOA. The performance of a data-layout depends on various factors, including array size, data access patterns, and hardware architectural properties. Several recent performance studies (Rosales *et al.*, 2016; Giles *et al.*, 2013) have argued for the inclusion of these type of data-layout options *via* either library-based abstractions or language extensions. Still, these layout choices have limitations, such as they do not allow arbitrary splitting and transposition of array dimensions. For higher-dimensional arrays, the lack of the feature to arbitrarily split and transpose array dimensions limits the number of data-layout choices. Such layout choices are required for some kernels and some architectures. Section 2.3 presents such a scenario.

Despite the need for better support for data-layouts, very few extant programming systems support data-layout abstractions or constructs. Most proposals have not grown beyond the research phase, including Intel's ispc compiler (Pharr and Mark, 2012) and a proposed extension to the OpenACC standard (Hoshino *et al.*, 2014). Both systems included a fixed set of layout choices as language extensions and keywords. Terra (DeVito *et al.*, 2013), Halide (Ragan-Kelley *et al.*, 2013), and some research prototype compiler systems (Majeti *et al.*, 2014; Xu and Gregg, 2014) support Array-of-Structs (AOS) and Structs-of-Arrays (SOA) data-layouts. The Kokkos C++ library (Carter Edwards *et al.*,

2014) supports custom data-layouts other than Array-of-Structs (AOS) and Structs-of-Arrays (SOA) for multi-dimensional arrays.

Conclusion

Our work looks at a general system of data-placement abstractions that is usable for defining both data-layouts and data-distributions. Chapter 4 presents the array index-space transformation algebra that is the basis of our data-placement abstractions. The implementation of the abstractions is in Section 5.3, and code-generation based on the abstractions is described in detail in Chapter 6.

2.3 Motivating Example

This section presents a motivating example that highlights the need for data-layout transformations to improve the SIMD vectorization performance of stencil computation on an Intel Haswell server. Although this example uses a single server, the observations apply to other recent x86-based server architectures.

Stencil computations are one of the most important computational patterns in scientific computation and HPC (Asanovic *et al.*, 2009). Stencils are iterative computations that update each array element according to a fixed computational pattern involving its neighboring array elements. Listing 2.1 presents a nine-point scalar stencil computation over a four-dimensional regular grid. The loop-nest shown in the example is perfectly nested and completely parallel, and is an ideal candidate for fine-grained inner-loop parallelization or vectorization.

Auto-vectorization

Compiling Listing 2.1 with Intel's ICC 17.04 compiler at a -O3 optimization level leads to auto-vectorization of the innermost loop. Listing 2.2 shows the generated assembly code for the main loop body. The code-generation target was an Intel Haswell architecture with 256-bit AVX2 registers. With single-precision floating-point values and a small problem size of $16\times32\times32\times32$, the total data footprint is only 4MB. The test machine for this example had 20MB of L3 cache. Therefore, the whole problem size easily fits inside this L3 cache. For this small problem, and based solely on wall clock execution time, the default auto-vectorization yields a 53% performance improvement compared to scalar execution with the "no-vec" compiler option.

```
constexpr std::size_t T = 32, Z = 32, Y = 32, X = 32;

void stencil_9pt (float * restrict A1, const float * restrict A2) {
  for(auto t = 1ul; t < T-1; ++t)
    for(auto z = 1ul; z < Z-1; ++z)
      for(auto y = 1ul; y < Y-1; ++y)
        for(auto x = 1ul; x < X-1; ++x) {
          A1[t*Z*Y*X + z*Y*X + y*X + x]
            = A2[(t-1)*Z*Y*X + z*Y*X     + y*X     + x]
            + A2[(t+1)*Z*Y*X + z*Y*X     + y*X     + x]
            + A2[t*Z*Y*X     + (z-1)*Y*X + y*X     + x]
            + A2[t*Z*Y*X     + (z+1)*Y*X + y*X     + x]
            + A2[t*Z*Y*X     + z*Y*X     + (y-1)*X + x]
            + A2[t*Z*Y*X     + z*Y*X     + (y+1)*X + x]
            + A2[t*Z*Y*X     + z*Y*X     + y*X     + x - 1]
            + A2[t*Z*Y*X     + z*Y*X     + y*X     + x + 1];
        }
  //... Elided boundary region computations
}
```

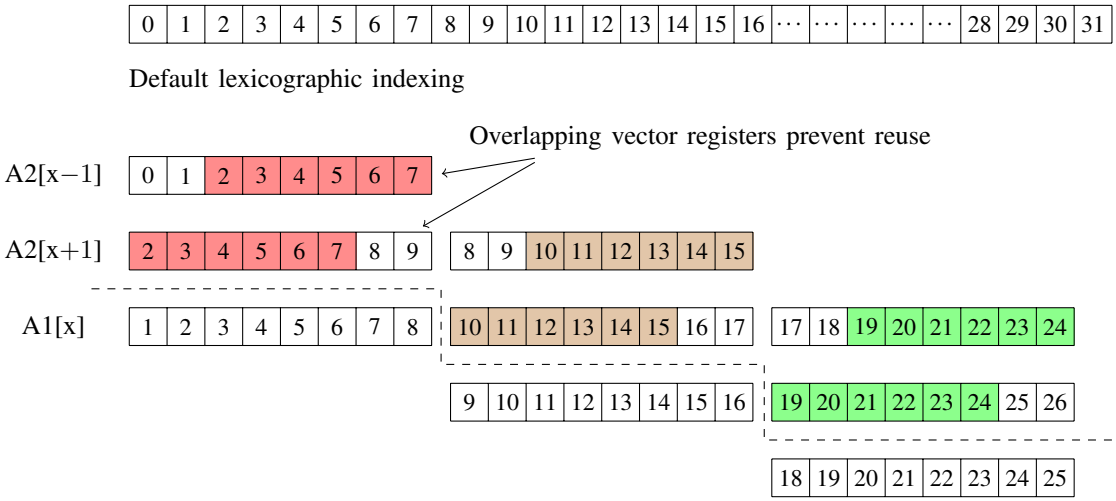
Listing (2.1) A nine-point scalar stencil

```
# shape of arrays A1,A2 is {T-Z-Y-X}. X is the fastest changing dimensions.
vmovups 4228+A2(...),   %ymm0         # (t-1) from 0*Z*Y*X + 1*Y*X + 1*Y + 1*X
vmovups 131204+A2(...), %ymm1         # (z-1) from 1*Z*Y*X + 0*Y*X + 1*Y + 1*X
vmovups 135172+A2(...), %ymm4         # (y-1) from 1*Z*Y*X + 1*Y*X + 0*Y + 1*X
vmovups 135296+A2(...), %ymm5         # (x-1) from 1*Z*Y*X + 1*Y*X + 1*Y + 0*X
vaddps  266372+A2(...), %ymm0, %ymm2 # (t+1) from 2*Z*Y*X + 1*Y*X + 1*Y + 1*X
vaddps  139396+A2(...), %ymm1, %ymm3 # (z+1) from 1*Z*Y*X + 2*Y*X + 1*Y + 1*X
vaddps  135428+A2(...), %ymm4, %ymm6 # (y+1) from 1*Z*Y*X + 1*Y*X + 2*Y + 1*X
vaddps  135304+A2(...), %ymm5, %ymm7 # (x+1) from 1*Z*Y*X + 1*Y*X + 1*Y + 2*X
vaddps  %ymm7,  %ymm6,  %ymm9
vaddps  %ymm3,  %ymm2,  %ymm8
vaddps  %ymm14, %ymm13, %ymm2
vaddps  %ymm9,  %ymm8,  %ymm10
vmovups %ymm10, 135300+A1(...)        # (x)   from 1*Z*Y*X + 1*Y*X + 1*Y + 1*X
```

Listing (2.2) AVX2 assembly generated by ICC 17.04

Hand-vectorization after data-layout transformation

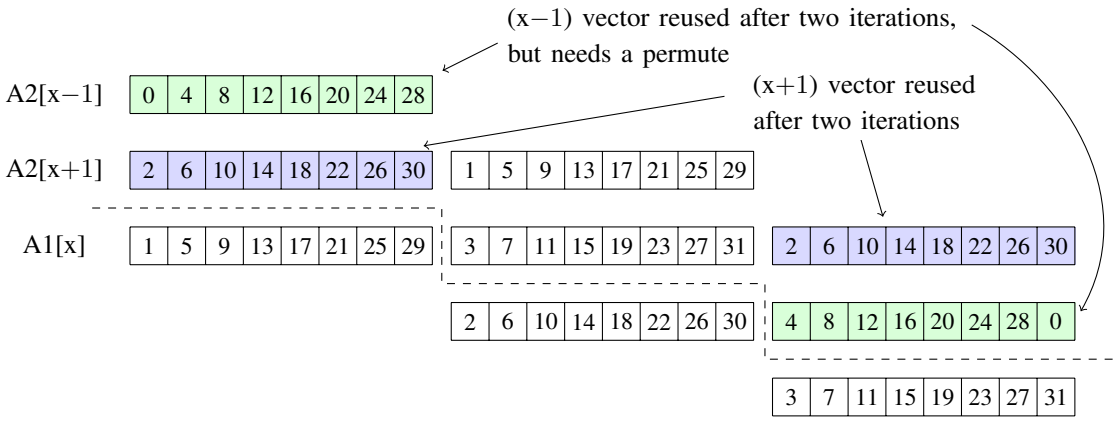
Custom hand-vectorization after a data-layout transformation yielded 35% better performance over the default ICC -O3 auto-vectorization. Our hand-vectorization used a data-layout transformation converting the arrays `A1` and `A2` to an AOSOA data-layout, and the writing AVX2 vector intrinsic manually. Chapter 6 presents in detail the data-layout transformation and auto-vectorization based on the data-layout transformation.



(a) Default vectorization with row-major layout

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Custom data-layout after reshaping and transposing the X-dimension



(b) Custom vectorization after a data-layout transformation

Figure 2.3: Improving SIMD vectorization after a data-layout transformation

Discussion

Figure 2.3a illustrates why custom vectorization after a data-layout transformation is an improvement over auto-vectorization over the default data-layout. In the default case, the two SIMD registers for the `A2[x+1]` and `A2[x-1]` shares multiple elements. The overlap of the data in SIMD registers limits reuse of the registers. Figure 2.3b shows the data reorganization done by the data-layout transformation. The layout transformation is akin to a *gather-scatter* reorganization of the array elements. The effect of this transformation creates a two-dimensional tiled vector data-layout. The tiled data-layout ensures that each register in a tile has multiple reuses. The internal rows are directly reused after one iteration, and the only SIMD permutations needed are at the boundaries of the vector tile. The data-layout transformation is the primary reason why our custom hand-vectorization outperforms the existing compiler auto-vectorization for this stencil computation. Applying this type of data-layout transformation by hand is not feasible, it requires low-level programming skills and is error-prone. The low-level code offers minimal portability as each architecture has a different vector instruction set extension.

Chapter Review

This chapter discussed two important issues that DSLs in HPC have to tackle. Both code-generation methods for EDSLs and data-placement abstractions play an important role in combining high programmer productivity with high computational efficiency. Chapter 3 introduces how the QUARC framework handles these two specific issues. Subsequent chapters then go into the specific design and implementation details of QUARC's code-generation technique and of its data-placement abstractions.

CHAPTER 3: THE QUARC FRAMEWORK

QCD's Array-based Rapid-prototyping Compiler (QUARC) is a framework for creating EDSLs using C++14 for lattice and grid-based domains (Deb *et al.*, 2016, 2017). QUARC consists of a high-level programming interface embedded in C++14, a domain-specific compiler that uses LLVM (The LLVM Foundation, 2018), and a runtime library for MPI parallelism. Figure 3.1 presents QUARC's high-level system architecture. QUARC's front-end has a split-language programming interface with two programming layers: Minimal Expression Template Array Language (METAL) and Array Transformation Language (ATL). METAL is the interface to build EDSLs using QUARC, and is a notational, array-based, implicitly data-parallel C++14 header-only library. METAL programs are free of explicit parallel constructs and make it easy for domain-experts to write readable code. ATL is a small specification language based on YAML (Oren Ben-Kiki, Clark Evans, Brian Ingerson, 2009). ATL specifications define the data-distribution and data-layout of abstract METAL arrays.

QUARC uses a domain-specific compiler, QUARC's Optimizer (QOPT), to compile METAL programs. QOPT is a plug-in to the LLVM compiler framework that defines a set of domain-specific analysis and code-generation passes. QOPT passes execute before the standard LLVM compiler passes and lower METAL abstractions into standard LLVM IR. This step involves domain-specific optimizations, MPI library call generation, SIMD vectorization, array access linearization, and loop generation. After compiling the high-level METAL code, QOPT invokes the standard LLVM optimization and code-generation passes to optimize the code further and to lower it to a binary executable.

The QUARC's Runtime (QUARC-RT) library is the final component of QUARC. QUARC-RT has a parser for ATL specifications, a polyhedral analyzer to compute MPI data movement, a set of wrapper functions for MPI routines, and a mapping interface to map distributed METAL arrays to MPI Cartesian communicators.

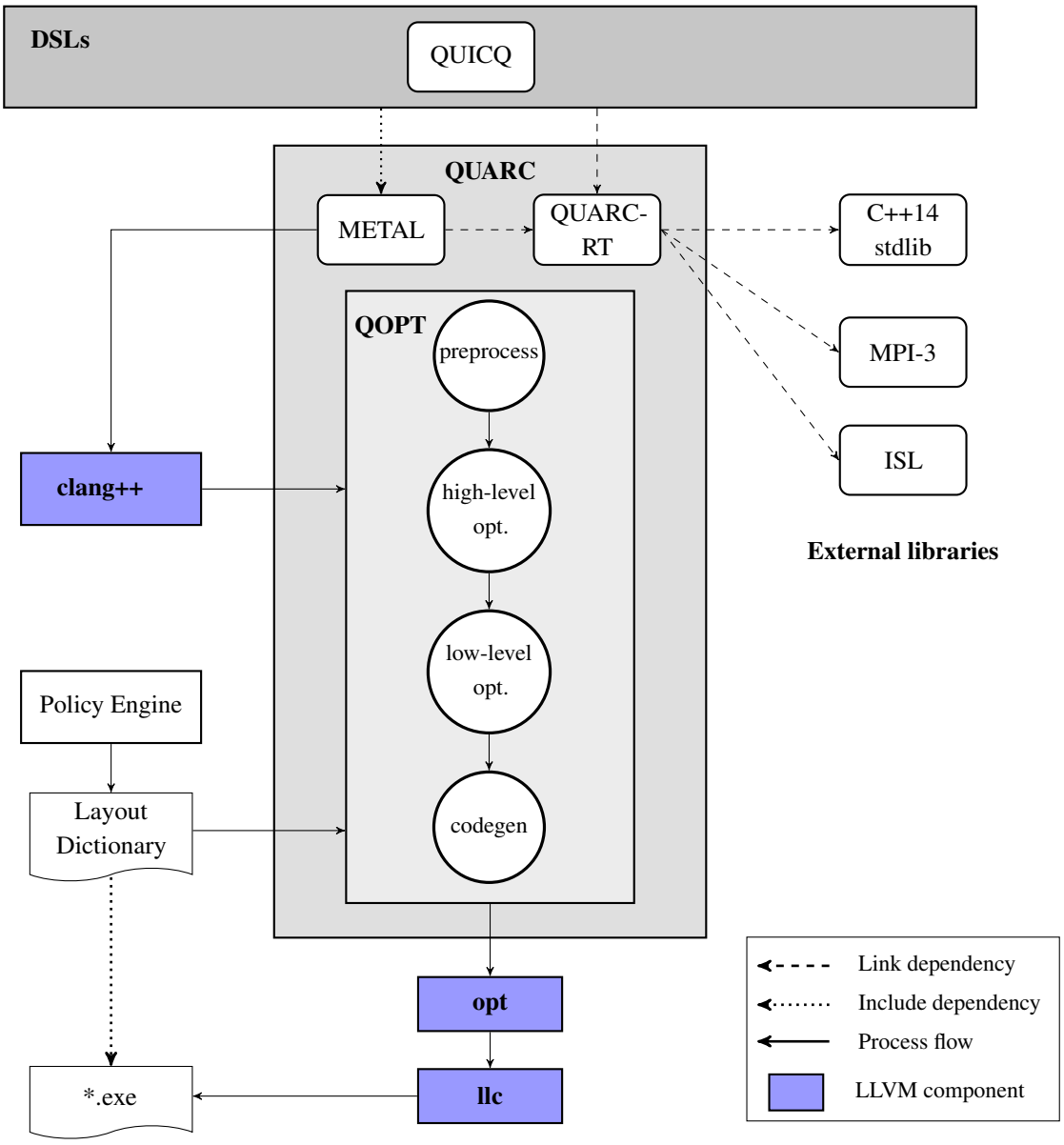


Figure 3.1: QUARC system architecture diagram

3.1 EDSL Design Approach

QUARC's EDSL design approach combines the best ideas in several established EDSL techniques, such as metaprogramming and split-languages. Chapter 2 surveyed these EDSL code-generation techniques. Although based on established ideas, QUARC's EDSL design approach has important enhancements that we list in this section.

QUARC uses C++ TMP to implement its METAL front-end. However, the use of C++ TMP differs from conventional C++ TMP-based EDSL designs. Conventional C++ TMP-based EDSLs generate low-level code using TMP. METAL takes an alternative approach, and uses C++ TMP to generate a domain-specific IR. The domain-specific IR consists of a set of side-effect-free domain-specific function calls. Using this approach, METAL communicates much of the high-level domain-specific semantics to QOPT. QOPT then uses the information for high-level optimizations on domain-specific constructs, and generates data-parallel code.

QUARC's approach bears similarities to MSP. Like most MSP-based methods, QUARC involves multiple levels of staging and code-generation. Our design offers a novel way of staging a domain-specific IR using side-effect-free domain-specific function calls inside an industry-standard compiler IR. These domain-specific function calls are equivalent to staging annotations in MSP. QUARC's domain-specific IR is legal LLVM IR, and as such, is analyzable *via* standard LLVM compiler passes. Analyzing a domain-specific IR using standard compiler passes makes it easier to implement domain-specific optimization and code-generation passes, and lowers the overall engineering cost of QOPT.

QUARC's use of the split-language design is limited when compared to those of other EDSLs frameworks that take a similar design approach. Other frameworks such as Halide and SPIRAL offload a significant chunk of the code generation logic into the specification layer of the split-language interface. The ATL interface is much smaller in comparison, and exposes as little as possible of the code-generation process to end users. Instead, QUARC leverages standard compiler analysis to make domain-specific code-generation decisions.

QUARC eschews JIT compilation for ahead-of-time code-generation. The ahead-of-time code-generation strategy generates multiple code versions based on prior application profiling. Application profiling and subsequent feedback to QOPT is separate from QUARC's overall infrastructure. The primary use of the strategy is in QOPT's SIMD auto-vectorizer. The auto-vectorizer generates multiple

vectorized versions of each METAL array expression based on different data-layout choices. The layout choices are provided as compiler flags to QOPT. The decision to use ahead-of-time code-generation instead of JIT compilation was based primarily on the ease of implementation. Integrating a JIT compiler into QUARC-RT would have entailed additional software engineering work, with minimal benefit for current use cases. It is a case for future consideration.

3.2 Parallel Programming Model

QUARC has an implicit data-parallel programming model exposed *via* METAL's array programming interface. METAL is free of explicit parallelization constructs; therefore, programmers do not have to reason about actual data-parallel execution of their application-level code. Instead, the ATL-specified data placement of an application's array-data types decides the parallel execution. Due to this programming model, programmers can adapt the execution of their application by only changing the ATL specification. Depending on the type of ATL specification, the same application can execute in serial, in parallel on multiple cores, or in parallel across distributed memory nodes.

The implementation of QUARC relies on MPI. Thus, QUARC shares MPI's memory model. All participating processors have their own private address spaces, and explicit communication is needed to move data among processors. QUARC-RT internally uses various optimizations to ensure that the communication overheads remain low. Chapter 6 explains these optimizations in detail.

3.3 The Core Components of QUARC

3.3.1 Minimal Expression Template Array Language (METAL)

METAL is a high-level array programming language that uses C++14 TMP. The language defines array containers, data-parallel operators, and array-expression data types. These basic constructs are composable into data-parallel array expressions. METAL array expressions can combine arrays and C++ scalar types. The implementation of METAL uses a new metaprogramming technique called Abstraction Characterization Templates (sACTs). ACTs use template recursion to generate a forest of side-effect-free function, or *DSL intrinsic*, calls. The DSL intrinsic calls encode the complete expression-tree of METAL array expressions and semantic information about the expression-tree nodes into QOPT's domain-specific IR. QOPT reconstructs METAL expression-trees by recognizing the encoded DSL intrinsic calls.

METAL's semantics are similar to other languages supporting array objects, such as FORTRAN 90 and High-performance FORTRAN (HPF). METAL behaves as though it fully evaluates the Right-Hand Side (RHS) of an array-assignment expression without side effect, and only then modifies the Left-Hand Side (LHS) sub-expression of an array-assignment. Chapter 5 presents the METAL language, its design, and its implementation details.

3.3.2 Array Transformation Language (ATL)

ATL is a small specification language based on YAML that specifies data placement of high-level METAL arrays. An ATL specification serves three main purposes: it specifies the data-distribution of an array, it specifies the data-layout of the array, and it specifies the mapping of the array blocks, or partitions to an MPI Cartesian communicator. Section 5.3 presents the implementation details of ATL. Chapter 4 presents the underlying algebra that ATL uses to define data-distributions and data-layouts.

3.3.3 QUARC Optimizer (QOPT)

QOPT is an LLVM-based domain-specific compiler for METAL, and is a plug-in of LLVM's Opt module. QOPT handles all low-level code-generation and parallelization decision for every METAL array expression. In the current implementation of QUARC, this includes speculative SIMD vectorization on x86_64 platforms, shared-memory parallelism using MPI-3, distributed memory parallelism using MPI-2, and other low-level optimizations. Chapter 6 presents the details of these optimizations, along with a detailed elucidation of QOPT's design and implementation.

3.3.4 QUARC Runtime (QUARC-RT)

The QUARC-RT library has wrapper functions for MPI operations, a parser for ATL specifications, and data-distribution functions. Section 6.5 explains the library and its components.

CHAPTER 4: AN ALGEBRA FOR ARRAY TRANSFORMATIONS

QUARC's data-placement abstractions are based on a more general array-transformation algebra. This chapter presents the formal semantics, legality constraints, and composability rules of this algebra. The algebra defines array index-space transformations using two operators, *reshape* (ρ) and *transpose* (ϕ). A ρ transformation reshapes array dimensions by partitioning existing dimensions, and creates a partitioned index-space. The ϕ transformation permutes array dimensions. Reshaping and permuting dimensions create new array index-spaces. The $\rho\phi$ algebra does not specify how an array index-space is mapped to a memory address space. Instead, it specifies how to generate a map between an initial index-space and a transformed index-space. Language implementations on top of this algebra must generate the needed data transformations based on the index-space maps.

QUARC's ρ and ϕ operators resemble similarly named operators defined by APL (Iverson, 1962) and by other array programming languages. However, QUARC's definition and usage of these operators differ from those of other array languages. Section 4.1.2 notes the differences.

Notation. We use the following notation to describe formally QUARC's array-transformation algebra. Lower-case Greek letters identify operators. All operators are written in C/C++ function-call syntax. As in C++, the term "vector" is used for a one-dimensional sequence container. Some of our array notations follow the style of Mullin's Mathematics of Arrays (Mullin, 1988). We use angle brackets, $\langle\rangle$, to represent vectors. Parentheses ( ) denote arrays. For arrays with more than two dimensions, the parentheses are nested. The uppercase letter $\mathbf{A}$ represents a typical QUARC array container wherever it is used in a definition. We use the notation $\mathbf{A}^n$ wherever the dimensionality of an array is mentioned, n being the number of dimensions of the array. The lowercase letter $\boldsymbol{v}$ represents an arbitrary vector. Array and vector indices are zero-based and are read from left to right. The usual C/C++ subscript operator `[]` is used to denote indexing into arrays and vectors. The $\stackrel{\Delta}{=}$ operator is used in all definitions to denote equivalence of two expressions. The $=$ operator is used as a relational operator.

4.1 Basic Operators

Definition 4.1. Dimensionality ($\delta(\mathbf{A})$)

The unary δ operator returns the rank or dimensionality of an array.

$$\delta(\mathbf{A}^n) \triangleq n. \tag{4.1}$$

Definition 4.2. Shape ($\sigma(\mathbf{A})$)

The unary σ operator takes either an array or a vector as its argument. For an array, it returns as a vector the number of components in each dimension of the array. For a vector argument, σ returns the total number of elements in the vector.

$$\sigma(\mathbf{A}) \triangleq \boldsymbol{s} \triangleq \langle e_0, \ldots, e_{(\delta\mathbf{A})-1} \rangle. \tag{4.2}$$

The components of vector, $\boldsymbol{s}$, are positive integers. Each component gives the extent of an array dimension. As QUARC does not support zero-ranked arrays, each component of $\boldsymbol{s}$ must be a positive integer greater than one, *i.e.*, $\forall i,\ \boldsymbol{s}[i] > 1$.

Note. APL users would recognize σ as the same operator as APL's monadic ρ shape operator. We chose to use a different symbol to avoid confusion with the dyadic ρ reshape operator.

Example 4.1.

For the following two-dimensional array

$$\mathbf{A} \triangleq \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{pmatrix},$$

$$\sigma(\mathbf{A}) \triangleq \langle 3, 4 \rangle.$$

Example 4.2.

Using the same array from Example 4.1.

$$\sigma(\sigma(\mathbf{A})) \triangleq 2.$$

Definition 4.3. Product ($\pi(\boldsymbol{v})$)

The binary π operator returns the cumulative product of the components of a vector. It requires two input arguments: a vector and a start-index position. The start-index position argument defaults to 0, and can be omitted.

$$\pi(\boldsymbol{v}, j) \triangleq \prod_{i=j}^{\sigma(\boldsymbol{v})-1} \boldsymbol{v}[i]. \tag{4.3}$$

Example 4.3.

Given a vector ($\boldsymbol{v}$) $\langle a, b, c, d\rangle$,

$$\pi(\boldsymbol{v}) \triangleq \pi(\boldsymbol{v}, 0) \triangleq a \cdot b \cdot c \cdot d$$

$$\pi(\boldsymbol{v}, 1) \triangleq b \cdot c \cdot d.$$

Definition 4.4. Stride ($\boldsymbol{st}$)

A stride is the number of array elements that must be traversed to reach the next array element along an array dimensional axis. The vector $\boldsymbol{st}$ denotes the stride in every dimension for an array.

$$\boldsymbol{st}[i] \triangleq \begin{cases} \pi(\boldsymbol{s}, i+1) & \text{if } 0 \leq i < \delta(\mathbf{A}) - 2 \\ 1 & \text{otherwise} \end{cases} \quad , where\ 0 \leq i < \delta(\mathbf{A}). \tag{4.4}$$

Example 4.4.

Let $\mathbf{A}$ be a four-dimensional array with a row-major lexicographic data-layout, and $\sigma(\mathbf{A}^4) \triangleq \langle a, b, c, d\rangle$.

$$\boldsymbol{st} \triangleq \langle b \cdot c \cdot d, c \cdot d, d, 1\rangle$$

Definition 4.5. Block Dimension

A block dimension is a new dimension created by partitioning an existing array dimension. A block dimension cannot be further reshaped.

Definition 4.6. Dimensional Attribute ($\boldsymbol{da}$)

A vector of Boolean values, each of which specifies the type of an array dimension. The value is '1' for a block dimension and '0' otherwise.

Definition 4.7. Reshape (ρ)

The binary ρ operator splits every array dimension based on a corresponding reshape factor. The operator takes an integral vector argument ($\boldsymbol{p}$). The components of $\boldsymbol{p}$ specify the reshape factors for all array dimensions. A legal $\boldsymbol{p}$ vector is defined as follows:

$$\delta(\mathbf{A}) = \sigma(\boldsymbol{p}). \tag{P1}$$

$$1 \leq \boldsymbol{p}[i] < \boldsymbol{s}[i],\ \forall i \mid 0 \leq i < \sigma(\boldsymbol{s}). \tag{P2}$$

$$\boldsymbol{s}[i] \bmod \boldsymbol{p}[i] = 0,\ \forall i \mid 0 \leq i < \sigma(\boldsymbol{s}). \tag{P3}$$

$$\boldsymbol{p}[i] = 1,\ \forall i \mid 0 \leq i < \sigma(\boldsymbol{s})\ and\ \boldsymbol{da}[i] = 1. \tag{P4}$$

- P1 states that a reshape factor is needed for each array dimension. Thus, multiple dimensions may be reshaped together.

- P2 states that the reshape factor should be between one and the extent of a dimension. As reshaping involves integral division, the factor needs to be greater than 0. A reshape factor also cannot be equal to the extent of the dimension. If permitted, such a reshape operation would only create a superfluous unit-length dimension.

- P3 states that each reshape factor should split a dimension evenly. This constraint is primarily there to simplify indexing operations. Future extensions to QUARC may relax this constraint by handling uneven divisions using array padding.

- P4 states that a block dimension cannot be reshaped. Block dimensions are meant to be mapped to an address space. Thus, reshaping a block dimension is not permitted. It is treated as immutable, once defined. Note that an original array dimension can be reshaped multiple times to create multiple levels of block dimensions.

ρ updates $\boldsymbol{s}$ to $\boldsymbol{s}^{'}$, and $\boldsymbol{da}$ to $\boldsymbol{da}^{'}$ for $\mathbf{A}$ as follows:

$$(\boldsymbol{s}^{'})_{j=0}^{2i} \triangleq \begin{cases} \boldsymbol{p}[\frac{j}{2}] & \text{if } j \bmod 2 = 0 \\ \frac{\boldsymbol{s}[\frac{j}{2}]}{\boldsymbol{p}[\frac{j}{2}]} & \text{otherwise} \end{cases} \quad , where\ 0 \leq i < \sigma(\boldsymbol{s}). \tag{4.5}$$

$$(\boldsymbol{da}^{'})_{j=0}^{2i} \triangleq \begin{cases} 1 & \text{if } j \bmod 2 = 0 \\ 0 & \text{otherwise} \end{cases} \quad , where\ 0 \leq i < \sigma(\boldsymbol{s}). \tag{4.6}$$

Equations 4.5 and 4.6 add entries into the new shape and dimensional attribute vectors. Both equations add entries in the new vectors, including a unit-reshape factor even though a unit-reshape factor performs no reshape. Therefore, as a final step, ρ removes all components corresponding to unit-length block dimensions from $\boldsymbol{s}^{'}$, $\boldsymbol{da}^{'}$, and $\boldsymbol{st}^{'}$ to create three new vectors, $\boldsymbol{s}^{''}$, $\boldsymbol{da}^{''}$, and $\boldsymbol{st}^{''}$.

$$\boldsymbol{s}^{''} \triangleq \boldsymbol{s}^{'}_{-k}, \tag{4.7}$$

$$\boldsymbol{da}^{''} \triangleq \boldsymbol{da}^{'}_{-k}, \tag{4.8}$$

where $\forall k |\ 0 \leq k < \sigma(\boldsymbol{s}^{'})\ and\ \boldsymbol{s}^{'}[k] = 1\ and\ \boldsymbol{da}^{'}[k] = 1$. The notation $\mathbf{v}^{'}_{-k}$ indicates the removal of the k^{th} entry from a vector $\mathbf{v}$.

Example 4.5.

This example uses a two-dimensional array, $\mathbf{A}$, with initial $\boldsymbol{s}$ equaling $\langle 64, 64 \rangle$. To demonstrate the effect of reshaping, every array element is shown as a two-tuple consisting of the element's initial two-dimensional index.

$$\mathbf{A} \triangleq \begin{pmatrix} 0,0 & 0,1 & \dots & 0,63 \\ \vdots & \vdots & \vdots & \vdots \\ 63,0 & 63,1 & \dots & 63,63 \end{pmatrix}$$

$$\rho(\mathbf{A}, \langle 2,2 \rangle) \triangleq \left(\begin{pmatrix} 0,0 & 0,1 & \dots & 0,31 \\ 0,32 & 0,33 & \dots & 0,63 \end{pmatrix} \begin{pmatrix} \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} 31,0 & 0,1 & \dots & 31,31 \\ 31,32 & 31,33 & \dots & 31,63 \end{pmatrix} \right.$$
$$\left. \begin{pmatrix} 32,0 & 32,1 & \dots & 32,31 \\ 32,32 & 32,33 & \dots & 32,63 \end{pmatrix} \begin{pmatrix} \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} 63,0 & 63,1 & \dots & 63,31 \\ 63,32 & 63,33 & \dots & 63,63 \end{pmatrix} \right)$$

After the reshape transformation, the new shape-vector for $\mathbf{A}$ is $\langle 2, 32, 2, 32\rangle$. The notation should be read as a 2×32 array of 2×32 blocks. The index positions inside a block are contiguous. A reshape transformation does not change the lexicographic ordering of the original indices.

Definition 4.8. Transpose (ϕ)

The binary ϕ operator permutes the dimensions of an array using a permutation vector ($\boldsymbol{p}$). ϕ permutes the existing $\boldsymbol{s}$, and $\boldsymbol{da}$, attributes of $\mathbf{A}$. A legal permutation vector, $\boldsymbol{p}$, is defined as follows:

$$\delta(\mathbf{A}) = \sigma(\boldsymbol{p}). \tag{P5}$$

$$\boldsymbol{p}[i] \neq \boldsymbol{p}[j],\ \forall i, j \mid 0 \leq i, j < \sigma(\boldsymbol{s})\ and\ i = j. \tag{P6}$$

$$0 \leq \boldsymbol{p}[i] < \delta(\mathbf{A}),\ \forall i \mid 0 \leq i < \sigma(\boldsymbol{s}). \tag{P7}$$

P5 states that the size of the permutation vector should be equal to rank of the array. P6 states that the permutation vector should not have repeated values. P7 states that a permutation vector should contain only values corresponding to the position of an array dimension.

ϕ updates the existing $\boldsymbol{s}$ to $\boldsymbol{s}'$, and $\boldsymbol{da}$ to $\boldsymbol{da}'$, for $\mathbf{A}$ as follows:

$$\boldsymbol{s}''[i] \triangleq \boldsymbol{s}'[\boldsymbol{p}[i]], \tag{4.9}$$

$$\boldsymbol{da}''[i] \triangleq \boldsymbol{da}'[\boldsymbol{p}[i]], \tag{4.10}$$

where $0 \leq i < \sigma(\boldsymbol{s}')$.

Example 4.6.

This example applies a ϕ transformation to the reshaped array created in Example 4.5. The transformed array's shape was $\langle 2, 32, 2, 32\rangle$. Following are two examples of possible ϕ transformations of this array.

$$\phi(\mathbf{A}, \langle 0,2,1,3 \rangle) \triangleq \begin{pmatrix} \begin{pmatrix} 0,0 & \dots & 0,31 \\ 1,0 & \dots & 1,31 \\ \vdots & \vdots & \vdots \\ 31,0 & \dots & 31,31 \end{pmatrix} & \begin{pmatrix} 0,32 & \dots & 0,63 \\ 1,32 & \dots & 1,63 \\ \vdots & \vdots & \vdots \\ 31,32 & \dots & 31,63 \end{pmatrix} \\ \begin{pmatrix} 32,0 & \dots & 32,31 \\ 33,0 & \dots & 33,31 \\ \vdots & \vdots & \vdots \\ 63,0 & \dots & 63,31 \end{pmatrix} & \begin{pmatrix} 32,32 & \dots & 32,63 \\ 33,32 & \dots & 33,63 \\ \vdots & \vdots & \vdots \\ 63,32 & \dots & 63,63 \end{pmatrix} \end{pmatrix}$$

The ϕ transformation permuted the second and the third dimensions of the reshaped array. The array was transformed from a 2×32 array of 2×32 blocks to a 2×2 array of 32×32 blocks. Note that the ϕ transformation reorders the original indices. This transposition is useful when partitioning an array across multiple processors.

$$\phi(\mathbf{A}, \langle 1,3,0,2 \rangle) \triangleq \begin{pmatrix} \begin{pmatrix} (\ 0,0 & 0,32\ ) & (\ 32,0 & 32,32\ ) \\ \vdots & \vdots & \vdots & \vdots \\ (\ 0,31 & 31,63\ ) & (\ 32,32 & 32,63\ ) \end{pmatrix} \\ \begin{pmatrix} (\ 31,0 & 31,32\ ) & (\ 33,0 & 33,32\ ) \\ \vdots & \vdots & \vdots & \vdots \\ (\ 31,31 & 31,63\ ) & (\ 63,32 & 63,63\ ) \end{pmatrix} \end{pmatrix}$$

This second ϕ transformation is a different permutation of the array dimensions. To clarify, the overall permutation can be viewed as the following series of permutations.

$$\phi(\mathbf{A}, \langle 1,0,2,3 \rangle) \rightarrow \phi(\mathbf{A}, \langle 1,0,3,2 \rangle) \rightarrow \phi(\mathbf{A}, \langle 1,3,0,2 \rangle)$$

The resulting final shape of the array is $\langle 32,32,2,2 \rangle$. This ϕ transformation transposes the block dimensions in the opposite direction as compared to the previous transformation. This transformation is useful when indices are rearranged to create an inner SIMD vector dimension that is lifted from outer

dimensions. Here, the innermost dimensions may be composed together to generate a four-wide SIMD vector dimension.

4.1.1 Composing array-transformations

The ρ and ϕ operators both update the $\boldsymbol{s}$ and $\boldsymbol{da}$ vector of $\mathbf{A}$. The transformations may be composed together if their input arguments do not violate any legality constraints.

A ϕ transformation is invertible. A ρ transformation, however is not invertible. To invert a ρ operation, our algebra would have to be extended *via* a concatenation or a ravel operator. Such operators are present in APL and in other array algebras, such as Mullin's Mathematics of Arrays and More's Array Theory (More, 1973). QUARC's present use cases did not require a ravel operator, and it was omitted from this algebra. As such, any sequence of array-transformations that involve a reshape transformation is not invertible.

4.1.2 Comparison of QUARC's array-transformations to APL

QUARC's ρ and ϕ operators are eponyms of APL's (Iverson, 1962) ρ and ϕ operators. The general semantics of the two sets of operators is similar. However, QUARC's operators differ from APL's in some important ways.

APL's dyadic ρ operator requires as its input a new shape-vector for an array. It applies the new shape to an existing index-space. Thus, it may increase or decrease an array's rank. APL's reshape operator accepts a shape-vector argument even when the product of that vector's components does not equal the total number of array elements. APL handles such cases either by ignoring all extra elements where the product is lesser, or by wrapping around when the product is greater. In comparison, QUARC's ρ operator is an index-space partitioning operator. It partitions the existing index-space to create a new shape that can only increase an array's rank.

Similarly, QUARC's ϕ operator is only a subset of APL's ϕ operator. QUARC does not permit repeats in the permutation vector argument of ϕ. The permutation vector argument to APL's ϕ operator can have repeated values, and produces what is known as a *diagonal section* of the transformed array. We would refer readers to (More, 1973; Mullin, 1988) for a detailed elucidation of APL's formal algebra.

4.2 Index-space Mapping

This last section describes the generation of mapping functions between a lexicographic index-space and an index-space created using a $\rho\phi$ transformation. These functions are the basis of writing data-redistribution or data-layout transformation routines.

First, we introduce another auxiliary operator (ι) to help describe data-mapping functions.

Definition 4.9. Index (ι)

The binary ι operator returns an index offset (I) from a datum. This operator is used to calculate a memory address within an array. It takes two vector arguments, an index vector ($\boldsymbol{i}$) and a stride vector ($\boldsymbol{st}$). ι is defined as follows:

$$I \triangleq (\boldsymbol{i}, \boldsymbol{st}) \qquad \triangleq \sum_{0}^{j=\delta(\boldsymbol{st})} \mathbf{i}[j] \times \boldsymbol{st}[j], where\ \delta(\boldsymbol{st}) = \delta(\mathbf{i}). \tag{4.11}$$

Let $\boldsymbol{i}$ be an index vector for an array $\mathbf{A}$, and $\boldsymbol{i}'$ be an index vector for the array $\mathbf{A}'$ that is produced by a ρ transformation on $\mathbf{A}$. Then, $\boldsymbol{i}'$ is derived as follows.

$$(\boldsymbol{i}'')_{j=0}^{2k} \triangleq \begin{cases} \dfrac{\boldsymbol{i}[\frac{j}{2}]}{\boldsymbol{s}[\frac{j}{2}]} & \text{if } j \bmod 2 = 0 \\ \boldsymbol{i} \bmod \boldsymbol{s}[\dfrac{j}{2}] & \text{otherwise} \end{cases} \qquad , where\ 0 \leq k < \sigma(\boldsymbol{s}). \tag{4.12}$$

$$\boldsymbol{i}' \triangleq \boldsymbol{i}''_{-k}. \tag{4.13}$$

With this definition for the index vector for the transformed array, the mapping function from $\mathbf{A}$ to $\mathbf{A}'$ is derived as follows.

$$\mathbf{A}'[\iota(\boldsymbol{i}', \boldsymbol{st}')] = \mathbf{A}[\iota(\mathbf{i}, \boldsymbol{st})], \tag{4.14}$$

where $\boldsymbol{i}$ spans the space of all indices of $\mathbf{A}$. The inverse mapping is simply the reverse assignment.

For ϕ transformations, $\boldsymbol{i}'$ is derived by permuting the original index vector using the same permutation vector.

Example 4.7.

Given an $\mathbf{A}^2$ of shape 32×32 that is $\rho\phi$ transformed as follows:

$$\rho(\mathbf{A}, \langle 2, 2\rangle) \rightarrow \phi(\mathbf{A}, \langle 1, 3, 0, 2\rangle).$$

This transformation results in the layout shown in Example 4.6. Then, a possible implementation of the mapping from the default lexicographic data-layout to the new data-layout is shown by the following C++ loop nest.

```
for(auto y = 0ul; y < Y; ++y)
  for(auto x = 0ul; x < X; ++x)
    A_transformed[y%(Y/2)][x%(X/2)][y/(Y/2)][x/(X/2)] = A[y][x];
```

Copying data back from the transformed data-layout to the original row-major lexicographic data-layout can be done simply by reversing the assignment.

```
for(auto y = 0ul; y < Y; ++y)
  for(auto x = 0ul; x < X; ++x)
    A[y][x] = A_transformed[y%(Y/2)][x%(X/2)][y/(Y/2)][x/(X/2)];
```

Chapter Review

This chapter presented the array-transformation algebra that is used by QUARC to define its data-placement abstractions. Although based on similar operators in APL, QUARC's operators have different semantics. Section 5.3 in Chapter 5 explains the use of this algebra in the ATL specification language.

CHAPTER 5: PROGRAMMING INTERFACE

QUARC has a two-level split programming interface. Application programs are written in the METAL, an extension to C++14 that implements an implicitly data-parallel array programming interface. An application-level METAL array does not have an intrinsic data-layout or data-distribution. These attributes are added at runtime using the ATL. ATL is a small specification language based on YAML (Oren Ben-Kiki, Clark Evans, Brian Ingerson, 2009). ATL controls the runtime data-parallel execution of METAL programs. There are several benefits to this split programming interface. It keeps METAL applications succinct, improves readability, and reduces maintenance overheads. It reduces the need to recompile a program for a different data-distribution or data-layout, and allows rapid prototyping. Any future QUARC code-generation target, such as GPGPUs, can be added by enhancing ATL and QUARC's code-generation and runtime libraries. Splitting ATL and METAL also enables auto-tuning in the space of data-layouts and data-distribution for METAL arrays.

METAL's Application Programming Interface (API) provides array containers, expressions objects, and data-parallel operators. The API allows developing expressive array-based EDSLs that are free of explicit parallelization constructs, elemental loops, and array accesses. Programming in METAL, or an EDSL developed on top of METAL, requires no custom annotations, pragmas or keywords. Section 5.1 describes METAL's grammar and API in detail. This section is primarily aimed at programmers implementing EDSLs on top of METAL, and can be skipped by general audiences.

METAL is nominally a C++ template-only library, unlike conventional C++ template-only libraries METAL templates do not generate low-level executable code inside C++. METAL templates do not *scalarize* array expressions, *i.e.*, they do not generate elemental loops and array accesses. Instead of generating elemental loops and array accesses, METAL generates an architecture-neutral domain-specific IR that encodes the expression tree for each METAL array expression. The encoded IR-generation uses a new design pattern called Abstraction Characterization Templates (sACTs). Every ACT function call represents a separate node of the parse tree. Internally, an ACT uses C++ TMP to generate calls of

specially annotated side-effect-free functions called domain-specific intrinsic (DSL intrinsic) functions. DSL intrinsic function calls encode the METAL constructs into the domain-specific IR generated for QUARC's compiler, QOPT. QOPT recognizes DSL intrinsic function calls, and recovers METAL expression trees and their properties. This allows various optimizations at the expression tree-level, followed by architecture-specific data-parallel code-generation. ACT and DSL intrinsic function calls get fully inlined over the course of code-generation. Section 5.2 presents the design and implementation details for ACTs.

Finally, Section 5.3 describes ATL. An ATL specification defines a METAL array's data-layout, and distribution over an actual processor grid. ATL uses $\rho\phi$ algebra to define operators that do these operations. It also provides a way to map METAL array blocks or partitions to actual processors. Depending on this mapping a METAL program can be executed serially, parallelized on multiple cores of a shared-memory node, or parallelized at a large scale on a distributed cluster.

5.1 Minimal Expression Template Array Language (METAL)

5.1.1 Grammar

Figure 5.1 shows METAL's complete EBNF grammar. METAL has a relatively small type system consisting of a dynamically allocated global array container, a fixed size array container, expression classes to define array expressions, and data-parallel operators for whole-array operations. Section 5.1.2 describes all these data types in detail.

The METAL high-level API does not implement any elemental functions for the global array container class. The implementation of elemental functions is left as a prerogative of the EDSL-layer developed on top of METAL.

5.1.2 Type System

5.1.2.1 Array Properties

Global shape (`glblshape`)

A `glblshape` type defines an abstract *global* index-space for a METAL distributed global array. A `glblshape` type is defined with two non-type template parameters. The first non-type parameter is

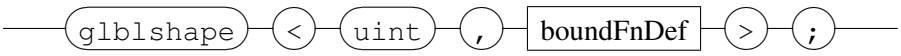
```
<quarc-kernel>   = <asgn-expr> | <reduction-expr>;
<asgn-expr>      = <mddarray-term> <asgn-op> <expr>;
<reduction-expr>= <supported-ty> "=" <reduce-expr>;
<expr>           = <if-even-choose-expr> | <choose-expr> | <gshift-expr> |
                   <drill-expr> | <mddarray-term> | <sdlarray-term> |
                   <binary-expr> | <unary-expr> | <reduce-expr>;
<reduce-expr>    = "REDUCE" "("<expr>","<accumulator-fn>")";
<bin-expr>       = <expr> <binary-op> <expr>;
<unary-expr>     = <unary-op> <expr>;
<if-even-choose-expr>
                 = "IF_EVEN_CHOOSE" "(" <gshift-expr> "," <gshift-expr> ")";
<choose-expr>    = "CHOOSE" "("<predicate-fn>","<gshift-expr>","<gshift-expr>")";
<gshift-expr>    = <mddarray-term> ["." "GSHIFT" "<" int{ "," int} ">"];
<drill-expr>     = "DRILL" "<" uint ">" "(" <mddarray-term> ")";
<mddarray-term> = <mddarray-ty> <id>;
<sdlarray-term> = <sdlarray-ty> <id>;
<mddarray-ty>    = "mddarray" "<" <supported-ty> "," <glblshape-ty> ">";
<binary-op>      = (operator<op> | <id>) "<"
                     <expr> "," <expr> "," <binary-mk-ty>
                   ">" "(" <expr> <id> "," <expr> <id> ")";
<unary-op>       = (operator<op> | <id>) "<" <expr>","<unary-mk-ty> ">"
                   "(" <expr> <id> ")";
<binary-mk-ty>   = <supported-ty> <id> "(" <supported-ty>","<supported-ty> ")";
<unary-mk-ty>    = <supported-ty> <id> "(" <supported-ty> ")";
<supported-ty>   = <arithmetic-ty> | <sdlarray-ty>;
<sdlarray-ty>    = "sdlarray" "<" (<arithmetic-ty> | <sdlarray-ty>)","uint ">";
<glblshape-ty>   = "glblshape" "<" uint "," <boundary-fn> ">";
<boundary-fn>    = uint "(" "*" ")" "(" int <id> "," int <id> ")";
<predicate-fn>   = bool <id> "(" uint {, unit} ")"
<accumulator-fn>= <supported-ty> <id> "(" <supported-ty>","<supported-ty> ")";
<arithmetic-ty> = (* Any C++ integral or floating point types *);
<op>             = (* Any C++ overloadable operator *);
<asgn-op>        = (* Any C++ assignment operator *);
<id>             = (* Any legal C++ identifier *);
```

Figure 5.1: METAL's EBNF grammar

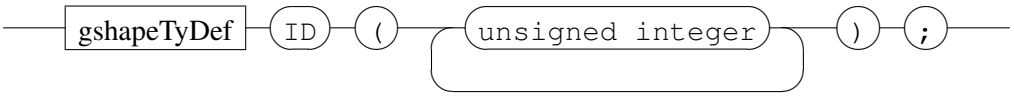
the rank of the index-space, and the second non-type parameter is a boundary function. A `glblshape` instance is created with a list of unsigned integer arguments, each of which is a dimensional upper-bound of the index-space. Every METAL global array is instantiated with a `glblshape` instance.

§Syntax

gshapeTyDef



gshapeDef



The `glblshapeTyDef` is the syntax rule for a `glblshape` type definition. `gshapeDef` is the syntax rule for instantiating a `glblshape` object. `ID` denotes a legal C++ identifier.

§Rationale. All global view arrays in a METAL array expression are required to have the same global shape. A separate data type for global shape makes it easier for programmers to follow this requirement. A single `glblshape` instance can be created, and shared by multiple array instances. Having a separate `glblshape` type also makes it easy for QUARC-RT to validate this requirement. QUARC-RT does so using pointer comparisons of the `glblshape` members of the arrays in an expression.

§**Implementation Note.** The `glblshape` class constructor uses C++14's variadic templates to initializes the extents of a `glblshape` instance in a type-safe manner. A compile-time check ensures that the number of extent arguments match the `glblshape` type's rank. By default the array boundary condition argument to a global shape object is set as a modulo periodic boundary function.

5.1.2.2 Array Containers

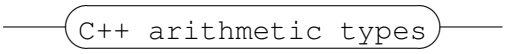
Single-dimensional local array (`sdlarray`)

An `sdlarray` is a fixed sized array container that can have numeric type elements, or can nest another `sdlarray`. Every `sdlarray` type definition needs two template arguments. The first argument specifies the data type of the elements of the `sdlarray`, and the second template argument statically specifies the number of elements. An `sdlarray` cannot be zero-dimensional, and does not satisfy

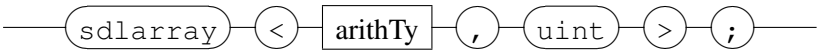
C++'s *plain-old-data-type* (POD) type trait. METAL allows defining `mddarray` global view arrays with an `sdlarray` element type.

§Syntax

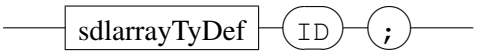
arithTy



sdlarrayTyDef



sdlarrayDef



The `arithTy` syntax rule is used here, and in subsequent syntax diagrams to represent all C++ arithmetic data types. The `sdlarrayTyDef` rule specifies an `sdlarray` type definition and the rule `sdlarrayDef` specifies instantiating an `sdlarray` object.

§Rationale. The `sdlarray` type has similar semantics to the standard C++ `std::array` type. For QUARC's design goal, the standard container was unsuitable and we implemented the `sdlarray` type for the following reasons.

- QUARC data-layout transformations can encompass outer `mddarray` dimensions and nested `sdlarray` dimensions. To ensure QOPT correctly translates nested `sdlarray` accesses after a layout transformation, it needs to recognize these accesses inside the IR. The `sdlarray` subscript operator is a METAL DSL intrinsics function that lets QOPT recognize these accesses, and recover a full delinearized view for each access.

- The standard array container class can be zero-dimensional, and it provides no guarantee that any nested array is allocated contiguously. `Sdlarray` cannot be zero-dimensional, and guarantees contiguous allocation of nested arrays.

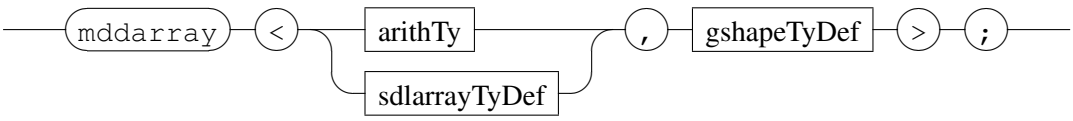
- By not being a POD, the `sdlarray` type prevents C++ compilers from implicitly optimizing `sdlarrays` copy operations. For example, LLVM's Clang C++ front-end implicitly converts `std::array` copy into `memcpy` calls. Such optimizations are advantageous in the general case, but impede QUARC's domain-specific code-generation.

Multi-dimensional distributed array (`mddarray`)

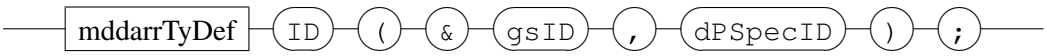
An `mddarray` is a distributed global view array container. Every `mddarray` type declaration requires an element type argument and a `glblshape` type argument. An `mddarray`'s elements can either be a C++'s scalar numeric type or an `sdlarray` type. The `mddarray` class constructor requires an ATL specification (Section 5.3) as an argument. The specification is parsed at runtime, and decides the data-distribution and data-layout of the `mddarray`. The data-placement of an `mddarray` is immutable. The class provides separate data copy functions to copy data in and out from an `mddarray`. Apart from the single constructor, the `mddarray` class does not provide any copy, move constructors or assignment operators. It also does not provide a `new` operator. Each `mddarray` instance is meant to be defined once, and then passed by reference everywhere. This is done to make reaching definition-based data-flow analyses easier inside QOPT.

§Syntax

mddarrTyDef



mddarrDef



The `mddarrTyDef` is the syntax rule for an `mddarray` type definition. `mddarrDef` is the rule to instantiate an `mddarray`. `gsID` is a pointer to a `glblshape` instance, and `dPSpecID` is an ATL spec file name.

§Rationale. C++ does not have multi-dimensional dynamic arrays. To get around this limitation, libraries, such as Boost MultiArray (Garcia *et al.*, 2001), Global Array Toolkit (Nieplocha *et al.*, 2006), and Kokkos (Carter Edwards *et al.*, 2014), added support for such arrays. For QUARC, we required a container that is both lightweight, and whose properties are recognizable by our underlying domain-specific compiler. The `mddarray` class uses DSL intrinsic calls to identify multi-dimensional accesses, constructor calls, and assignment operations. Recognizing these properties is a prerequisite for domain-specific optimizations and code-generation.

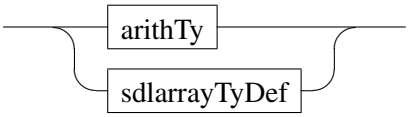
§**Implementation Note.** The `mddarray` class does not provide any copy or move constructor or assignment operators. It prevents scenarios such as returning and passing `mddarray` objects by value. `Mddarray` cannot be used inside standard C++ containers, such as `std::vector`. These limitations made aspects of QOPT's implementation simpler. QUARC provides a speculative SIMD vectorizer that analyzes the uses of an `mddarray` to generate vectorized code specialized for a set of data-layouts. The *def-use* analysis of an `mddarray` is simplified by making the `mddarray` class non-copyable and non-movable.

5.1.2.3 Elemental Functions

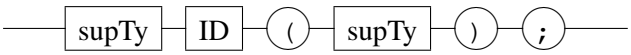
Elemental functions, or "mkernels", describe an operation applied to `mddarray` elements. As `mddarray` elements may be `sdlarray`, mkernels may operate on `sdlarray` types. Mkernels must be free of side-effects, and their definitions should be accessible inside the translation unit where they are used. Mkernels require a pass-by-value and return-by-value semantics that is enforced by METAL's API. Mkernels can be both binary or unary operators. METAL does not provide any mkernels, and EDSLs must define their own domain-specific mkernel functions.

§Syntax

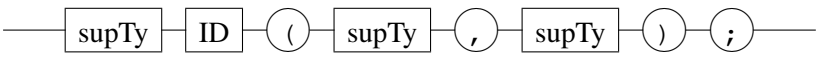
supTy



uMkernel



bMkernel



The `supTy` rule specifies the allowed data types for an mkernel function argument. These are also the types allowed as elements of an `mddarray`. Currently, only C++ arithmetic data types, `sdlarray`, or arithmetic types are allowed. `uMkernel` and `bMkernel` are the syntax rules for mkernel function signature. The rules specify that mkernels require both pass-by-value and return-by-value semantics.

§Rationale. The mkernel function's design allows QOPT to fully analyze these functions during code-generation. Being aware of the calling context of these functions, QOPT can make domain-specific code-generation decisions that are not possible otherwise. QOPT takes into consideration any data-layout transformations on the `mddarray`, and uses the information to SIMD vectorize the functions. QOPT also fully inlines these functions during code-generation.

§**Implementation Note.** During code-generation QOPT inlines all mkernels, and possibly converts all arithmetic operations to equivalent SIMD vectorized operations. For this reason, QUARC currently limits the type of operations that are permitted in an mkernel function. Mkernel functions are only allowed to have *static for-loops* that are defined using C++ templates. Static for-loops get fully unrolled during template expansion. Mkernels should not have any other control flow apart from these special template-based loop abstractions. Function calls are also not allowed within mkernel functions. In addition, all mkernel functions must have *static inline* qualifiers. This qualifier ensures that the definition of an mkernel function has internal linkage within the translation unit where it is used.

5.1.2.4 Array Operations

METAL array operation data types encapsulate mkernel functions, and generate DSL intrinsic function calls that identify the encapsulated mkernel inside QOPT's IR. METAL has two array operation data types: unary operation (`unary_op`) and binary operation (`binary_op`). As part of their type signature, both these data types require a non-type template argument specifying the mkernel function. The data types have a static `apply_op` function that calls back the encapsulated mkernel function. The `apply_op` function is a METAL DSL intrinsic function, and QOPT recognizes them inside its IR. Using this approach, QOPT identifies calls to the user defined mkernel functions inside its IR. The `apply_op` calls are inlined at the end of code-generation.

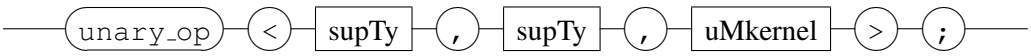
The array operation classes are never directly instantiated, rather they are part of the type signature of a METAL expression class. Template recursion generates the `apply_op` call at the point of evaluation of the expression class that encapsulates the array operation. Section 5.1.2.5 discusses evaluation of expression classes.

Unary operation (`unary_op`)

A `unary_op` object abstracts a unary operation that applies to a METAL sub-expression. Each `unary_op` type is defined to accept a unary mkernel. Every `unary_op` type needs three template arguments. The first argument is the input type accepted by the unary mkernel function encapsulated by the `unary_op`. The second argument specifies the return type of the mkernel. The last argument is a function pointer type specifying the type signature of the mkernel. A `unary_op` object is required to create a unary array expression.

§Syntax

unaryOpTy

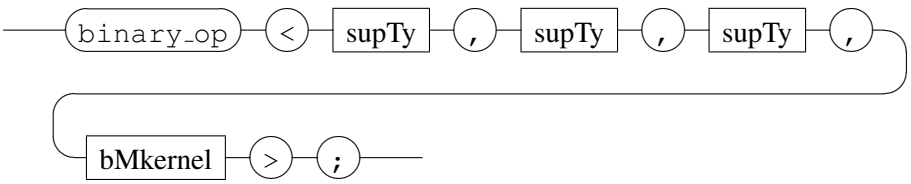


Binary operation (`binary_op`)

A `binary_op` object is analogous to a `unary_op`, but abstracts a binary operation that applies to two METAL sub-expressions. Each `binary_op` needs four template arguments. The first two are the input types accepted by the mkernel function encapsulated by the `binary_op`. The third argument is the return type of the mkernel. The fourth argument is a function pointer type specifying the type signature of the mkernel. A `binary_op` object is required to create a binary array expression.

§Syntax

binaryOpTy



§Rationale. Unary and binary operation objects serve two important roles. All mkernels in METAL are required to be pure functions that pass and return objects by value. The required mkernel signature is enforced by the unary and binary operation classes. These classes also free up EDSL developers from

having to manually annotate mkernel functions. `unary_op` and `binary_op` initialization internally invokes a DSL intrinsic that annotate the supplied mkernel function.

§**Implementation Note.** The `unary_op` and `binary_op` are not exposed by METAL's public API. Instead, METAL provides factory functions that EDSLs need to use to define new types of expressions. The factory functions create both the expression type and the needed operation type.

5.1.2.5 Array Expressions

METAL array expression are data-parallel operations over `mddarrays`. Array expressions abstract *foreach* operations where the same operation is performed on each array element in parallel. METAL supports unary, binary, and special expression types. EDSL built using METAL need to define the operators that create unary and binary expressions. These EDSL operators are abstractions for higher order array functions. An EDSL operator encapsulates an mkernel call back function, and generates a METAL unary or binary expression by calling a factory function.

METAL's API provides custom template functions that define the special expression types. The `DRILL` function is applied across the whole `mddarray` to indirectly access `sdlarray` elements nested at each location. The `GSHIFT` function defines a whole array shift of an `mddarray`. The `IF_EVEN_CHOOSE` function defines a built-in predicate that provides alternative actions at each location depending on the parity of the array index expression. The `REDUCE` function defines a reduction of another METAL expression.

Each array expression type includes a member template function called `evaluate_expr`. This template function needs to be invoked to evaluate an array expression. Typically, array expressions get evaluated inside `mddarray` assignment statements, and reduction assignment statements. Every expression type invokes a DSL intrinsic function when its `evaluate_expr` gets called. Binary and unary expressions invoke the `apply_op` function of the encapsulated array operation. An `mddarray` terminal expression would invoke an `array_access_fn` DSL intrinsic call. The next sections discuss the different expression types and their evaluation.

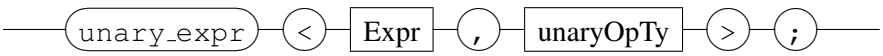
Unary expression (`unary_expr`)

A `unary_expr` abstracts a unary operation for a single METAL sub-expression.

§Syntax

Expr

unaryExprTy

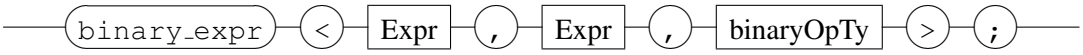


Binary expression (`binary_expr`)

A `binary_expr` abstracts a binary operation that applies to two METAL sub-expressions.

§Syntax

binaryExprTy



Mddarray terminal expression (`mddarray_term_expr`)

An `mddarray_term_expr` represents an `mddarray` element access. Evaluating this type of expressions generates an `access_fn` DSL intrinsic call. The DSL intrinsic function call captures the fully delinearized array access functions needed to index into the `mddarray`.

Scalar terminal expression (`scalar_term_expr`)

A `scalar_term_expr` is generated when a scalar arithmetic expression is used inside a METAL array expression. Scalar expressions are used for operations such as scaling of `mddarray` elements by a fixed value, and storing the output of a reduction.

Sdlarray terminal expression (`sdlarray_term_expr`)

A `sdlarray_term_expr` are like scalar terminal expressions, but use `sdlarray` values instead of arithmetic types.

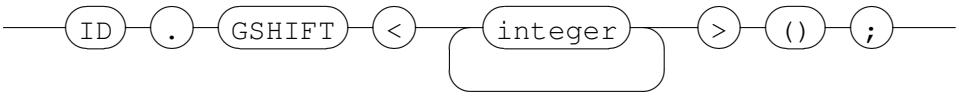
GSHIFT expression (`gshift_expr`)

A `gshift_expr` is a type of unary expression that represents a "shift" of an `mddarray`. A `gshift_expr` generates an mddarray terminal expression. This expression is created by METAL's `GSHIFT` function. A shift of an `mddarray` conceptually returns a new array of the same shape, but with its elements rearranged into a new configuration. METAL supports only one type of shift where every element is moved to a new address that is at a fixed linear offset from the element's initial location. Boundaries are handled using the boundary function specified in the `mddarray`'s `glblshape` attribute.

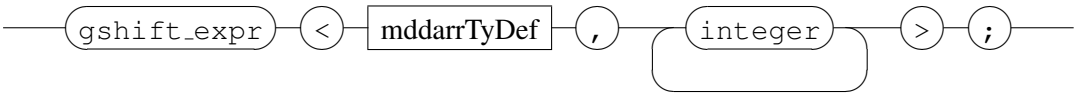
A `gshift_expr` does not actually return a new array. It is implemented as a linear indexing expression. `GSHIFT` requires a list of signed integer values. Each value in the list represents a constant offset of a *uniformly generated reference* (Gannon *et al.*, 1988; Wolf and Lam, 1991) for each dimension of the `mddarray`. The linearization of each reference produces the shifted address for an element.

§Syntax

gshiftOp



gshiftExprTy



§Rationale. A `gshift_expr` allows QUARC to retain a fully delinearized view of an `mddarray` access inside QOPT's IR. Having a delinearized view of an access makes it easier to perform several analyses used for code-generation. They are used for reuse distance calculation, and computation of communication sets when generating code for multi-node clusters. QOPT can also perform scalar redundancy elimination optimizations directly on `gshift_expr` objects.

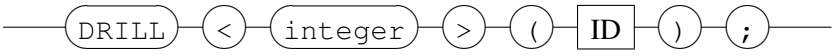
§**Implementation Note.** `GSHIFT` is implemented as a C++14 variadic function template that takes an **n**-tuple of non-type template arguments, where **n** equals the `mddarray` rank. This template generates a `gshift_expr` object defined with the same non-type template arguments.

DRILL expression (`drill_expr`)

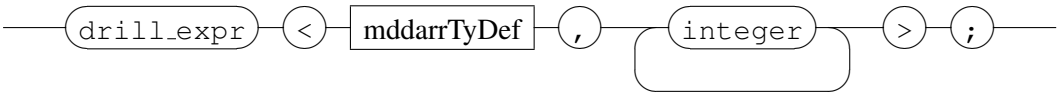
The `DRILL` operator generates a unary `drill_expr`. The expression abstracts an indirect access of a nested `sdlarray` element.

§Syntax

drillOp



drillExprTy



§Rationale. This expression class provides the option to write METAL array expressions that access a nested `sdlarray`, and pass the `sdlarray` element to an mkernel. This allows reusing the same mkernel across array expressions that use different types of `mddarray`.

§**Implementation Note.** The present implementation of the `DRILL` operator only allows drilling into an `mddarray` terminal expression. `DRILL` cannot be used on any other expression types. For example, `DRILL` does not allow `gshift_expr` or another `drill_expr`.

IF_EVEN_CHOOSE expression (`if_even_choose_expr`)

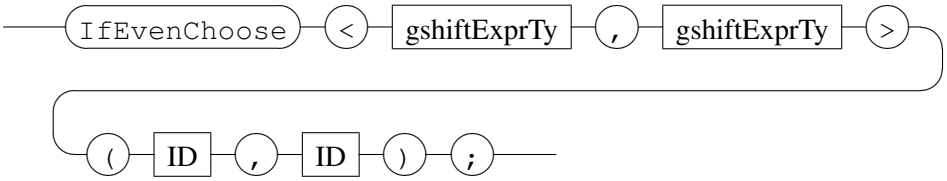
An `if_even_choose_expr` is a binary expression that encapsulates a selection operation for each `mddarray` elemental access. METAL's `IF_EVEN_CHOOSE` operator generates this type of expression. `IF_EVEN_CHOOSE` allows two alternate `GSHIFT` operations at each `mddarray` index position. A runtime selection between the two alternatives is made based on the `mddarray` index's "parity". Here parity refers to a domain-specific global index that is separate from the `mddarray's` internal indexing.

```
template <typename... Args>
bool is_even (Args... args) {
  size_t sum = 0;
  size_t arr[sizeof...(args)] = {(size_t)args...};
  for(auto i = 0ul; i < sizeof...(args)-1; ++i)
    sum += arr[i];
  return sum % 2 == 0;
}
```

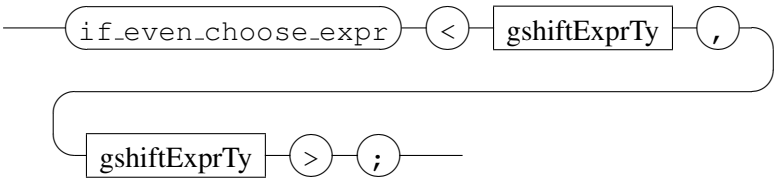
Listing 5.1: Implementation of an "even or odd" predicate function in C++

§Syntax

IfEvenChooseOp



IfEvenChooseExprTy



§Rationale. METAL's `IF_EVEN_CHOOSE` operator was designed specifically to help write "even-odd" preconditioned iterative solvers in lattice quantum chromodynamics (LQCD). The operation splits a multidimensional lattice or grid into sub-lattices, each of which contains either even or odd sites. Each sub-lattice is operated independently of the other. This makes it possible to parallelize iterative solvers by removing loop carried dependence.

§**Implementation Note.** Listing 5.1 shows a possible portable C++14 implementation of METAL's built-in "even-or-odd" predicate function. This is equivalent to the code QOPT generates.

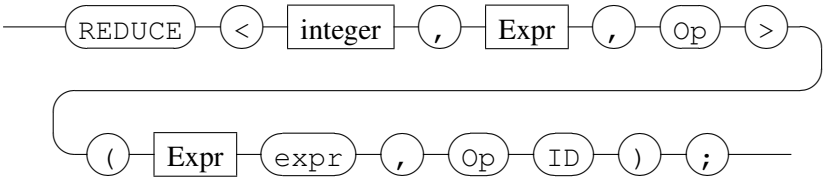
Reduction expression

The `REDUCE` operator constructs a reduction expression to reduce the elements of a METAL `mddarray` into a single scalar or `sdlarray` result. The `REDUCE` operator must be associative, but may also be marked as commutative. The `REDUCE` operator requires three template arguments. The first non-type integer argument conveys the commutative property of the reduction. A non-zero value

indicates the reduction is commutative, and a zero indicates the reduction is non-commutative. The second template argument should be a METAL expression that the `REDUCE` operator reduces. The expression argument can be any legal METAL expression except another reduction expression. The third template argument should be a function that specifies a domain-specific reduction function.

§Syntax

ReduceOp



§Rationale. Reduction operations are a fundamental part of scientific programming. These are used in most types of linear solvers. Providing an architecture neutral general-purpose reduction operator is needed to support such kernels in QUARC. The compiler has more opportunities for optimization if the operator is commutative.

5.1.2.6 Callback Functions

Boundary function

A boundary function is an indexing function defining the boundary condition of a METAL distributed array container (`mddarray`). EDSLs built using METAL specify the boundary functions for their domains. A boundary function adheres to the following function signature. The first argument is a signed integer signifying a *shifted* index value for an array dimension. The second argument is the extent of that dimension. The output of a boundary function is an index value used in memory address calculations.

§Syntax

boundFnDef

§Rationale. A differential equation system typically involves boundary value problems. A boundary value problem defines the value of the independent equation variables at the physical boundary of the

```
1 size_t PERIODIC (int64_t i, size_t extent)
2 {
3   return ((i %=extent) < 0 ) ?  i + extent : i;
4 }
```

Listing 5.2: Default implementation of periodic boundary conditions

domain. To handle boundary conditions scientific codes typically include conditional checks using loop-index variables. However, as an array programming language, METAL array expressions abstract away loops and direct array accesses. Instead, METAL includes the boundary function data type. The actual conditional checks are added during QOPT code-generation.

§**Implementation Note.** METAL's implementation presently limits an `mddarray` to a single boundary function for all array boundaries. During code-generation QOPT tries to inline all boundary function calls. To do so EDSLs must define the boundary function as `static` within the scope of a translation unit.

In our prototype EDSL implementation presented in Chapter 9 we used a periodic boundary condition. Listing 5.2 shows a possible portable C++ implementation of this function. For performance reasons this function is marked as a DSL intrinsic recognizable inside QOPT's IR. This helps QOPT easily inline this boundary function call with an equivalent version written in LLVM IR.

Reduction function

Reduction functions have the same signature and properties as binary mkernel functions. Refer Section 5.1.2.3.

5.2 Abstraction Characterization Templates (ACTs)

ACTs are standard compliant C++ function templates. As with any C++ template, ACTs use template metaprogramming to generate code at compile-time. However, unlike most C++ template metaprogramming techniques, the code generated by ACTs does not produce an executable. Instead of generating low-level code, such as loops and array accesses, an ACT generates a call to a specially annotated function. The specially annotated functions called from inside ACTs are called DSL intrinsic functions. This design pattern of using ACTs and DSL intrinsic functions allows encoding METAL array expressions as graph-based IR into QUARC's LLVM-based QOPT compiler with very little loss

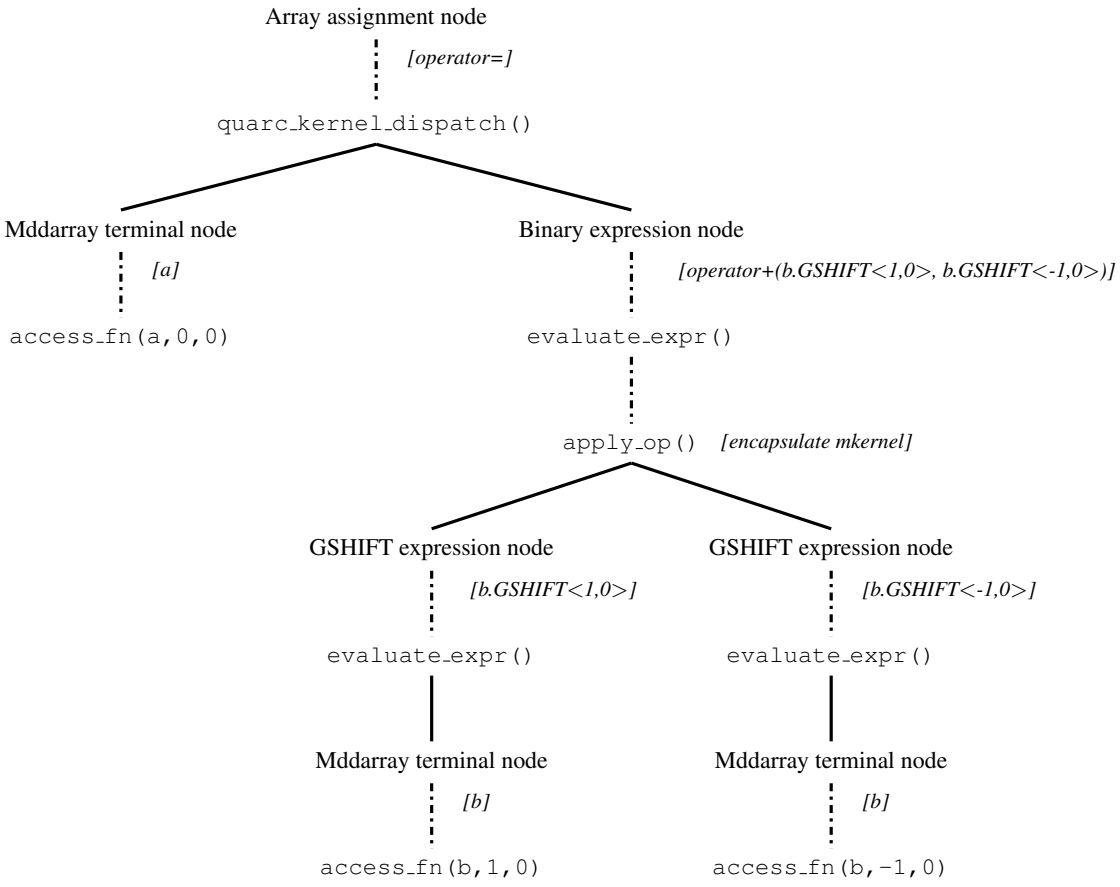


Figure 5.2: Binary expression tree for the expression a = b.**GSHIFT**<1,0>() + b.**GSHIFT**<-1,0>().

of high-level semantics. There is another advantage of ACTs when compared to using annotations and pragmas in high-level code. DSL intrinsic function calls are invisible to application programmers. No programmer intervention is needed to generate these calls. The complete set of annotations required by QUARC gets generated automatically using template metaprogramming.

Example 5.1.

Every ACT function template represents a node of a METAL parse/expression tree. This example shows the expression tree encoded using ACT function calls for a three-point stencil METAL array expression. Figure 5.2 presents this expression tree. The expression tree is constructed recursively in a bottom-up manner. Each ACT has an evaluation member function that in turn calls the evaluation member function of the child nodes of the ACT. Leaf or terminal nodes end the recursion. Terminals are either `mddarray` accesses or scalar accesses. This recursive evaluation of ACT nodes, happens *lazily*.

That is the whole expression tree is evaluated only when the result needs to be computed. In this case, the evaluation starts when the overloaded template assignment operator is instantiated.

Figure 5.2 shows the nested template instantiation hierarchy for the ACT and DSL intrinsic function templates. Every *dashed-dotted* arrow leads to a DSL intrinsic function call. Solid arrows represent parent-child relationship between nodes. All the function calls that are shown in the figure are DSL intrinsic function calls. The `quarc_kernel_dispatch` call indicates the start of the RHS sub-tree of expression. The `evalaute_expr` calls annotate the point of evaluation of each ACT expression node. The `apply_op` call indicates an elemental operation. The function encapsulates an mkernel function. Finally, `access_fn` indicates the terminal `mddarray` accesses. QOPT recognizes these DSL intrinsic function calls, and can recover the whole expression tree. Section 6.2.2 discusses that process.

Aside from annotating expression tree nodes, DSL intrinsic function calls serve an important secondary purpose. Some DSL intrinsic convey additional information that is useful during code-generation. The `apply_op` calls indirectly help QOPT identify mkernel function calls inside the generated IR. The function body for every `apply_op` function is empty except for a callback to the mkernel. This domain-specific information about the code structure allows QOPT to identify user provided mkernel functions. The arguments to an `access_fn` calls stores a complete delinearized view of that `mddarray` access. QOPT then parses the function call arguments to recover every delinearized `mddarray` access. The information is very useful in performing index calculations, reuse distance analysis, and other important code-generation steps.

5.2.1 Types of ACTs and DSL Intrinsic

METAL uses a relatively small number of DSL intrinsic functions to encode its expression trees into QOPT's IR. Table 5.1 lists all the DSL intrinsic functions that are used currently. These functions are specially annotated using Clang's `__attribute__((annotate("string")))` feature. The string value passed to this special macro serves as the key to recognize the functions inside the IR. This way it is ensured that the DSL intrinsic functions are recognizable regardless of their C++ mangled function names. The primary purpose served by each DSL intrinsic function call is to encode a type of METAL expression tree node. Example 5.1 introduced few of them, Table 5.1 describes the rest. Some DSL intrinsic function calls capture additional information about the high-level METAL program. Example 5.1 described the use of `apply_op` and `access_fn` functions. In addition, other DSL intrinsics serve

DSL intrinsic	Expression tree node	Arguments
access_fn	Denotes an mddarray access terminal node.	Constant offset for the affine access function in each array dimension.
apply_op	Denotes the call site of an mkernel function.	N/A
binary_expr_builder	Denotes creation of a binary expression node.	N/A
choose_expr_builder	Denotes creation of a IF_EVEN_CHOOSE expression node.	N/A
drill_expr_builder	Denotes creation of a DRILL expression node.	N/A
drill_op	Denotes a drill operation that is performed on evaluation of a DRILL expression.	Constant index value for a nested sdlarray dimension.
evaluate_expr	Denotes the evaluation point of a METAL expression node. It is emitted by all types of METAL expressions.	Constant index value for a nested sdlarray dimension.
gshift_expr_builder	Denotes creation of a GSHIFT expression node.	N/A
gshift_expr_builder	Denotes creation of a GSHIFT expression node.	N/A
quarc_kernel_dispatch	Denotes the start point for evaluating a METAL expression.	N/A
quarc_Rkernel_dispatch	Denotes the start point for evaluating a METAL reduction expression.	N/A
scalar_access_fn	Denotes a scalar terminal expression node	The actual scalar value encapsulated within the expression node.
scalar_expr_builder	Denotes creation of a scalar expression node.	N/A
sdlarray_copy	Denotes an assignment expression for sdlarray objects.	N/A
sdlarray_subop	Denotes an access to an sdlarray element.	N/A
unary_expr_builder	Denotes creation of a unary expression node	N/A

Table 5.1: METAL DSL intrinsic functions

similar secondary purposes. Nested `sdlarray` accesses are preserved by `sdlarray_subop`. The argument passed to `drill_op` intrinsic calls convey the index of an indirect access of `sdlarray` elements of an `mddarray`. The first argument to `reduction_kernel_dispatch` indicates if a reduction expression is commutative.

5.3 Programming Data-Placement Using ATL

Array Transformation Language (ATL) is a small specification language for programming data-placements for METAL `mddarrays`. This Section describes ATL, and aspects of its overall design. ATL is written as YAML (Oren Ben-Kiki, Clark Evans, Brian Ingerson, 2009), and each ATL entry is a YAML "key-value" pair. Each entry defines an `mddarray` data-placement attribute. These include the array's data-layout, data distribution, and other aspects corresponding to how the array is distributed over an MPI Cartesian communicator. An ATL file is required to initialize an `mddarray`, and gets parsed only at runtime.

5.3.1 ATL attributes

P-grid defines the processor space on to which the array partitions are mapped. In QUARC's current implementation the p-grid corresponds to an MPI Cartesian communicator. It is specified as a list of integers that specify the size of the communicator in each dimension.

Dist-rtf defines the blocking factors for each `mddarray` dimension. It too is specified as a list of integers. The operation encoded by this attribute represents a ρ transformation followed by a ϕ transformation that permutes all the block dimensions outwards. The resulting new shape of the array has a set of outermost block dimensions.

Example 5.2.

For a two-dimensional mddarray, $\mathbf{A}$, the dist-rtf value of $\{2, 2\}$ is equivalent to the following $\rho\phi$ transformation.

$$\rho(\mathbf{A}, \langle 2\ 2 \rangle) \rightarrow \phi(\mathbf{A}, \langle 0\ 2\ 1\ 3 \rangle).$$

Simd-rtf defines the blocking factors to build an innermost SIMD dimension by blocking one or more `mddarray` dimensions. The semantics of simd-rtf are like dist-rtf, except that the encoded $\rho\phi$ transformation is different. In this case, the ϕ transformation permutes all the block dimensions inwards.

```
using GS  = global_shape<2>;
using ATE = mddarray<T, GS>;

GS<2> gs(16,16);
ATE A(&gs, "atl-spec");
```

(a) Defining a two-dimensional mddarray.

```
{
  "p-grid"   : "2,2",
  "dist-rtf" : "2,2",
  "simd-rtf" : "2,2",
  "mapper"   : "STATIC"
}
```

(b) ATL specification as a YAML file

Figure 5.3: Example of an ATL specification.

The innermost block dimensions cumulatively show be equal to the SIMD register length for the target architecture. The transformation defines a new data-layout for the `mddarray`. The simd-rtf values can be user specified for cases where the user wants to generate code for a specific data-layout. Optionally, QOPT can speculatively generate a set of data-layout choices that are encoded as multiple simd-rtf values. QOPT then generates a code version for each of the data-layouts.

When an `mddarray` has nested `sdlarray` elements the effect of the $\rho\phi$ transformation gets applied to each nested `sdlarray` dimension. All the nested dimensions are permuted out, and the innermost dimension is still a SIMD vector dimension.

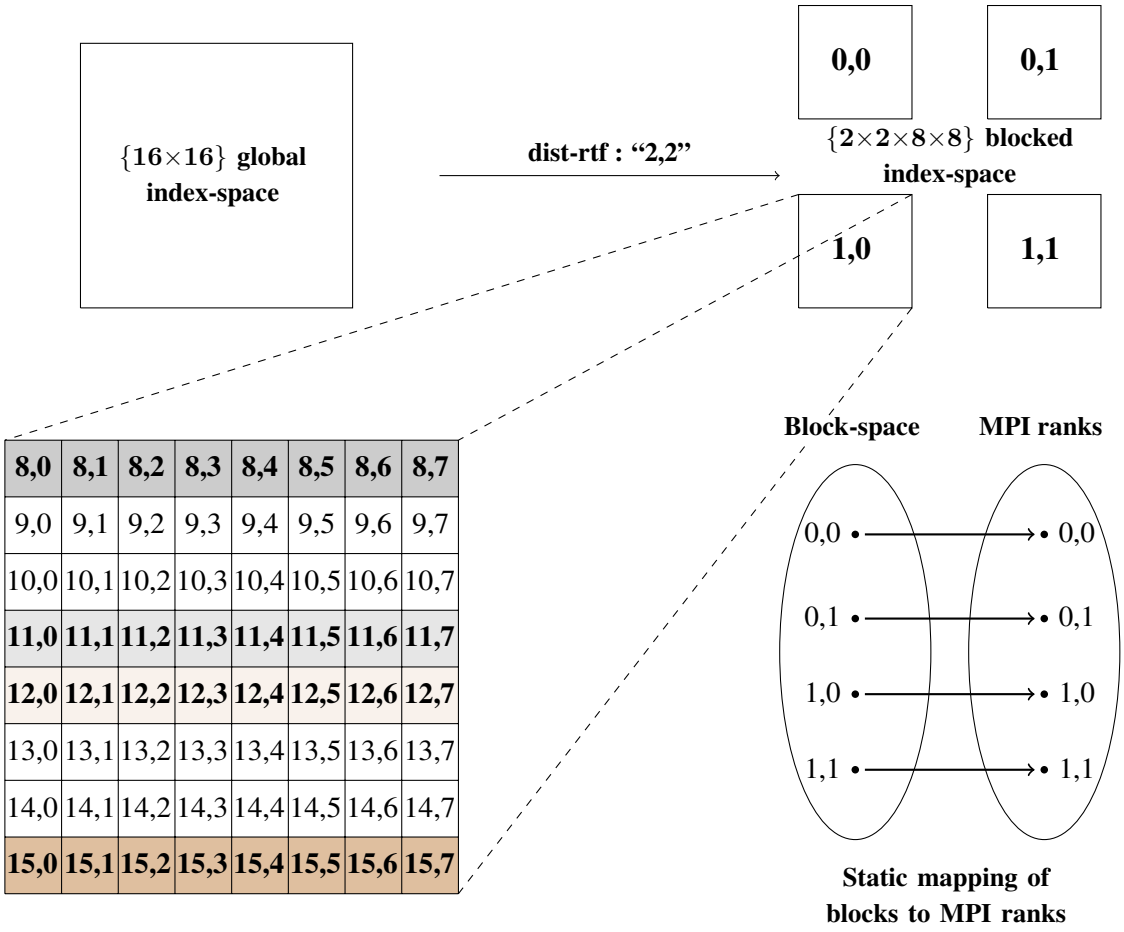
Example 5.3.

For a two-dimensional mddarray, $\mathbf{A}$, the simd-rtf value of $\{2, 2\}$ is equivalent to the following $\rho\phi$ transformation.

$$\rho(\mathbf{A}, \langle 2\ 2 \rangle) \rightarrow \phi(\mathbf{A}, \langle 1\ 3\ 0\ 2 \rangle).$$

Mapper specifies the mapping function that maps `mddarray` blocks on to the p-grid. The mapper value is defined as a string. QUARC-RT internally has a dictionary mapping the string name for a mapper to a library implementation of a mapping function.

§**Implementation Note.** The current scope of QUARC was limited to grid and lattice-based applications that exhibit regular data access patterns. Such applications benefit from rectilinear array partitioning to define "blocked" data-distributions. For this reason, currently, QUARC provides only one mapping function that statically maps `mddarray` blocks onto a single MPI rank within the p-grid. Future extensions can add more types of mapping functions for other use cases.



8,0	**8,1**	**8,2**	**8,3**	**8,4**	**8,5**	**8,6**	**8,7**
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7
10,0	10,1	10,2	10,3	10,4	10,5	10,6	10,7
11,0	**11,1**	**11,2**	**11,3**	**11,4**	**11,5**	**11,6**	**11,7**
12,0	**12,1**	**12,2**	**12,3**	**12,4**	**12,5**	**12,6**	**12,7**
13,0	13,1	13,2	13,3	13,4	13,5	13,6	13,7
14,0	14,1	14,2	14,3	14,4	14,5	14,6	14,7
15,0	**15,1**	**15,2**	**15,3**	**15,4**	**15,5**	**15,6**	**15,7**

Figure 5.4: ATL specifications to define a two-dimensional block distribution for an `mddarray`. The dist-rtf specification is added via ATL, and blocks the global index space into four blocks. These blocks are mapped bijectively to the ranks of an MPI Cartesian communicator.

5.3.2 METAL-ATL interface

The `mddarray` class constructor requires an ATL specification as an input parameter. The ATL specification defines the data-distribution and data-layout for the `mddarray`, and once defined these attributes are immutable. Listing 5.3a shows a simple example of an `mddarray` constructor call. Listing 5.3b shows the corresponding ATL specification file. This ATL specification is to define a two-dimensional $\{2\times2\}$ MPI Cartesian communicator. The dist-rtf, and simd-rtf attributes are the same as discussed in Example 5.2 and Example 5.3. Figure 5.4 provides a visualization of this data-distribution strategy. It only shows the block distribution over the MPI Cartesian communicator, and omits the data-layout transformation within each block.

5.3.3 Compile-time ATL v/s Runtime ATL

The initial design of QOPT incorporated ATL as a compile-time compiler flag. Along with data-placement options, ATL specified even the `mddarray` shapes. The design provided several code-generation advantages. Knowing an array's shape and placement at compilation allowed specializing array expression loops to compile time know trip counts. QOPT could avoid several extra checks that are required when the trip counts are not known at compile time. Additionally, some auxiliary variables needed to support data communication could be allocated statically on the stack without requiring dynamic heap allocation at runtime.

Despite the advantages, we found it hard to extend the compile-time ATL design to real-world applications. A fixed array shape and data-placement required a recompilation for each problem size. Moreover, integrating with existing application code required multiple hooks to pre-compiled binaries generated by QUARC. Another issue that proved hard to resolve was using different shaped array types in the same QUARC program. There was no easy way to annotate the array shape and data-placement to an array declaration with a single compiler flag. The closest solution was to add extra ATL annotations to METAL's source code. This was something we chose not to do to satisfy our design goal of a complete separation of domain-level algorithms from their architecture and parallel execution concerns. The final implementation of QUARC uses a runtime specification design for ATL due to the difficulties with a compile-time ATL design.

CHAPTER 6: CODE GENERATION AND RUNTIME SYSTEM

This chapter describes QUARC's compiler and runtime system. Section 6.1 introduces the overall compiler architecture, the pass pipeline, and the different intermediate representations (IRs) used during code-generation. Section 6.2 presents the high-level code-generation steps. Section 6.3 describes our speculative SIMD vectorization technique. Section 6.4 describes QUARC's scalarization steps for METAL array expressions. Each section provides the necessary implementation details of the set of compiler passes used in that stage of compilation. Section 6.5 presents QUARC's runtime system. The runtime is a lightweight library that uses integer set analysis to generate MPI communication. It also provides an API to define data-distributions for METAL `mddarrays`, and implements the interface for selecting a data-layout from the available options that were speculatively generated during compilation.

6.1 QOPT: QUARC's Domain-specific Compiler

QUARC Optimizer (QOPT) is a domain-specific extension to the open source LLVM (The LLVM Foundation, 2018) compiler framework, and is implemented as a plug-in to LLVM's optimizer and analyzer (Opt) module. METAL programs first get translated into LLVM IR without any optimization (`-O0`) using LLVM's C++ front-end (Clang). This initial `-O0` LLVM IR retains METAL array expression trees as encoded DSL intrinsic function calls (Section 5.2.1). QOPT's multi-stage code-generation process lowers the initial LLVM IR to successive domain-specific IRs. At the end of code-generation standard LLVM IR gets generated. Figure 6.1 shows QOPT's pass pipeline and the transition between the different IRs. The final residual LLVM IR is optimized further using Opt, and compiled into an executable.

Building a domain-specific compiler may seem orthogonal to an EDSL framework like QUARC. It may be argued that a domain-specific compiler introduces engineering complexity and maintenance cost that defeat the purpose of embedding a DSL in a general-purpose language. However, various important optimizations cannot be designed solely using metaprogramming-based EDSL techniques. Such optimizations require data flow and loop dependence-based compiler analysis. Moreover, EDSL

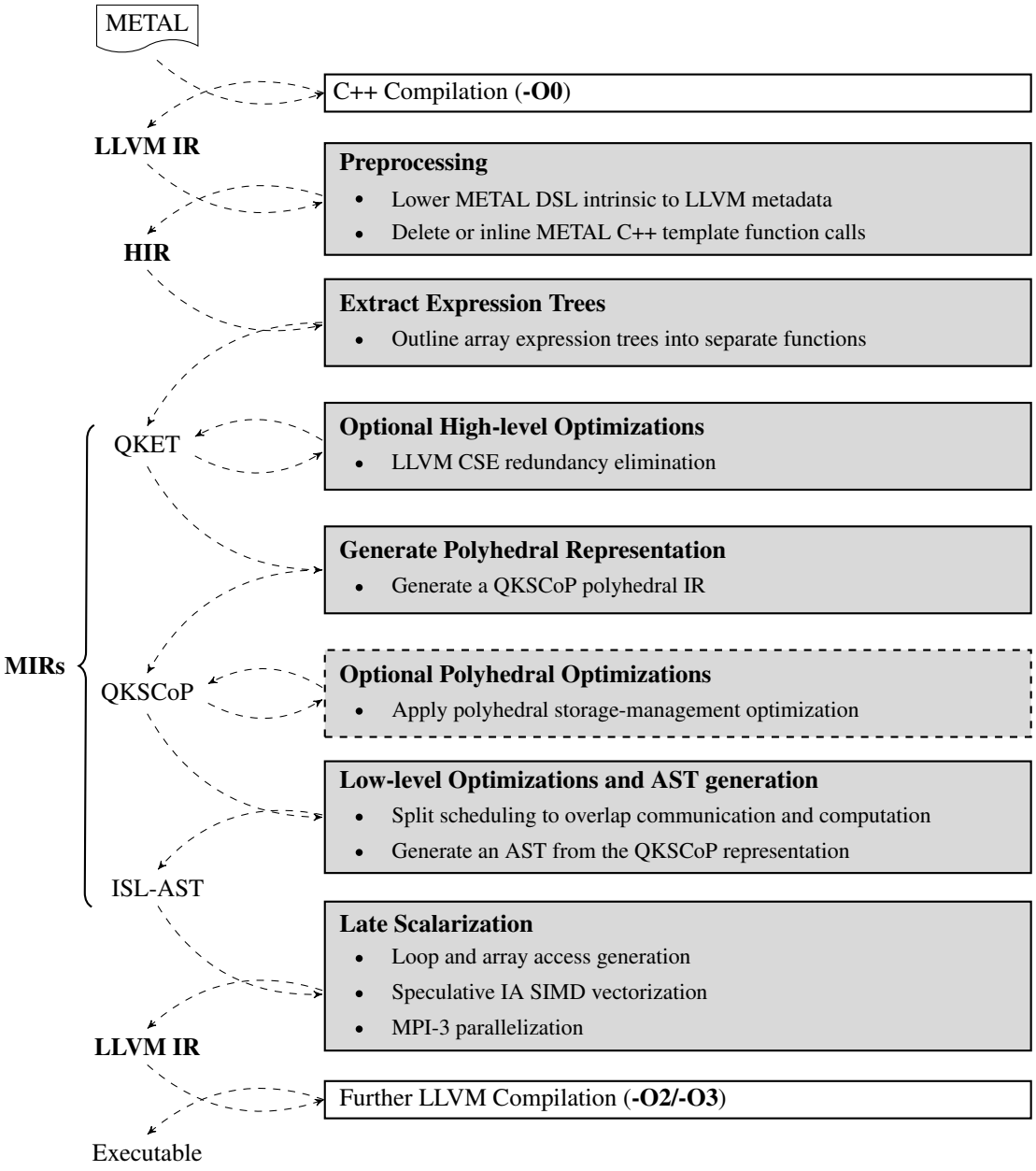


Figure 6.1: QOPT's compilation pipeline. The grey process boxes indicate QOPT compilation stages. The dashed "Optional Polyhedral Optimization" stage is a proposed QOPT step that is not presently implemented. The white boxes are standard Clang/LLVM compilation steps.

generated code often has obfuscations and library calls that impede compiler analyses and optimizations. Handling these scenarios require a domain-specific compiler. Several modern DSLs, such as the image processing DSL Halide (Ragan-Kelley *et al.*, 2013), and the numerical analysis DSLs Julia (Bezanson *et al.*, 2017) and Numba (Lam *et al.*, 2015), bundle their own compiler back-ends for this reason. All three languages use LLVM for low-level code generation, but have standalone optimization and analysis modules that do not utilize LLVM.

QOPT's approach is different from these other contemporary DSL frameworks. Our approach integrates EDSL code-generation and optimization closely into a general-purpose compiler. Doing this allows us to leverage compiler passes that the general-purpose compiler already provides. New domain-specific passes are also easier to implement on industry-standard static single assigned (SSA) (Rosen *et al.*, 1988) control flow graph (CFG) IR. SSA is a robust and easier format for code optimization and transformation. Most other EDSLs, like Halide, use custom non-SSA IRs. While, a non-SSA IR may seem expedient in designing a DSL, it limits extensibility and makes implementing data flow-based optimizations harder. EDSL compilers usually cannot interface with code outside EDSL expressions. This may in some scenarios limit the scope of code optimization. With its integrated EDSL compiler design, QUARC suffers from no such limitation.

Another advantage lies in the utilization of modern polyhedral code-generation and optimization techniques; many production compilers like LLVM, GCC (Free Software Foundation, 2018), and IBM's XLC (IBM Corporation, 2015) already provide the necessary boilerplate interface. Therefore, domain-specific polyhedral optimizations and code-generation require lesser engineering effort.

There are other secondary software engineering benefits of integrating an EDSL compiler into a general-purpose compiler. All code-generation and transformation phases are part of a single infrastructure. This removes the need for glue interfaces, and stitching together of different build technologies.

6.1.1 Architecture and Pass Pipeline

Figure 6.1 presents QOPT's compilation pipeline. As shown by this pipeline, QOPT compilation phases interpose Clang/LLVM compilation phases. When compiling a METAL program all QOPT passes complete first, and only then user-specified LLVM compilation options such as `-O2` or `-O3` execute. The pass pipeline does not preclude QOPT from invoking standard LLVM analysis and transformation passes,

and various stages of QOPT internally utilize LLVM passes. This is one of the key engineering benefits of implementing QOPT as a plug-in for LLVM Opt.

Table 6.1 lists all passes currently provided by QOPT. The QOPT analysis passes are listed in Table 6.1(a). Analysis passes do not alter the IR. They only extract high-level METAL language properties encoded inside the LLVM IR. Table 6.1(b) lists the QOPT transformation passes. Both type of QOPT passes depend on an initial set of preprocessing passes.

QOPT code-generation uses multiple forms of IRs. The initial Clang `-O0` compilation results in an LLVM IR that retains all METAL ACTs and DSL intrinsic (Section 5.2.1) function calls. The DSL intrinsic function calls are annotated by METAL. All the annotations are stored in the `-O0` IR as a global string constant. This global string is composed of sub-strings that are key-value pairs of the METAL annotation strings and the annotated functions. QOPT performs an initial preprocessing step to convert the `-O0` IR into a form that is termed as QOPT's High-level IR (HIR).

High-level Intermediate Representation. The QOPT preprocessing pass converts METAL annotation strings into LLVM IR metadata nodes. This ensures that METAL annotations persist across different QOPT transformation passes. Preprocessing also invokes LLVM's `mem2reg` pass to promote memory load and store instructions to virtual registers. This step constructs the pruned static single assignment (SSA) IR that is used by all subsequent passes. After constructing the pruned SSA form the preprocessing pass does domain-specific inlining of METAL function calls. Domain-specific inlining has the effect of pruning METAL expression trees, and removing all intermediate ACT function calls. The LLVM IR constructed after preprocessing is called the HIR. Section 6.2.2 describes QOPT's preprocessing stage in detail.

Mid-level Intermediate Representations. QOPT transformation and analysis passes construct four IR forms out of the HIR. These mid-level Intermediate Representations (MIRs) are separate from the standard LLVM IR, and represented as in-memory data structures. QOPT uses the following four types of MIRs.

- **QUARC Kernel Expression Tree (QKET)**

  QKET is a binary expression tree format to represent a METAL array expression. A QKET is rooted at an "assignment" node represented by an `sdlarray_copy` DSL intrinsic call or an LLVM *store* instruction. All internal nodes are METAL DSL intrinsic calls that represent mkernel

Pass name	Stage	Dependency	Description
qopt-detectqket	All	qopt-preprocess	It detects all QKs in a function. For each QK it builds a binary expression tree (QKET) using recursive *def-use* analysis of METAL DSL intrinsic function calls.
qopt-mka	Late Scalarization	qopt-preprocess	It analyzes METAL array element-wise functions (mkernels) to generate metadata that is used by qopt-code to inline mkernels.

(a) QOPT Analysis Passes

Pass name	Stage	Dependency	Description
qopt-preprocess	Preprocessing	mem2reg[a]	It applies domain-specific function inlining to METAL expression trees. Also, converts C++ annotations to LLVM IR metadata nodes.
qopt-extractqket	High-level Opts	qopt-preprocess, qopt-detectqket	It outlines QKETs into separate functions to enable high-level optimization.
qopt-inlineqket	High-level Opts	qopt-preprocess, qopt-detectqket	It is a custom inliner for QKETs that were outlined in separate functions.
qopt-simplifyqket	High-level Opts	qopt-preprocess, qopt-detectqkets, qopt-extractqket	Optional pass that applies high-level optimization to outlined QKET functions. High-level optimizations are either QKET rewrites, or LLVM scalar redundancy elimination applied to QKET nodes.
qopt-codegen	Late Scalarization	qopt-preprocess, qopt-detectqket, qopt-mka	This pass scalarizes QKs. It generates multiple versions of IA SIMD loops if ATL data-layout specifications were provided, inlines all METAL mkernel calls, and adds QUARC-RT library calls for MPI parallelization.

[a] mem2reg is LLVM's pruned-SSA form generation pass.

(b) QOPT Transformation Passes

Table 6.1: QOPT analysis and code generation passes

calls or whole array operations, *i.e.*, `DRILL`, `CHOOSE`, `REDUCE` (Section 5.1.2.5). Leaf nodes are always `mddarray` or `scalar` accesses. Section 6.2.3 formally defines the structure of a QKET and describe the steps of building a QKET. A QKET representation stores an internal attribute to indicate if the QKET is a reduction expression.

- **QUARC Kernel Expression Forest (QKEF)**

  A QKEF is a disjoint union of multiple QKETs. QKEFs are produced by high-level optimization of the HIR that fuses individual QKETs. Section 6.2.4 defines the rules guiding construction of a QKEF.

- **QUARC Kernel Static Control Part (QKSCoP)**

  QOPT uses polyhedral code-generation to scalarize METAL array expressions. It uses a polyhedral representation called QKSCoP for that purpose. A QKSCoP is constructed for every QKET or QKEF. The QKSCoP form is based on the static-control-parts (SCoP) format used by the Integer Set Library (ISL) (Verdoolaege, 2010). A SCoP is a control flow graph (CFG) region that has statically known branching and memory accesses. That is, a SCoP is generally a CFG region with only for-loops and if-conditions. Although, certain relaxation of this condition is possible (Benabderrahmane *et al.*, 2010). We refer readers to (Grosser, 2011) for further details LLVM's SCoP representation.

  QKSCoP is different in its construction than the standard SCoP format. As a QKSCoP corresponds to a QKET or QKEF, it does not directly meet the definition of a SCoP. Instead, a QKSCoP can be conceptually understood as an "abstract" CFG region that corresponds to the loop-nest and conditional branches abstracted by a METAL array expression.

  QOPT's present implementation does not include non-QKET control flow structures inside a QKSCoP. This limits some optimizations opportunities. This work is proposed as a future extension to QOPT.

- **Integer Set Library Abstract Syntax Tree (ISL-AST)**

  ISL is part of LLVM's polyhedral code generator and optimization module Polly. ISL analyzes code in the SCoP format, and produces an abstract syntax-tree (AST) representation of each SCoP.

```
Procedure codegen(Module M)
   Input: LLVM -O0 IR Module M
   Output: Fully code generated Module M′
   Parameter : boolean HLO, layout-choices
 1 if M has no METAL DSL intrinsics calls;
 2  then
 3 |  exit;
 4 end
   // preprocess and convert METAL annotations to LLVM metadata
 5 M′← preprocess(M);
 6 if HLO then
   |  // outline expression trees into separate functions
 7 |  M′← extractQkets(M′);
   |  // apply LLVM's scalar redundancy elimination passes to M'
 8 |  M′← qoptHloOpts(M′);
 9 end
10 foreach function F in M′ that is a QK do
   |  // Lower QKs into loops, add MPI calls.
   |  // If layout choices are provided, then generate a SIMD code version
   |  // for every QK for each layout choice
11 |  lateScalarizarion (F, layout-choices);
12 end
13 return M′;
```

Figure 6.2: Codegen shows at a high-level QOPT's code generation process

This AST format is called as the ISL-AST IR. QOPT uses ISL to convert QKSCoP to ISL-AST. The ISL-AST is then lowered to standard LLVM.

Code-Generation. Figure 6.2 presents *codegen*, QOPT's overall code-generation procedure. Each sub-procedure called from *codegen* is discussed in detail over the next subsections. The *codegen* procedure acts on an LLVM module or translation unit. *Codegen* is parameterized by two optional arguments: *HLO* and *layout-choices*. HLO triggers high-level optimizations such as redundancy elimination and QKET transformations. The layout choices are ATL simd-rtfs specifications that define data-layout choices. QOPT does SIMD vectorization only when layout choices are specified. *Codegen* involves two main steps. The first high-level code-generation step does initial preprocessing of the `-O0` IR and optional high-level optimizations. The next step, called late scalarization, does all loop and array access generation. After late scalarization the generated IR is handed off to LLVM's standard optimization pathway for further optimization and machine code-generation.

```
using qcomplex = quarc::metal::sdlarray<2,float>;
using su3      = quarc::metal::sdlarray<3,qcomplex>;
using GS2D     = quarc::metal::global_shape<2>;
using SU32DArr = quarc::metal::mddarray<su3, GS2D>;
/// Mkernel adding two su3 sdlarrays. The addition loop has been fully unrolled.
auto su3add(su3 s1, su3 s2) {
  su3 ret;
  ret[0][0] = s1[0][0] + s2[0][0];
  ret[0][1] = s1[0][1] + s2[0][1];
  ret[0][0] = s1[0][0] + s2[0][0];
  ret[0][1] = s1[0][1] + s2[0][1];
  ret[0][0] = s1[0][0] + s2[0][0];
  ret[0][1] = s1[0][1] + s2[0][1];
  return ret;
}
/// Builds a binary expression encapsulating su3add. binary_expr_builder is a
/// METAL DSL intrinsic that indicates a binary array expression.
template < typename Tp1, typename Tp2 >
const auto& operator+ (const Tp1 & ref1, const Tp2 & ref2) {
  return
  quarc::metal::expression_factory::binary_expr_builder<
    Tp1, Tp2, typename Tp2::value_type, su3add
  >(ref1, ref2);
}
/// A 2D stencil array expression
void twoDStencilQK (const SU32DArr &a1, SU32DArr &a2) {
 a2 = a1.GSHIFT<1,0>() + a1.GSHIFT<-1,0>()
    + a1.GSHIFT<0,1>() + a1.GSHIFT<0,-1>();
}
```

Listing 6.1: A two-dimensional five-point stencil using SU3 vector types

6.1.2 Running Example

Listing 6.1 presents a two-dimensional stencil written in METAL. This is a running example used to elaborate the steps of *codegen*. The stencil uses a relatively simple mkernel function. The `su3add` mkernel adds two `su3` data type, and returns the result. The `su3` data type denotes a mathematical vector object belonging to the special unitary group of degree three $SU(3)$. $SU(3)$ algebra is the basic algebra used in LQCD. Using `su3` data types also illustrate code-generation involving nested arrays. All IR Listings in this Section use an abridged form of the standard LLVM IR. For space and readability reasons the examples omit most type signatures, replace mangled C++ function names by readable pseudonyms that are analogous to the C++ names, and do not include LLVM specific attributes and annotations that are present in the full LLVM IR. In all Listings, both METAL and IR, QUARC-specific identifiers are emphasized using **boldface** font.

```
; -O0 IR for the operator+ function
define internal %"struct.binary_expr"* @operator_add(%ref1, %ref2) {
entry:
  %ref1.addr = alloca %"struct.binary_expr"*
  %ref2.addr = alloca %"struct.gshift_expr"*
  store %"struct.binary_expr"* %ref1, %"struct.binary_expr"** %ref1.addr
  store %"struct.gshift_expr"* %ref2, %"struct.gshift_expr"** %ref2.addr
  %0    = load %"struct.binary_expr"*, %"struct.binary_expr"** %ref1.addr
  %1    = load %"struct.gshift_expr"*, %"struct.gshift_expr"** %ref2.addr
  %call = call %"struct.binary_expr"* @binary_expr_builder(%0, %1)
  ret %"struct.binary_expr"* %call
}
; -O0 IR for one of the GSHIFT operator calls
define internal %"struct.gshift_expr"* @GSHIFT(%"struct.mddarray"* %a) {
entry:
  %a.addr = alloca %"struct.mddarray"*, align 8
  store %"struct.mddarray"* %a, %"struct.mddarray"** %a.addr, align 8
  %a1 = load %"struct.mddarray"*, %"struct.mddarray"** %a.addr, align 8
  %call = call %"struct.gshift_expr"* @gshift_expr_builder(%a1)
  ret %"struct.gshift_expr"* %call
}
; -O0 IR for twoDStencilQK
define internal void @twoDStencilQK(%a1, %a2) {
entry:
  %a1.addr = alloca %"struct.mddarray"*
  %a2.addr = alloca %"struct.mddarray"*
  store %"struct.mddarray"* %a1, %"struct.mddarray"** %a1.addr
  store %"struct.mddarray"* %a2, %"struct.mddarray"** %a2.addr
  %0     = load %"struct.mddarray"*, %"struct.mddarray"** %a1.addr
  %call  = call %"struct.gshift_expr"* @GSHIFT(%0)
  %1     = load %"struct.mddarray"*, %"struct.mddarray"** %a1.addr
  %call1 = call %"struct.gshift_expr"* @GSHIFT(%1)
  %call2 = call %"struct.binary_expr"* @operator_add(%call, %call1)
  %2     = load %"struct.mddarray"*, %"struct.mddarray"** %a1.addr
  %call3 = call %"struct.gshift_expr"* @GSHIFT(%2)
  %call4 = call %"struct.binary_expr"* @operator_add(%call2, %call3)
  %3     = load %"struct.mddarray"*, %"struct.mddarray"** %a1.addr
  %call5 = call %"struct.gshift_expr"* @GSHIFT(%3)
  %call6 = call %"struct.binary_expr"* @operator_add (%call4, %call5)
  %4     = load %"struct.mddarray"*, %"struct.mddarray"** %a2.addr
  %call7 = call %"struct.mddarray"* @sdlarray_copy(%4, call6)
  ret void
}
```

Listing 6.2: -O0 IR generated by Clang from the METAL source.

6.2 QOPT High-level Code-generation

6.2.1 Clang `-O0` compilation

Listing 6.2 shows the `-O0` IR for the `twoDStencilQK`, `GSHIFT`, and `operator+` functions. The `-O0` IR is very close to the high-level METAL source. It retains all high-level METAL ACT calls and DSL intrinsic calls. Section 5.2 introduced ACTs as a metaprogramming technique to encode

METAL's array expression trees into LLVM IR. This example uses three ACTs, `GSHIFT`, `operator+`, and `operator=`. The four `GSHIFT` ACT calls of the original program (Listing 6.1) are compiled to the four function calls on lines 30, 32, 35, and 38 of the `-O0` IR. A `GSHIFT` ACT inserts a *gshift expression* node into the METAL expression tree. A gshift-expression node is detected using the `gshift_expr_builder` DSL intrinsic function call. Line 14 of Listing 6.2 shows the generated code for one of the `GSHIFT` ACTs, and line 19 is the DSL intrinsic function call.

The overloaded `operator+` functions in Listing 6.1 build *binary expression* nodes. These get compiled into the `operator_add` calls on lines 33, 36 and 39. These also follow the same design pattern. Line 2 shows the generated code for one of the `operator+` (`operator_add`) functions. This ACT emits the `binary_expr_builder` DSL intrinsic call shown on line 10.

The `operator=` ACT generates an *assignment expression* node. In this case, the assignment expression calls the copy constructor defined inside METAL's `sdlarray` class. This ACT function is annotated inside METAL's code with Clang's `__attribute__((always_inline))` attribute. This caused it to get fully inlined even during `-O0` compilation. Therefore, the METAL DSL intrinsic function `sdlarray_copy` is directly called on line 41 of Listing 6.2.

Listing 6.2 does not show all the function calls that are part of a complete METAL expression tree. Each expression node has a child *evaluation* node. An evaluation node is encoded by an `eval_expr` DSL intrinsic call. An evaluation node, depending on the type of expression, may be the parent of other evaluation nodes, mkernel wrapper nodes, array accesses, or array assignments. Each child node is encoded with a different DSL intrinsic call (Section 5.2.1).

6.2.2 Preprocessing

The `-O0` IR retains the complete METAL expression tree as outlined ACT and DSL intrinsic function calls. The first step of QOPT's high-level code generation, `preprocess`, prunes the expression tree by domain-specific inlining several of these functions. It also converts METAL's C++ annotations into LLVM metadata nodes. Figure 6.3 presents the steps of the `preprocess` procedure.

The IR generated by `preprocess` is called HIR. This HIR does not have any outlined ACT calls, but some DSL intrinsic calls are still retained. Listing 6.3 shows the preprocessed HIR for Listing 6.1's `twoDStencilQK` function. The example shows the inlined `quarc_access_fn` DSL intrinsic calls that were encapsulated by `GSHIFT` ACT calls in METAL. The parameters of these calls give the shift

Procedure preprocess(Module M)

Input: METAL Module M
Output: HIR Module M′

```
1  M′← M;
2  foreach function F in M′ that has a METAL annotation do
3  |  convert METAL annotations to LLVM string metadata;
4  |  add the string metadata to F as an LLVM IR metadata node (MDNode);
5  end
   // Identify and annotate mkernels
6  foreach function F in M′ that is an apply_op DSL intrinsic;
7   do
8  |  add ALWAYS_INLINE attribute to F;
9  |  assert that F only has a single function call instruction;
10 |  add an MDNode to the called function identifying it as an mkernel;
11 end
   // Domain-specific inlining of METAL ACT calls
12 foreach function F in M′ that has an EVAL_EXPR metadata;
13  do
14 |  CalledFunctions ← get list of all functions called by F;
15 |  foreach CF in CalledFunctions do
16 |  |  if CF is not access_fn, drill_op, if_even_choose then
17 |  |  |  CF ← add always_inline attribute to CF;
18 |  |  end
19 |  end
20 end
21 return M′;
```

Figure 6.3: Preprocess converts C++ annotations into LLVM metadata nodes, and does domain-specific inlining of METAL expression trees to simplify future analysis steps.

offsets that were originally passed to GSHIFT. The su3add mkernel calls, previously encapsulated by binary expression nodes, are also now inlined.

Preprocess first identifies all METAL evaluation nodes. Evaluation nodes are ACT function calls that denote the evaluation of a METAL expression. They are encoded by an eval_expr DSL intrinsic call. *Preprocess* recursively inlines all functions called from inside an evaluation node. The only exclusions are the DSL intrinsic calls encoding mddarray accesses, DRILL, and IF_EVEN_CHOOSE expressions.

Domain-specific inlining of METAL's outlined expression trees is done for two main reasons. This makes subsequent analysis and code generation simpler. All such steps only require local data flow analysis rather than interprocedural analysis. Inlining also opens the possibility of applying high-

```
; preprocessed HIR for twoDStencilQK
define void @twoDStencilQK(%a1, %a2) {
entry:
  %sret18 = alloca %"struct.sdlarray"
  %sret17 = alloca %"struct.sdlarray"
  %sret1  = alloca %"struct.sdlarray"
  %0 = call %"struct.sdlarray"* @access_fn(%a1,  1,  0)
  %1 = call %"struct.sdlarray"* @access_fn(%a1, -1,  0)
  call void @su3add(%sret1, %0, %1)
  %2 = call %"struct.sdlarray"* @access_fn(%a1,  0,  1)
  call void @su3add(%sret17, %sret1, %2)
  %3 = call %"struct.sdlarray"* @access_fn(%a1,  0, -1)
  call void @su3add(%sret18, %sret17, %3)
  %4 = call %"struct.sdlarray"* @access_fn(%a2,  0,  0)
  %5 = call %"struct.sdlarray"* @sdlarray_copy(%4, %sret18)
  ret void
}
```

Listing 6.3: HIR generated by preprocessing the -O0 IR

level optimizations on the expression tree nodes. Such optimizations involve either scalar redundancy elimination, or domain-specific transformation of the expression tree.

Preprocess does not directly inline functions. Instead, it adds LLVM's `ALWAYS_INLINE` function attribute to all functions that are to be inlined. After that QOPT runs LLVM's *always-inline* function inlining pass.

Along with inlining, `preprocess` also converts METAL's C++ annotations into LLVM IR's CFG nodes. LLVM provides special CFG nodes called `MDNode` for this purpose. METAL's C++ annotations are lowered into the `-O0` IR as a global string variable. This global variable has key-value entries for the C++ annotation string and the function name on which the annotation was applied. *Preprocess* parses this global string variable to extract the entries. It then converts them into corresponding LLVM metadata nodes that are attached to the LLVM function definitions. Converting from C++ annotations to LLVM metadata eases further code generation. It also removes a level of indirection introduced by *wrapper* DSL intrinsic calls (Section 5.2.1). The wrapper DSL intrinsic call, `apply_op`, encapsulates mkernels that are user-defined and cannot be directly annotated by METAL. *Preprocess* identifies the `apply_op` calls, inlines them, and directly adds MDNodes to the mkernel functions. Doing this ensures QOPT can identify user provided mkernel functions that are defined outside of METAL.

6.2.3 QKET Construction

All optimization and code generation stages use the QKET binary expression tree MIR. QOPT uses a procedure called *qketgen* to generate a QKET from LLVM IR. Prior to describing the steps in *qketgen*, we formalize the definition of a QUARC Kernel (QK) and a Reduce QUARC Kernel (RQK).

Definition 6.1. QUARC Kernel (QK)

A QK is a whole array assignment statement whose LHS is an `mddarray` access expression with no shifts. The RHS sub-expression can be any METAL array expression, but not a reduction expression. All `mddarray` values in a QK should have the same global shape and data placement.

Definition 6.2. Reduction QUARC Kernel (RQK)

An RQK is a METAL assignment statement where the LHS is a scalar or `sdlarray` variable, and the RHS sub-expression is a reduction expression.

Qketgen is implemented inside QOPT's `qopt-detectqket` analysis pass. This pass is a basic block level pass, *i.e.*, its scope is restricted to a single basic block inside a function. A basic block is a maximal length sequence of branch-free instructions within a function. *Qketgen* uses recursive *def-use* graph analysis to build the QKET. A def-use graph is a graph that contains an edge from each definition point in a program to every possible use of the variable at runtime (Kennedy and Allen, 2002).

Qketgen builds this QKET in a bottom-up fashion. It starts by identifying the leaf nodes, *i.e.*, `mddarray` or scalar access nodes inside a basic block. The LLVM instructions denoting leaf nodes would have the `access_fn` LLVM MDNode metadata. After identifying a leaf node, it uses the def-use graph to identify the next instruction that uses the leaf node. Typically, this would be either an mkernel, `DRILL`, `IF_EVEN_CHOOSE`, or an assignment node. *Qketgen* repeats the def-use analysis after reaching the user of leaf nodes. The recursion terminates on finding the root node of the QKET that is always an assignment operation. For QKs that only have `mddarrays` with scalar elements, the root is an LLVM store instruction. If the `mddarrays` used nested `sdlarray` members the root node of the QK is an `sdlarray` copy constructor call. This structure is guaranteed by METAL. *Qketgen* exits once both sub-trees of the root node are constructed.

6.2.4 High-level Optimizations

High-level optimization of QKs tries to eliminate redundancy, and potentially fuse QKs. The goal is to potentially fuse array expressions that access the same memory location. There are two strategies for high-level optimization of QKs. LLVM scalar redundancy elimination using value numbering can identify fusion opportunities for simpler cases. Polyhedral dependence analysis can help identify fusion cases for more complicated cases. QOPT's prototype implementation only implements the first strategy. We propose a design for extending QOPT for the polyhedral strategy.

The *qketfusion* procedure implements a local optimization, *i.e.*, the scope is limited to a basic block. The procedure starts by identifying QKs that can be potentially fused. The decisions rests on the following two constraints:

Constraint 6.1. Currently, only QKs that are adjacent and access at least one common `mddarray` reference are candidates for fusion.

Constraint 6.2. An LHS array reference for any QK in the set of adjacent QKs can only be accessed in any of the RHS *iff* that arguments to the RHS `access_fn` call are all zeroes.

Qketfusion only looks to fuse QKs, and RQKs are not considered. Two QKs are considered adjacent if the end instruction of the first QK's QKET is immediately followed by the start instruction of the second QK's QKET. QOPT ignores any debug or LLVM intrinsic instructions when identifying adjacent QKs. QOPT uses only value tracking to check if two adjacent QK share at least one array reference. Thus, any kind of pointer-based indirection prevents fusion. If two adjacent QKs meet Constraint (6.1), then they are evaluated against Constraint (6.2). This constraint ensures that QK fusion does not introduce a loop carried dependence. This is a very broad check that may preclude legitimate fusion. Such fusion cases cannot be handled with data dependence-based analysis. The future extension proposal to enhance QOPT using polyhedral data dependence analysis addresses this issue.

Once a candidate set of QKs is identified, *qketfusion* outlines the set of QKETs into a separate function. Outlining is done to restrict the scope of scalar redundancy elimination to only the candidate set of QKETs inside one basic block. The outlined function is optimized using LLVM's global value numbering (GVN) redundancy elimination pass. After running GVN, *qketfusion* invokes a slightly modified version of the QKET generation procedure. The procedure called *qkefgen* works the same way,

```
b = DRILL<0>(g) * a.GSHIFT<1>() + DRILL<1>(g)*a.GSHIFT<-1>();
d = DRILL<0>(g) * c.GSHIFT<1>() + DRILL<1>(g)*c.GSHIFT<-1>();
```

(a) Shared DRILL expressions across two QK.

```
a  = b.GSHIFT< 1 , 0>();
a += b.GSHIFT<-1 , 0>();
a += b.GSHIFT< 0 , 1>();
a += b.GSHIFT< 0 ,-1>();
```

(b) Multiple add-assignment expressions to write a five-point stencil.

Figure 6.4: METAL array expressions fusible using *qketfusion*

```
using GS2D       = quarc::metl::global_shape<2>;
using floatArr2D = quarc::metl::mddarray<float, GS2D>;

void unNormalizedBoxFilter (const floatArr2D &a1, floatArr2D &a2) {
 auto a3 = a1.GSHIFT<-1,0>() + a1 + a1.GSHIFT<1,0>();
 a2      = a3.GSHIFT<0,-1>() + a3 + a3.GSHIFT<0,1>();
}
```

Listing 6.4: An unnormalized box filter kernel from 2D image processing. These two QKs cannot be fused using GVN-based redundancy elimination.

but instead of constructing a single QKET generates multiple QKETs each represented inside a QKEF. Note that in a QKEF there are multiple QKETs, and one or more of these QKETs share common nodes.

Listing 6.4a and Listing 6.4b are two examples where *qketfusion* can use GVN to fuse the QKs. In both cases, GVN would identify the redundant values across multiple QKs, and replace those values with a single value. Listing 6.4a is an excerpt from a multiple RHS linear solver kernel. GVN identifies the two `DRILL<0>(g)` and the two `DRILL<1>(g)` accesses that are common across both QKs. Listing 6.4b is equivalent to the five-point stencil kernel from our running example shown in Listing 6.1. Instead of writing the whole stencil as a single statement, multiple add-assignment operators are used to break it out into multiple statements.

§**Implementation Note.** As currently implemented, *qketfusion* cannot identify some potential fusion candidates. Listing 6.4 shows such an example. For this example, *qketfusion* identifies the two QKs as potential fusion candidates. However, the QKs do not share any exact array reference, and GVN is unable to locate any redundancy. *Qkefgen* fails to build a QKEF for the same reason, and the two QKETs are deemed non-fusible. The QKs are in fact fusible using a technique known as array storage optimization.

The optimization uses dependence analysis to identify that the array `a3` can in fact be replaced by a temporary. Doing so then enables fusing these two QKs. This is an important optimization that is especially useful in image processing pipelines. The Halide image processing DSL implements this type of fusion optimization. The application of the optimization is dependent on external explicit specification of the fusion, and Halide does not provide an analysis framework for auto-discovery. Bhaskaracharya *et al*. (Bhaskaracharya *et al.*, 2016) do provide an automated polyhedral method to discover and apply this type of fusion.

Apart from this array storage management example, most other QK fusion cases fall under classical loop fusion. Modern polyhedral data dependence analysis, such as the one provided by ISL, allow identifying such cases. QOPT already integrates ISL in its compiler infrastructure. The current usage is restricted only to code generation out of QKSCoPs and MPI communication generation. To benefit from ISL's data dependence analysis, we would have to expand QKSCoPs to encompass multiple QKs. Doing that would allow an inter QK dependence analysis, and leading to discovery of additional fusion and parallelization opportunities.

6.3 Speculative SIMD Vectorization

This section presents QOPT's speculative SIMD code generation method. Here we only discuss the rationale for using a speculative strategy, and how the interface is designed. The actual SIMD vectorization is described under the late scalarization process in Section 6.4.

Large number of potential data-layout candidates

QOPT's SIMD vectorizer is designed to generate SIMD code for a particular memory data-layout that was specified using a $\rho\phi$ transformation. Section 5.3.1 described the process for specifying data-layouts for an `mddarray`. The number of possible data-layouts depends on the shape of the `mddarray`, and the architectural SIMD register width. For higher dimensional arrays, this can be a large number. It is equivalent to finding all multiplicative partitions for the vector register width, and then identifying all the permutations to factorize the `mddarray` dimensions to build each multiplicative partition. The following example illustrates this for a four-dimensional `mddarray` and an architecture with SIMD register width of eight.

Example 6.1.

1. There are three multiplicative partitions for the number eight. These are $\{8\}$, $\{2,4\}$, $\{2,2,2\}$. These partitions represent possible data-layouts that can be constructed using $\rho\phi$ transformations.

2. Each of the multiplicative partition can be constructed by factorizing one, two, or three array dimensions. Note that for constructing data-layouts the factors $\{2,4\}$ is not the same as $\{4,2\}$. Each represents a different way of transforming the array dimension. Therefore, when calculating the number of possible data-layout choices, the total number of possible permutations of the factors is required, as opposed to calculating the possible combinations.

   The sum of all the permutations is ${}^4P_1 + {}^4P_2 + \frac{{}^4P_3}{3!}$, *i.e.*, 20.

We refer readers to (Odlyzko, 1995) to understand the details for these calculations.

The example shows that the number of data-layouts is already large for a four-dimensional case on an architecture with vector register length of eight. It grows for higher dimensional arrays, and longer architectural vector register lengths. Thus, it is not feasible to exhaustively generate code versions for all possible data-layouts derived using $\rho\phi$ transformations. This is the reason for using a speculative strategy, and generating code for a limited set of choices.

Steps in speculative SIMD vectorization

The data-layout choices for the SIMD code versions to be generated is done outside of QOPT. Chapter 8 describes the policy used for that purpose. The data-layout choices are provided as ATL specifications to QOPT, and the number of choices decide the number of code versions. For every QK, a code version corresponding to a particular data-layout is generated. A default non-SIMD code version is always generated. Depending on what data-layout is defined for the `mddarray` at runtime, one of the SIMD code versions, or the non-SIMD code version executes.

QOPT can add optional validations to ensure that a SIMD code version complies to the actual array shape specified at runtime. A data-layout is legal if none of the shifts on a reshaped dimension exceeds the size of that dimension. This check is to ensure no *stream alignment conflicts* occur, and no divisions or modulo operations are required to compute the shifted array access for a transformed array. The notion

of stream alignment conflicts is formally defined in Chapter 7. Section 6.4.4 describes why this constraint is required to avoid division and modulo operation in shifted array access calculations.

§**Implementation Note.** The currently implemented QOPT interface for specifying data-layout candidates is relatively simple. It allows specifying multiple layouts in the ATL format (Section 5.3) for `mddarrays` of a given rank. So, there is no provision to specify different data-layouts for two different `mddarrays` that have the same rank. A standalone policy engine also means that programmers should separately train the policy engine for it to generate data-layout candidates for their QKs. However, the advantage is in updating the policy without having to make changes to the compiler infrastructure.

The emphasis of the current implementation was to serve as a proof-of-concept of the speculative vectorization technique. It is possible that the system can be further automated and made more general. The policy engine can be moved into QOPT, and data-layout candidate generation made autonomous of programmer intervention. We propose such work for future investigation and implementation.

6.4 QOPT Late Scalarization

The last stage in QOPT's code-generation pipeline is called *late scalarization*. METAL array expressions get lowered into loops and array accesses at this stage. The term late scalarization was chosen to draw a contrast with other C++ template-based array programming techniques such as expression templates that perform scalarization in the template expansion stage. QUARC's late scalarization design overcomes inherent limitations in scalarizing early during template expansion. Scalarizing early leads to both loop and array access linearization in the C++ front-end. It may also lead to generation of calls to OpenMP, MPI, CUDA runtime libraries to support parallel execution. This makes it difficult for a compiler to retain enough context to infer the programmer's intent. Subsequent analysis and optimization, such as standard loop optimization techniques, becomes hard. Even simple high-level optimizations such as those discussed in Section 6.2.4 usually are impossible to apply on scalarized array expressions due to complicated loop structures and nested library calls.

6.4.1 Preventing Invalid Scalarization

QUARC's scalarization semantics are similar to other array languages such as FORTRAN 90 and High Performance FORTRAN. The RHS array expressions are fully evaluated without side-effects,

and only then are the results stored into the LHS. Implementing this "load-before-store" semantics requires correctness guarantees. To understand the reason, let us slightly modify the QK in our running example. Listing 6.5a shows the modified QK with the `a1` array used on both LHS and RHS. Listing 6.5b shows a scalarized version of this modified QK. Unfortunately, the scalarized version of the QK is not parallelizable and if executed in parallel would lead to incorrect results. The reason is that every `i-th` and `j-th` iteration of the scalarized loop nest depends on the results obtained in a previous iteration. This is known as *loop-carried dependence* (Kennedy and Allen, 2002). Given QUARC's data parallel programming model this type of scalarization is considered invalid. Invalid scalarization is a potential problem for any high-level array language. Some FORTRAN 90 compilers handled the situation by doing two-step scalarization of array assignment statements. The first step involved a "naive" scalarization of array assignment statements. The loops generated in the first step would approximate the loop nest shown in Listing 6.5b. A subsequent step would apply standard loop transformations to try and remove loop-carried dependence, and make the loops parallel. Such transformations require data dependence analysis to ensure validity. Several loop transformations can be performed after data dependence-based analysis to introduce parallelism, *e.g.* loop reversal, loop interchange, loop skewing, loop tiling, and generating array temporaries. (Kennedy and Allen, 2002) provides a detailed introduction to the methods.

QUARC, given its limited scope, enforces a set of hard constraints to avoid invalid scalarization. Unlike, FORTRAN 90 or similar languages where writing the kind of array statement shown in Listing 6.5a is legal, in QUARC this is an invalid statement whose output is undetermined. QUARC allows accesses on the same `mddarray` on both RHS and LHS of a QK if and only if all RHS accesses are free of shifts. This is part of the definition of a QK, and checked during QKET construction. However, compile-time checking using value-based analysis cannot guard against all cases. Pointer-based indirection can only be detected at runtime. QOPT can optionally generate additional runtime checks for such cases. The runtime checking is kept optional to allow programmer's and EDSL designer's control over when such a check is required. Without the runtime check, QUARC defers to the programmer to do the correct thing, and provides no implicit correctness guarantee. Note that this restriction cannot be violated by QK fusion optimization. Constraint (6.2) enforced by *qketfusion* ensures that QK fusion does not inadvertently introduce any loop-carried dependence.

These restrictions limit the type of programs that can be presently written using QUARC. Relaxing these restrictions is planned as a future extension. There are multiple options to add data dependence-

```
a1 = a1.GSHIFT< 1, 0>() + a1.GSHIFT<-1, 0>()
   + a1.GSHIFT< 0, 1>() + a1.GSHIFT< 0,-1>();
```

(a) Same array used on both LHS and RHS

```
// X, Y are the extents for each array dimension.
for(auto i = 1ul; i < X-1; ++i)
  for(auto j = 1ul; j < Y-1; ++j)
    a1[i][j] = a1[i+1][j] + a1[i-1][j] + a1[i][j+1] + a1[i][j-1];
```

(b) Incorrectly scalarized loop-nest

Figure 6.5: An example showing invalid scalarization of a METAL array assignment expression.

based analysis to QOPT. QOPT already uses ISL for polyhedral code-generation, and can potentially leverage ISL's polyhedral dependence analysis infrastructure. Even without using ISL's dependence analyzer QOPT can easily leverage other dependence analysis methods. METAL `GSHIFT` expressions by construction only involve single subscripts and are always linear. Thus, QOPT can potentially use simpler single-subscript dependence tests for QKs.

6.4.2 Loop Generation

QOPT uses polyhedral code-generation (Ancourt and Irigoin, 1991) to lower QKs in the QKET/QKEF form into loop nests and array accesses. Polyhedral code-generation uses integer sets to represent loop nests, and each integer set is mapped to a multi-dimensional time instance. This mapping, known as a schedule, determines the relative execution order of the loop iterations.

The first step in QOPT's loop generation is to build an index set representation for the *block-local* index space of an `mddarray`. METAL requires all `mddarrays` in a QK to have the same data placement, so a single integer set can be used per QK. The block-local index space represents the set of `mddarray` elements inside a block. Section 5.3 described the notion of hypercubic blocking of an `mddarray` using ATL data placement specifications. The integer set that is constructed for the block-local index space represents a set of loops that would iterate over each `mddarray` element in that block. Since, the size of a block is defined at runtime the upper bounds of the block-local index space is kept parameterized at compile-time. The parameters get resolved at runtime once the ATL specified partitioning is known. The block-local index set is constructed as part of the QKSCoP mid-level IR. On the block-local index set, QOPT applies index set partitioning to represent *split loops*. This is done

for QKs that have `GSHIFT` expressions. Split loops allow overlapping local computation with the MPI communication needed to gather remote data. After this step, once the final polyhedral representation inside a QKSCoP is built, it is converted into an Abstract Syntax Tree (AST) using the ISL polyhedral library. The ISL-AST is lowered into LLVM loops. QOPT embeds QUARC-RT library function calls at this stage to induce MPI communication and MPI synchronization. The final step in late scalarization is generation of the loop bodies. This step involves converting individual METAL expression tree nodes into LLVM code; various optimizations such as *if-conversion* (Allen *et al.*, 1983), scalar-expansion, and SIMD vectorization are introduced at this point. The following sections describe in detail all of the steps in the late scalarization.

§**Implementation Note.** The steps described in this section present the loop generation over a single block of an `mddarray`. Conceptually, a QUARC supports *overpartitioned* data distributions where a single process can own multiple blocks. In such cases, an outermost "block-loop" is required to iterate over all the local blocks. This is currently unimplemented, and block-loops are not part of the QKSCoP IR. QUARC and QUARC-RT prototypes only support a bijective mapping of `mddarray` blocks to processes, *i.e.*, each process owns only one block. Adding support for overpartitioned data distributions is part of a planned future extension of QUARC.

QKSCoP construction

QKSCoP construction is the first step in generating loops for a QK. A QKET represents an abstract perfectly nested set of loops, but the corresponding QKSCoP can represent multiple loops nests. These loop nests are not necessarily perfectly nested. There is no difference in the QKSCoP construction process between SIMD vectorized, and scalar loop generation scenarios. The only difference lies in generating the loop bodies. That process is explained in Section 6.4.4 when describing array access generation. Prior to going into the details of QKSCoP construction, the following definitions formalize the core concepts. The definitions use operators and notations introduced in Chapter 4.

Definition 6.3. Block index vector ($\boldsymbol{bi}$)

Given an n-dimensional `mddarray`, the block index vector $\boldsymbol{bi}$ of a particular element in a block of the `mddarray` is a vector of integers that gives the element's lexicographic position within that block. The rank of the block index vector is always the same as the rank of the blocked shape ($\boldsymbol{bs}$) of the

`mddarray`. Thus, a block index vector is given by

$$\boldsymbol{bi} \triangleq [bi_0, bi_1, \ldots, bi_{n-1}]$$

where $bi_k, 0 \leq k < n$, is the index for each dimension of $\boldsymbol{bs}$, and $\forall k, 0 \leq bi_k < \iota(k, \boldsymbol{bs})$, *i.e.* the index component for any dimension is less than the extent of the that dimension. △

An index vector always points to a lexicographic position, irrespective of the actual data-layout of the `mddarray`. For cases where the index space is for an `mddarray` with a $\rho\phi$ transformed data-layout, each index vector points to a SIMD vector. All other cases the index points to a single `mddarray` element. The set of all $\boldsymbol{bi}$ for an `mddarray` is its block-local index space.

Definition 6.4. QKSCoP

A QKSCoP is a five-tuple (*domain, parameters, inner region statement, [boundary domains], [boundary region statements]*). The integer set covering all loop iterations for a QKSCoP is known as its *domain*. The set of symbolic integer values that represent the upper bounds of the domain are known as the set of *parameters*. A statement is a set of loop iterations. Every QKSCoP has at least one statement known as the *inner region statement*. This statement includes all loop iterations that require only local data available inside an `mddarray` block. If a QK has `GSHIFT` expressions, then the QKSCoP has set of subsets of the domain known as *boundary domains*. These identify the index vectors where computing the output requires handling boundary conditions, and may require non-local data. Corresponding to the boundary domains, a QKSCoP may have a set of *boundary region statements*. Each boundary statement is a set of loop iterations over one or more boundary domains. Boundary domains and statements are optional attributes of a QKSCoP. △

Definition 6.5. Statement

A statement is a three-tuple (*parent, domain, schedule*). *Parent* refers to the QKSCoP to which the statement belongs. The *domain* of a statement is the integer set that identifies the set of loop iterations executed by the statement. The *schedule* of a statement is a mapping of its loop iterations to a multi-dimensional point in time. The schedule determines the relative ordering of various statements included in a QKSCoP. △

Procedure qkscopGenaration

Input: QK qk
Output: QKSCoP S

define an unbounded integer set I;
// generate a set symbolic parameters
foreach *dimension* n *of* $\boldsymbol{bs}$ **do**
    create a symbolic parameter D_n;
end
// add constraints to define the full index space for qk
foreach *dimension* n *of* $\boldsymbol{bs}$ **do**
    add a constraint to I setting the lower bound as $0 \leq$ for this dimension;
    add a constraint to I setting the upper bound as $< D_n$ for this dimension;
end
// generate subsets of I to represent disjoint boundary points for
// every RHS GSHIFT expression.
foreach *RHS* `GSHIFT` *expression* **do**
    $\boldsymbol{sv} \leftarrow$ `access_fn` function arguments;
    $S.BregSets \leftarrow$ `genBoundaryDomains` $(I, \boldsymbol{sv})$;
end
// add statement for local loop iterations of the QKSCoP
$S.IregStmt \leftarrow$ construct statement to loop over the entire index set I;
// add statements for boundary loop iterations
if *there are any RHS* `GSHIFT` *expression* **then**
    $\boldsymbol{msv} \leftarrow$ store maximal shift in each direction for every dimension;
    $S.BregStmts \leftarrow$ `addBoudaryRegionStatement` $(I, \boldsymbol{msv})$;
end
return S;

Figure 6.6: Steps involved in generating a QKSCoP representation for a QK.

Figure 6.6 presents QOPT's QKSCoP construction procedure. The following paragraphs describe each individual step.

Defining the domain. The domain for every QKSCoP is initially constructed as an unbounded integer set. It is then constrained by a default set of constraints that define its lower and upper bounds. The upper bounds use symbolic parameters, and the lower bound is always zero. For the five point stencil in our running example, the domain for the QKSCoP is given by

$$[D_0, D_1] \rightarrow \{[i_0, i_1] : 0 \leq i_0 < D_0 \; and \; 0 \leq i_1 < D_1\}$$

where, D_0, D_1 are symbolic parameters, and i_0, i_1 represent generic index variables for the two dimensions.

Procedure genBoundaryDomains

Input: Index set I
Input: Vector of shift offsets $\boldsymbol{sv}$
Output: Vector of boundary index sets B

```
// Initialize the boundary index sets vector with I
B ← I;
foreach shift offset s in sv do
    if s ≠ 0 then
        foreach index set b in B do
            I_tmp ← I;
            if s < 0 then
                // project out the upper bound for the shifted dimension
                I_tmp ← project out upper bound constraint on I_tmp for the shifted dimension;
                I_tmp ← add a new upper bound constraint <−s;
            end
            else if s > 0 then
                // project out the lower bound for the shifted dimension
                I_tmp ← project out lower bound constraint on I_tmp for the shifted dimension;
                I_tmp ← add a new lower bound constraint ≥ D_n−s;
            end
            // create the boundary index set by intersecting I_tmp with b
            b_new ← I_tmp ∩ b;
            // subtract I_tmp from b to construct subsequent disjoint boundary region
            // sets for any shifts on other lower dimensions
            b ← b \ I_tmp;
            // add b_new to B.
            insert b_new into B;
        end
    end
end
// remove the first set inside B, as this is the residual non-boundary region
B ← remove B[0];
return B;
```

Figure 6.7: Steps involved in generating boundary region index sets from a vector of shift offsets.

The domain does not include nested or `sdlarray` dimensions. Nested dimensions are only accessed inside METAL's mkernel elemental functions. Any loops over nested dimensions inside an mkernel function is typically fully unrolled, and the mkernel function itself inlined. Section 6.4.4 provides further details.

Boundary separation. The `qkscopGeneration` procedure performs an additional step when there are RHS `GSHIFT` expressions in the QK. It constructs disjoint subsets of the domain to represent boundary regions corresponding to each `GSHIFT` expressions. An `mddarray` access inside a boundary

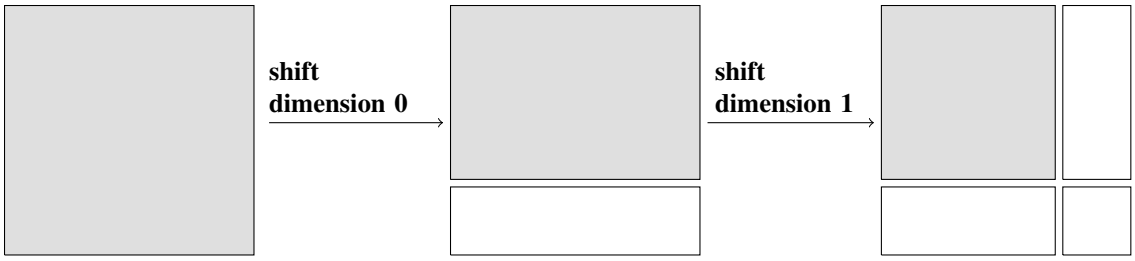


Figure 6.8: Separating the boundary regions for a multi-dimensional `GSHIFT<1,1>()` expression. The gray boxes depict the inner region points, and the white boxes are the boundary points. At the end of the process the `GSHIFT` results in three disjoint boundary region integer sets.

region requires applying the boundary function defined for the `mddarray`. The access may be for a non-local array element, in which case MPI data communication is required. The primary reason for the boundary separation is to allow overlapping remote data communication with local computation.

The *genBoundarySets* procedure shown in Figure 6.7 implements the boundary region domain construction. Boundary region domains are constructed individually for each `GSHIFT`. For each dimension with a shift, *genBoundarySets* projects out the existing constraints on that dimension. The projection operation is provided by the ISL library, and makes the integer space unbounded for the projected out dimension. After projecting out existing constraints, a new set of constraints based on the shift's integer value is constructed. As shown in Figure 6.7, the constraint depends on the sign or *direction* of the shift. A positive value is a forward shift, and a negative value is a backward shift. These new constraints are applied to a copy of the original domain to create a subset of the original integer set. This subset is the boundary region domain for the particular shift. The *genBoundarySets* procedure builds four boundary region domains for out five-point stencil running example. These boundary regions can be visualized as the last and first rows, and the last and first columns of a rectangular two-dimensional block. The four boundary regions are

$$[D_0, D_1] \rightarrow \{[D_0{-}1, i_1] : D_0 > 0 \; and \; 0 \leq i_1 < D_1\};$$

$$[D_0, D_1] \rightarrow \{[0, i_1] : D_0 > 0 \; and \; 0 \leq i_1 < D_1\};$$

$$[D_0, D_1] \rightarrow \{[i_0, D_1{-}1] : D_1 > 0 \; and \; 0 \leq i_0 < D_0\};$$

$$[D_0, D_1] \rightarrow \{[i_0, 0] : D_1 > 0 \; and \; 0 \leq i_0 < D_0\};$$

where, D_0, D_1 are symbolic parameters, and i_0, i_1 represent generic index variables for the two dimensions.

GenBoundarySets handles multi-dimensional shifts as well. Figure 6.8 depicts how *genBoundarySets* handles multi-dimensional shifts. It starts with the outermost shifted dimension, and generates a boundary region set for that shift. The boundary is subtracted from each existing index set produced till that point. The new boundary set in then appended to the collection of boundary sets. This produces multiple disjoint subsets for a multi-dimensional shift. In Figure 6.8 the multi-dimensional `GSHIFT<1,1>()` produces three boundary region sets. Note that the boundary region collection is initialized with the complete integer set as the initial entry. At the end of *genBoundarySets* the first entry represents the residual inner region, and is purged from the collection.

Statement construction is the final step in QKSCoP construction. *Qkscopgen* adds an inner region statement to iterate over the whole index space for the QK. The inner region statement's domain is the same as the parent QKSCoP's domain, and it includes loop iterations over all boundary regions. Boundary checks inlined within the innermost loop of the generated loop nest handle the boundary region accesses. Additional boundary region statements are constructed whenever the QK has `GSHIFT` expressions. The domain of each boundary region statement is calculated by integer set operations.

§**Implementation Note.** The statement generation design is an optimization that has to do with QUARC's use of the MPI-3 for shared memory parallelism. The MPI-3 standard allows MPI processes on the same shared memory domain to access each other's memory using direct loads and stores. Therefore, non-local data accesses inside a boundary region may in fact be on the same shared memory domain, and if MPI ranks are provisioned carefully the non-local data may even be within the same cache block. Scheduling the boundary iterations separately would not yield any benefit in these scenarios.

QUARC-RT decides whether to use MPI-3 load and stores, or MPI-2 one-sided communication when it generates the communication plan. The checks inside the inner region statement use this information. If a boundary access can be handled by a load from a shared memory address it gets executed inside the inner region statement. Otherwise, the computation happens in a boundary region statement.

Boundary region statements are constructed whenever the QK has `GSHIFT` expressions. Boundary statements do not necessarily have a one to one correspondence to boundary region domain. To minimize the number of boundary statements that are needed, a single boundary statement can include iterations over multiple boundary region domains. Therefore, boundary checks are also required inside boundary

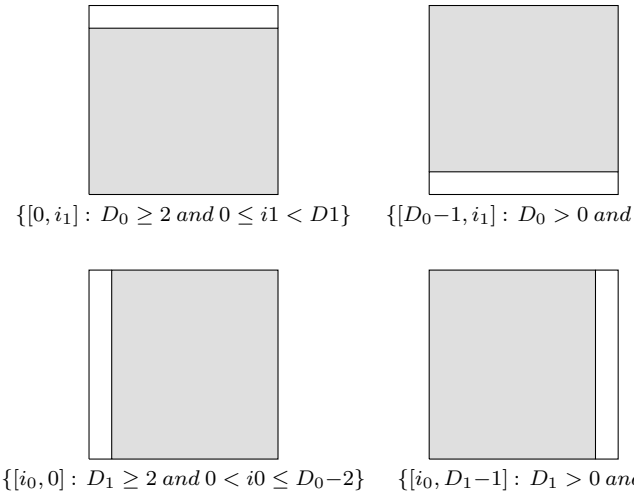
Procedure addBoudaryRegionStatement

Input: Index set I
Input: Vector of maximum shift offsets $\boldsymbol{msv}$
Output: Vector of boundary statements $BregStmts$

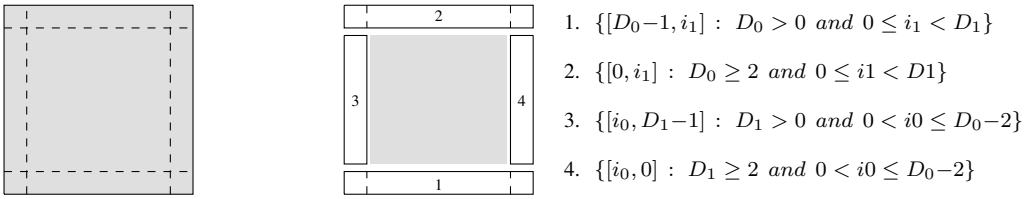
```
// Initialize a one-dimensional ''time'' to schedule the statements
```
$T \leftarrow 0$;
foreach *shift offset* s *in* $\boldsymbol{msv}$ **do**
  $I_{tmp} \leftarrow I$;
  if $s < 0$ **then**
    // project out the upper bound for the shifted dimension
    $I_{tmp} \leftarrow$ project out upper bound constraint on I_{tmp} for the shifted dimension;
    $I_{tmp} \leftarrow$ add a new upper bound constraint $< -s$;
  end
  else if $s > 0$ **then**
    // project out the lower bound for the shifted dimension
    $I_{tmp} \leftarrow$ project out lower bound constraint on I_{tmp} for the shifted dimension;
    $I_{tmp} \leftarrow$ add a new lower bound constraint $\geq D_n - s$;
  end
  // create the boundary index set by intersecting I_{tmp} with I
  $Breg_{dom} \leftarrow I_{tmp} \cap I$;
  // subtract I_{tmp} from I to construct subsequent disjoint boundary statements
  $I \leftarrow I \setminus I_{tmp}$;
  $BregStmt \leftarrow$ new QKSCoP-Statement with $Breg_{dom}$ and T;
  // increment T value for the next statement
  $T \leftarrow T+1$;
  insert $BregStmt$ into $BregStmts$;
end
return $BregStmt$;

Figure 6.9: Steps involved in generating boundary statements from the maximal shifts in each direction for every dimension of an `mddarray`.

region statements. Figure 6.9 presents the *genBoundaryStatements* procedure that is used to the create boundary statements for a QKSCoP. *GenBoundaryStatements* requires a single input vector containing the maximum forward and backward shift for each `mddarray` dimension. The set operations within *GenBoundaryStatements* are identical to the set operations inside *genBoundarySets*. However, as boundary region statements are constructed for the maximal shift in a cardinal direction, the set intersection and difference operations have to be performed only once per boundary region statement. After generating all the boundary region statements they are scheduled with the outermost dimension boundary region statements scheduled first, followed by inner dimension boundary region statements. Within a dimension the forward direction boundary statement is scheduled before the backward direction boundary statement.



a. Boundary domains



b. Inner-region statement c. Boundary-region statements

Figure 6.10: Boundary domains and statements for a five-point stencil. D_0, D_1 are the symbolic upper bounds for each dimension. The dashed lines in (b), (c) shows the intersection with boundary domains. The inner-region statement intersects all four boundary domains. The top and bottom boundary statements intersect the lower dimension boundary domains.

The boundary statement domains for our the five-point stencil running example are given by

$$[D_0, D_1] \rightarrow \{[D_0{-}1, i_1] : D_0 > 0 \; and \; 0 \leq i_1 < D_1\};$$

$$[D_0, D_1] \rightarrow \{[0, i_1] : D_0 \geq 2 \; and \; 0 \leq i1 < D1\};$$

$$[D_0, D_1] \rightarrow \{[i_0, D_1{-}1] : D_1 > 0 \; and \; 0 < i0 \leq D_0{-}2\};$$

$$[D_0, D_1] \rightarrow \{[i_0, 0] : D_1 \geq 2 \; and \; 0 < i0 \leq D_0{-}2\};$$

where, D_0, D_1 are symbolic parameters, and i_0, i_1 represent generic index variables for the two dimensions.

```
1  // Inner region statement's headers
2  for (int i0 = 0; i0 < D0; i0 += 1)
3    for (int i1 = 0; i1 < D1; i1 += 1);
4
5  // Boundary regions' loop headers
6  {
7    if (D0 >= 1) {
8      for (int i1 = 0; i1 < D1; i1 += 1);
9      if (D0 >= 2)
10       for (int i1 = 0; i1 < D1; i1 += 1);
11   }
12   if (D1 >= 1) {
13     for (int i0 = 1; i0 < D0 - 1; i0 += 1);
14     if (D1 >= 2)
15       for (int i0 = 1; i0 < D0 - 1; i0 += 1);
16   }
17 }
```

Listing 6.5: Loop headers generated for the five-point stencil running example.

Figure 6.10 shows the final boundary domains and statements generated for the five-point stencil example. Figure 6.10.a shows the four boundary domains that each correspond to the four `GSHIFT` expressions. Figure 6.10.b shows the inner-region statement. The dotted lines represent the iterations within the statement that overlap with a boundary domain. These are handled with inline boundary check conditions within the innermost loop body. Figure 6.10.c shows the four disjoint boundary statements. The number inside each box shows the schedule of the boundary statement. Dotted lines are used once more to show boundary domain intersections.

ISL-*AST generation.* Every QKSCoP is lowered into an ISL-AST. The AST generation translates the domains for each statement into loop headers. This step is the final step in QOPT's loop generation process. Listing 6.5 shows the loop headers built for our five-point running example. Note that the loop bodies are empty at this point. The loop bodies get populated in the array access and mkernel generation step of late scalarization. The split loop nests in the ISL-AST format are next lowered into LLVM loops.

6.4.3 QUARC-RT library calls generation

QUARC-RT library calls abstract MPI data communication and synchronization routines. Prior to generating the loop bodies for QK, QOPT adds the needed QUARC-RT library calls. Figure 6.11 shows the basic block-level CFG that is generated to add these calls. To make it readable, the CFG was significantly simplified. All LLVM instructions apart from the function calls are elided. Single

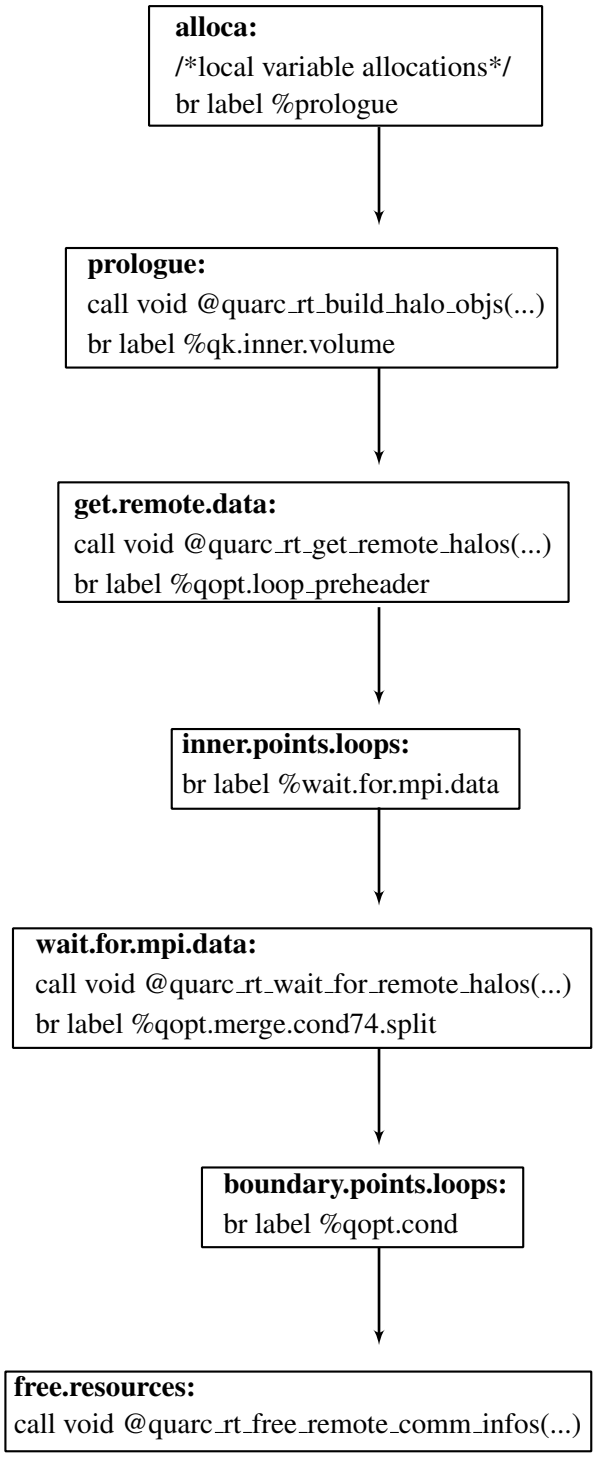


Figure 6.11: Basic block CFG for a typical QK

superblocks are used to represent the inner region, and boundary region loop nests. The superblock abstracts the CFG region consisting of multiple basic blocks for each loop nest.

The process of MPI communication is split into two steps. The first one builds a communication plan. The step involves identifying neighboring MPI ranks, allocating buffers for remote data, and calculating starting offsets into the buffers from which data would have to be read. The function all inside the `prologue` basic block represents this step. The next step does actual MPI one-side data movement. QOPT tries to overlap local computation with remote data movement. This is the reason for creating the split loop nests for any QK that has at least one `GSHIFT` expression. This can also be seen in the basic block control flow in Figure 6.11. The call inside the `get.remote.data` basic block initiates an asynchronous MPI one-sided communication step. This is followed immediately by the `inner.points.loops` block where the loop nest corresponding to the inner region QKSCoP statement gets built. The QUARC-RT call to wait for remote data are inserted inside `wait.for.mpi.data`. All boundary region loops are built inside `boundary.points.loops`. The `global.reduction` is an optional step that is used only for reduction QKs. All final cleanups, including freeing resources, is done inside `free.resources`. The following QUARC-RT library calls are part of the QOPT code-generation processes.

A `quarc_rt_build_halo_objs` call uses the `GSHIFT` information for the QK to generate a communication plan. This involves identifying the remote processors from which data needs to be moved, and allocation of local buffers to store remote data. Note that this step only builds a communication plan, and does not execute any actual communication. This split of the communication plan generation and the actual communication is done for an important reason. If a QK is executed multiple times inside a loop, as is likely when writing iterative solvers, a communication plan once created is reusable for each execution of the QK. QOPT tries to detect such scenarios, and if successful hoists this call out of the loop enclosing the QK.

A `quarc_rt_lock_local_win` call uses MPI-3 passive target synchronization (Gropp *et al.*, 2014) to start an exclusive *access epoch* on the LHS array of a QK. The call internally uses an `MPI_Win_lock` operation. Inside the exclusive access epoch, the LHS array can only be accessed by the local MPI process. This is done to prevent any remote process from accessing the LHS array using an MPI-3 one-sided routine, before all local updates are complete.

A `quarc_rt_get_remote_halos` initiates MPI data movement. The previously generated communication plan is used to access remote data. The data movement involves the non-blocking MPI one-sided `MPI_Get` routine. The intent is to overlap movement of remote data to the local processor with the local computation for the QK. Note that this call does not necessarily mean that data movement is required. A `GSHIFT` can be on a non-distributed array dimension, or all array partitions may be on the same shared memory domain. For such cases this call would return without invoking any MPI communication.

A `quarc_rt_wait_for_remote_halos` call blocks execution of an MPI rank till the remote data has been moved to a local buffer. QOPT inserts a call to the function at a point where the local computation of a QK is known to be complete.

A `quarc_rt_unlock_local_win` call unlocks the LHS array and ends the exclusive access epoch on the LHS array.

A `quarc_rt_mpi_allreduce` call is an optional global communication call that is used for reduction QKs. As the name indicates, this QUARC-RT call wraps around an `MPI_Allreduce` routine.

A `quarc_rt_free_remote_comm_infos` call frees up resources by deleting local data buffers and the QUARC-RT communication plan. As with `quarc_rt_build_halo_objs`, if a QK is inside a loop nest QOPT tries to move this call out of the loop nest by sinking it after the outer loop exit block. The sinking happens in conjunction with the hoisting of the `quarc_rt_build_halo_objs` call.

6.4.4 Loop body generation

Loop body generation for QKSCoP statements is the final compilation step inside QOPT. The process starts by doing a post-order walk of the QKET's RHS sub-tree, emitting the required code for each QKET node that it visits. The *qketBuildRHS* procedure shown in Figure 6.12 presents a high-level overview of the loop body generation steps. Each QKET node has different code-generation requirements. This section goes over the specifics of each case.

Building a `DRILL` expression node requires only updating the index vector that is used to build a linearized array access. The METAL `DRILL` operator generates a constant index value for a nested `sdlarray` dimension. This constant value is appended to the index vector, and used during address linearization.

```
Procedure qketBuildRHS
    Input: QKET qketNode
    Input: QKSCoP for the QKET qkscop
    Output: Vector of LLVM virtual register values V
 1  switch qketNode.node_type do
 2      case DRILL do
 3          updateIndexVextor ();
 4          lv ← qketBuildRHS (qketNode.leftChild);
 5      end
 6      case IF_EVEN_CHOOSE do
 7          lv ← qketBuildRHS (qketNode.leftChild);
 8          rv ← qketBuildRHS (qketNode.rightChild);
 9          V ← applyIfConversion (lv, rv);
10      end
11      case binary mkernel do
12          lv ← qketBuildRHS (qketNode.leftChild);
13          rv ← qketBuildRHS (qketNode.rightChild);
14          V ← inlineMkernel (lv, rv);
15      end
16      case unary mkernel do
17          lv ← qketBuildRHS (qketNode.leftChild);
18          V ← inlineMkernel (lv);
19      end
20      case array access do
21          addinlineBoundaryChecks (qkscop, qketNode);
22          V ← createArrayAccess (qketNode);
23      end
24      case scalar access do
25          V ← createScalarAccess (qketNode);
26      end
27      otherwise do
28          Other
29      end
30  end
31  return V;
```

Figure 6.12: A recursive post-order walk is used to lower the RHS sub-tree of a QKET into LLVM instructions. The output of *qketBuildRHS* is a set of LLVM virtual register that stores the output of a memory read operation or an arithmetic operation. The LLVM virtual register values get assigned to the LHS using LLVM memory write operation.

Building an `IF_EVEN_CHOOSE` expression node results in an implicit if-conversion optimization. As defined in Section 5.1.2.5, METAL's `IF_EVEN_CHOOSE` operator generates this type of expression node. The expression has two children that are terminal `GSHIFT` expressions. The expression encapsulates a built-in predicate (Listing 5.1) that uses the local `mddarray` indices to compute a global "parity" value at each `mddarray` index position. *QketBuildRHS* generates the predicate as an inlined computation inside the innermost loop body. It then adds LLVM `select` instructions to select one of the two accesses at runtime. Thus, generating code for `IF_EVEN_CHOOSE` expressions avoids adding extra control flow into the loop.

Inlining mkernel functions is done using a QOPT's custom domain-specific function inliner. This custom inliner benefits from domain-specific information that a general-purpose function inliner cannot decipher. The custom inliner applies optimizations that a general-purpose inline would not be able to discover.

Mkernel call inlining occurs only in the context of QKET loop body generation. As *qketBuildRHS* does a depth-first walk of the tree, the mkernel function inliner is aware what arguments are passed to the mkernel call. Along with that it is also aware of guarantees provided by METAL API. For mkernels that operate on `sdlarray` data types, the inliner removes all nested `sdlarray` accesses. These nested accesses get replaced by a single linearized address offset from the parent `mddarray's` base address. It may do so as METAL ensures that nested `sdlarray` members are allocated contiguously inside an `mddarray`. For SIMD vectorized code-generation the inliner converts all scalar arithmetic operations into SIMD vector operations.

Building scalar `mddarray` accesses requires two things to be considered: adding boundary condition checks when the QK has `GSHIFT` expressions, and generating LLVM memory operations.

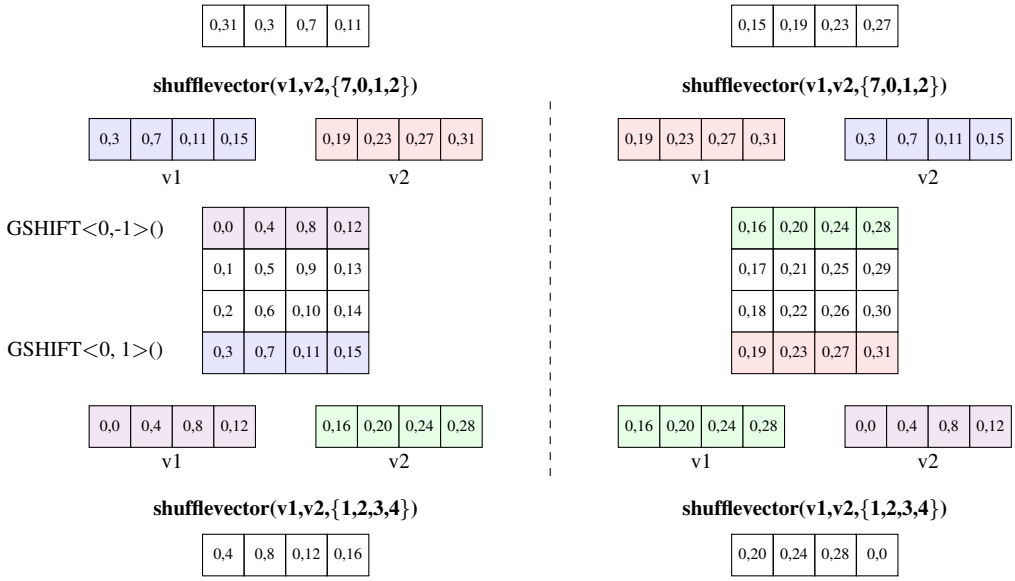
Boundary condition checks are derived using integer set operations involving a QKSCoP statement's iteration domain and the QKSCoP's boundary domains. A boundary check is inserted whenever a statement's domain intersects a boundary domain. If an `mddarray` access falls inside a boundary, then it is loaded from the memory region corresponding to that boundary. The memory region can be a shared memory address, an address inside the same block, or a memory buffer that stores data copied over from a remote process. This distinction between memory regions is abstracted by QUARC-RT. Listing 6.6 shows the loop nests for the five-point example with the boundary condition checks added.

```
// Inner region statement's headers
for (int i0 = 0; i0 < D0; i0 += 1)
  for (int i1 = 0; i1 < D1; i1 += 1) {
    // Check if last row
    if (i0 == D0-1) { }
    // Check if first row
    if (i0 == 0) { }
    // Check if last column
    if (i1 == D1-1) { }
    // Check if first column
    if (i1 == 0) { }
  }
// Boundary regions' loop headers
{
  if (D0 >= 1) {
    for (int i1 = 0; i1 < D1; i1 += 1) {
      // Check if last column
      if (i1 == D1-1) { }
      // Check if first column
      if (i1 == 0) { }
    }
    if (D0 >= 2)
      for (int i1 = 0; i1 < D1; i1 += 1) {
        // Check if last column
        if (i1 == D1-1) { }
        // Check if first column
        if (i1 == 0) { }
      }
  }
  if (D1 >= 1) {
    for (int i0 = 1; i0 < D0 - 1; i0 += 1);
    if (D1 >= 2)
      for (int i0 = 1; i0 < D0 - 1; i0 += 1);
  }
}
```

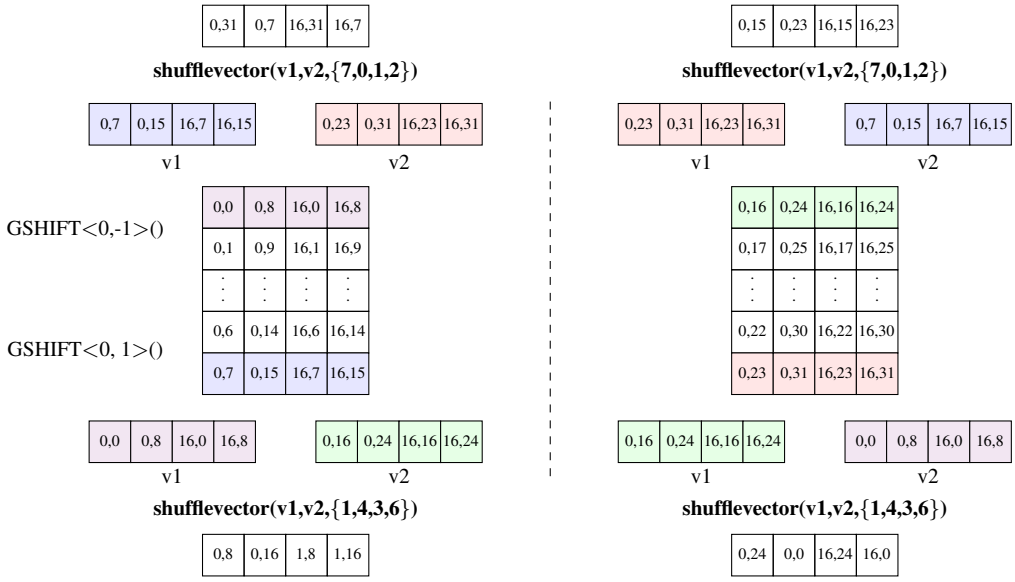
Listing 6.6: Inline boundary checks inside generated loops

LLVM load instructions are inserted after the boundary checks. The code path that loads for non-boundary domain cases, adds a load from the local MPI window for the `mddarray`. For the code path where an access falls inside a boundary domain, the load is from the memory region pointer returned by QUARC-RT. Each nested `sdlarray` element is loaded with a separate load operation.

Building SIMD `mddarray` accesses involves extra steps compared to building scalar `mddarray` accesses. The boundary condition checking is same as scalar `mddarray` accesses, but the loads inside a boundary iteration involves vector shuffle operations. These shuffle operations are needed due to an extra boundary condition introduces by QUARC's $\rho\phi$-based data-layout transformations. Reshaping of an `mddarray` dimension introduces an internal boundary within an `mddarray` block. Nearest neighbor operation on the elements in an internal boundary region require elements from within the same block as



(a) SIMD data-layout created using ATL specification v:RT(1,4). Only the inner array dimension is reshaped and transposed to build the four-wide vector dimension.



(b) SIMD data-layout created using ATL specification v:RT(2,2). Both array dimensions are reshaped and transposed to build the four-wide vector dimension.

Figure 6.13: Showing the handling of boundaries for $\rho\phi$ transformed `mddarray` layouts using vector shuffle operations. The example uses a 32×32 `mddarray` that is blocked on the inner dimension. The sub-figures show two possible data-layouts within a block. A global two-dimensional indexing scheme is used to help understand the data distribution and data-layout. The array uses a periodic boundary condition.

well as a neighboring block. The data elements need to be shuffled to get them in the right vector lanes, before applying any SIMD arithmetic operations.

Figure 6.13 shows two scenarios that illustrate this need for vector shuffling. The two scenarios show two different $\rho\phi$ data-layout transformations on a two-dimensional `mddarray` with a 32×32 global shape. The inner dimension of the `mddarray` has been blocked by a factor of two. Each subfigure shows the first row within the two neighboring blocks. Notice that after the layout transformations, there is an inner vector dimension within each row. Therefore, each row consists of multiple vectors. The data-layout in both scenarios is different. In Figure 6.13a, the inner `mddarray` dimension is $\rho\phi$ transformed to build a four-wide vector dimension. In Figure 6.13b, both `mddarray` dimensions are transformed to build the vector dimension. As shown, the two `GSHIFT` operations, `GSHIFT<0,1>()` and `GSHIFT<0,-1>()`, on the inner dimension require data to be gathered from two different vector registers. These two vector registers must be shuffled or blended to get the needed elements in the right position inside a single vector register. The shuffle operation uses architecture-specific blend instructions, such as the AVX `VBLENDVPS` and `VPBLENVPS` instructions, for this purpose. A blend instruction needs an instruction mask specified as a list of unsigned integer values to select the required elements from either vector register. The instruction mask needs to be generated at compile time.

We ensure that all shifts in a given direction for a reshaped dimension use the same instruction mask. That is, the instruction mask value only depends on the $\rho\phi$ transformation, and not on the shift value. This is done by enforcing a legality constraint when selecting a $\rho\phi$ transformation to define an `mddarray` data-layout. This constraint is defined as follows:

Constraint 6.3. The absolute value of a shift on a $\rho\phi$ transformed `mddarray` dimension should be less than the reshaped extent of that dimension.

Constraint (6.3) ensures that all array elements in inner-region iterations have the shifted neighbor in the same vector lane on another vector register. For boundary-region iterations, the shifted neighbors of the array elements are in the next or prior vector lane. Moreover, boundary region separation follows the same logic as described in Section 6.4.2.

This constraint essentially restricts the space of applicable $\rho\phi$ transformations. The rationale for the constraint is based on the index mapping formulae defined in Section 4.2. Translating an array accesses from a lexicographic index space to a $\rho\phi$ transformed index space requires integer division and modulo

```
auto sq_norm = REDUCE(a1*a1, su3add);

/* Semantically equivalent loop nest for the METAL REDUCE expression
 * auto sq_norm = 0.0;
 * for (auto k = 0ul; k < a1.get_local_extent(0); ++k) {
 *   for (auto l = 0ul; l < a1.get_local_extent(1); ++l) {
 *     autp sq = sqmag(a1.at(k,l), a1.at(k,l));
 *     sq_norm = su3add(sq_norm, sq);
 *   }
 * }
 *
 * quarc_rt_mpi_allreduce(...);
 */
```

Listing 6.7: After optimizing the REDUCE expression

operations. Applying the constraint ensures no division or modulo operation is needed, and a fixed translation of the shifted is possible for all shifts for a given $\rho\phi$ defined data-layout.

Building scalar terminal nodes involve generating a load operation for the scalar variable referenced by the QKET leaf node. For SIMD code-generation, each scalar load is expanded into a vector load with the scalar value replicated across all the vector lanes. This is a standard compiler optimization known as *scalar expansion*.

6.4.5 Reductions

Code-generation for reduction QKs is very similar to the steps described for regular QKs. Listing 6.7 shows a METAL code snippet calculating the squared norm for an `mddarray`. The `mddarray` is of the same type as in our five-point stencil example, its elements are of the `su3` data type. The `su3add` mkernel introduced in Listing 6.1 is used as the accumulator for this reduction operation. The `sqmag` mkernel is elided in the example, it squares the value of each component an `su3` vector.

The commented code section in Listing 6.7 shows the corresponding loop-nest generated for the METAL expression. Every reduction internally uses an accumulator variable where the output of the local reduction step is stored. The variable `sq_norm` is the accumulator in this example. A global reduction follows the local reduction step. The mkernel function provided to the `REDUCE` operator is converted to an `MPI_User_function` and used for the global reduction. The global reduction step is executed by the QUARC-RT `quarc_rt_mpi_allreduce` library function call.

6.5 QUARC-RT: Runtime Time System

6.5.1 Halo Generation and Communication Optimization

QUARC-RT implements a polyhedral integer set analysis based automatic method for MPI communication generation. Our method has similarities with earlier work done in the Rice dHPF compiler (Adve and Mellor-Crummey, 1998). Like dHPF's implementation, QUARC-RT uses affine array access functions to derive the set of array indices that are part of a communication set. QUARC's implementation has advantages over dHPF, and even a more recent design based on polyhedral dependence analysis proposed by Bondhugula (Bondhugula, 2013). QUARC's derivation of communication sets happens at runtime, and with exact awareness of how the arrays have been distributed over an actual MPI Cartesian communicator. This makes it easier to do exact analysis, and omit all statically generated checks that dHPF or Bondhugula's method requires.

The communication set generation algorithm is essentially the same as the *genBoundarySets* procedure in Figure 6.7. The only difference is that unlike *genBoundarySets* QUARC-RT does not use symbolic parameters as the array block shape is already known at runtime. The communication sets for a `GSHIFT` expression is computed as an index set inside another neighboring array block. QUARC-RT then optimizes these index sets by consolidating multiple sets for different `GSHIFT` expressions inside the same QK using set union operations. The consolidated set of indices that is needed from the same block is then moved using a single MPI remote memory access (RMA) operation using MPI's hvector data types. There is a further optimization inside QUARC-RT. RMA is used only when QUARC-RT determines that the neighboring array block is on another processor node. Our implementation uses MPI's split communicator functionality for this purpose. If QUARC-RT determines that the needed array block is within the same shared memory domain, then direct memory accesses are used instead of RMA operations. Whenever RMA is required, QUARC-RT allocates a local memory buffer or *halo* to store the remote data. The size of the halo is directly derived from the communication set.

6.5.2 Data Distribution Functions

QUARC allows the possibility of using application specific data distributions by defining custom mapping functions. The design cleanly separates the definition of hypercubic blocks for `mddarrays`,

using the ATL dist-rts argument, from the mapping of the blocks on to a processor grid that was defined using the ATL p-grid argument. Using this design non-trivial data distributions, such as those implemented in the Rice dHPF compiler (Mellor-Crummey *et al.*, 2002), become implementable relatively easily. Implementing new types of data distributions that follow the basic hypercubic array blocking design would not involve any changes to METAL or ATL. The only addition would be inside QUARC-RT. Array blocks defined via ATL are assigned a unique multi-dimensional block id. QUARC-RT defines the processor grid as an MPI Cartesian communicator, and each processor or MPI rank in that communicator is assigned a multi-dimensional Cartesian coordinate by the MPI runtime. Defining a data distribution involves providing the mapping between the multi-dimensional block id space to the MPI Cartesian coordinate space. This mapping can be done as per the application need, and all data movement and communication get transparently handled based on which blocks are mapped to which processor.

6.5.3 Data-Layout Selection

QUARC currently implements an external policy engine (Chapter 8) to select a set of data-layout candidates for an application. The ATL specification generated according to the policy only has the set of simd-rtf values, and the relative ranking of each value. The QUARC-RT runtime library selects one of the simd-rtf values for an `mddarray`. The selection is made according to the `mddarray` global shape and block distribution. QUARC-RT evaluates if the simd-rtf value is applicable to the blocked shape of the `mddarray`, but applies no other fitness criteria. QOPT can optionally add validations inside the generated code to check if the data-layout specified via the selected simd-rtf value works for a given QK.

CHAPTER 7: PERFORMANCE ANALYSIS OF DATA-LAYOUTS

We now analyze the impact of $\rho\phi$ data-layout transformations on SIMD vectorization of stencil kernels. The analysis was done on three modern Intel Architectures (IA) servers using QOPT's custom vectorizer. The results in this chapter form the empirical basis for QUARC's layout selection policy engine described in Chapter 8.

7.1 Background

7.1.1 Stencil Kernels

A stencil kernel is an iterative computation that updates an array element according to a fixed computational pattern involving neighboring array elements in the same or in a separate array. The fixed computational pattern is known as a stencil. Stencil computations is encountered in several scientific domains, such as differential equation solvers, image processing, finite-element methods, and cellular automata. For this reason, stencil kernels are recognized to be one of the core kernels of scientific computing (Asanovic *et al.*, 2009).

A large body of work exists around optimizing stencil kernels for different performance concerns, such as cache reuse, communication optimizations, and SIMD vectorization (Roth *et al.*, 1997), (Kamil *et al.*, 2005), (Datta *et al.*, 2008), (Tang *et al.*, 2011), (Ragan-Kelley *et al.*, 2013), (Henretty *et al.*, 2013), (Acharya and Bondhugula, 2015), (Rawat *et al.*, 2015), (Yount, 2015). A majority of the methods look to improve cache performance of stencil kernel loops by introducing loop tiling or cache blocking methods. Roth *et al*. (Roth *et al.*, 1997) explored data communication optimizations to improve the performance of stencil kernels on distributed-memory machines.

From this list of prior work, notably Henretty *et al*. (Henretty *et al.*, 2011), and Yount (Yount, 2015) have looked at techniques specific to improving the SIMD vectorization performance on IA short-vector machines. Both techniques deal with a fundamental issue with SIMD vectorization of stencil kernels that has its root in the architectural design of short-vector machines. The central premise of short-vector

SIMD vectorization is *streaming* contiguous chunks of memory into short SIMD vector registers that have multiple channels or lanes. A single SIMD instruction is then applied to the data in each lane. The big limitation in SIMD vectorization is that content of two vector registers need to be *stream aligned*, *i.e.*, the operands of a SIMD operation need to be in the same lane of the respective registers. Typically, cross lane SIMD operations are not supported by short-vector machines. Henretty *et al*. used the term *stream alignment conflict* to describe this problem. To get over the issue of stream alignment conflicts, they proposed a data-layout transformation technique. The technique that they termed as *dimension-lift and transpose* is subsumed in our $\rho\phi$ algebra. Yount used a different technique. His method applied in-register swizzle operations to get data stream aligned.

7.1.2 Short-vector SIMD architectures

The SIMD vector processing units on most modern architectures, such as x86, AMD64, Power, and ARM64, are classified as streaming SIMD multimedia extensions. These architectures use Instruction Set Architecture (ISA) extensions to add short-vector registers to the architecture to support SIMD vectorization. The register size of SIMD ISA extensions on IA has doubled with each newer processor generation. The earliest Intel MMX extension offered 64-bit vector registers that provided eight 8-bit lanes. This has increased, as of 2018, to 512-bit in current architectures that support the AVX512 ISA extension. A 512-bit vector register has eight 64-bit lanes, 16 32-bit lanes, or 64 8-bit lanes.

These ISA extensions were introduced primarily to improve graphics and multimedia application performance on consumer-grade processors. Graphics applications use different number of bits to indicate the color of a pixel. This value is known as the bit or color depth. Graphic applications from the 1990's mostly used 8-bit or 16-bit color. Most application in the 2010's have moved to 24-bit or 32-bit color depths. SIMD ISA extensions allow packing multiple color data types into a SIMD register, and operating on them in parallel.

Short-vector architectures differ significantly from the large vector processors like Cray-1 from 1970's and 1980's, and even from GPGPUs. Most short-vector SIMD ISA extensions do not offer conditional execution via mask registers. They also do not offer non-unit stride and gather-scatter addressing modes in hardware. This is largely due to the memory organization of modern architectures. Almost all modern DRAM memory chips are organized as a two-dimensional array of DRAM cells. Rather than addressing individual memory location, memory addresses are multiplexed into two parts,

```
1  void stencil_9pt (float * restrict A1, const float * restrict A2) {
2    for(auto t = 1ul; t < T-1; ++t)
3      for(auto z = 1ul; z < Z-1; ++z)
4        for(auto y = 1ul; y < Y-1; ++y)
5          for(auto x = 1ul; x < X-1; ++x) {
6            A1[t][z][y][x] = A2[t-1][z][y][x] + A2[t+1][z][y][x]
7                           + A2[t][z-1][y][x] + A2[t][z+1][y][x]
8                           + A2[t][z][y-1][x] + A2[t][z][y+1][x]
9                           + A2[t][z][y][x-1] + A2[t][z][y][x+1];
10         }
11   //... Elided boundary region computations
12 }
```

Listing 7.1: A nine-point scalar stencil

i.e., *row address selection* and *column address selection*. Memory is addressed first using the row address, and then the column address is decoded to access an individual element in that row. Data is also not moved as individual words, rather it is always accessed in terms of cache-line sized blocks. Due to these reasons no gather-scatter addressing at the level of words is implemented in hardware.

7.1.3 Stream alignment conflict

The stream alignment conflict (SAC) metric is an array reuse distance-based measure for detecting scenarios that need data-layout transformations. A SAC occurs when the same array element is read more than once in successive iterations of an innermost loop, and the reuse distance is less than or equal to the architectural SIMD register width. For such cases, vectorization of the innermost loop would lead to the overlapping SIMD register scenario that was illustrated in Section 2.3. The following definitions formalizes this notion.

Definition 7.1. Reuse distance (R_d)

R_d of an array element is defined as the number of other distinct array elements that are accessed between two consecutive accesses of the same array element. R_d is measured in terms of the number of memory references, and is a measure of temporal locality. △

Definition 7.2. Stream alignment conflict

A SAC exists inside a stencil kernel if there are two distinct array read accesses a_1 and a_2 that access the same array element in two different iterations i and i', and the R_d of that element is either less than or equal to the SIMD register width, or is not a multiple of the SIMD register width. △

Listing 7.1 is the same stencil used in Section 2.3. Definition 7.2 offers a way to identify the need for a data-layout transformation for this case. The R_d of each array element for the two accesses `A2[x-1]` and `A2[x+1]` is two, which is less than the SIMD vector width of eight. However, if the accesses are changed to `A2[x-1]` and `A2[x+7]` the R_d is eight, and SAC does not exist. Both these scenarios are considering the array has a default lexicographic row-major data-layout.

7.1.4 Mitigating SAC

Data-layout transformations based on our $\rho\phi$ algebra (Chapter 4) are a way to mitigate SAC. When using a $\rho\phi$ to define a SIMD vectorizable data-layout, one or more array dimensions are reshaped and then transposed to build a new innermost dimension. This dimension is of the same size as the SIMD vector width. Section 5.3 presented QUARC's ATL interface that is used for defining such transformations. The primary rationale for the transformation is to place the array elements in memory so that the R_d for any pair of array reference in a stencil kernel does not result in a SAC.

7.2 Experimental Setup

The experiments were done in phases for three different stencil kernels on two IA servers. Each phase used a different array size, and considered all possible combinations of $\rho\phi$ transformations for that array size. The applicable $\rho\phi$ transformations varied based on the rank of the array, and the architectural SIMD register width. The array size was varied to range from fully resident inside L2 cache to falling out of L2, but resident in L3 cache. All the experiments used only single threaded execution, and used single-precision floating point numbers.

The reported performance is the median execution time per stencil iteration over 20 separate runs of the experiment. For each run the stencil loop was executed 50 times, and the median time out of the 50 runs reported. This was done to negate any impact of cold cache misses. The experiments were repeated for each data-layout choice for every problem size on both architectures. Along with, execution time the total instructions count, and L1 and L2 data cache miss rate (DCM) was also measured using hardware performance counters.

7.2.1 Stencil Benchmarks

2D-Jacobi stencil computes a five-point sum for each element of a two-dimensional array. Each stencil site involved four real addition operations, and had a data footprint of 20 bytes.

3D-Jacobi stencil computes a seven-point sum for each element of a three-dimensional array. Each stencil site involved six real addition operations, and had a data footprint of 28 bytes.

Wilson-Dslash (WD) is a nine-point stencil kernel from LQCD. This is a complicated stencil that involves complex $SU(3)$ complex vector algebra at each stencil point. Each stencil site involves multiple complex matrix-vector products for a total FLOP count of 1320, and a data footprint of 1440 bytes. The stencil is described in detail in Section 9.1.1.

7.2.2 Architectures

Knight's Landing (KNL) is a single socket 68 core Intel® Xeon Phi™ CPU 7250 server with 1MB shared L2 cache. KNLs do not have an L3 cache, but have a high bandwidth memory that was configured to run in cache mode. All experiments on this server used 512-bit AVX512 vectorization.

Skylake X (SKX) is a dual-socket 24 core Intel® Xeon® Platinum 8160 CPU with 1MB private L2 cache, and a 33MB shared L3 cache. All experiments on this platform were executed with 512-bit AVX512 SIMD vectorization.

7.2.3 Data-layouts

The data-layouts for these experiments are based on $\rho\phi$ transformations. The number of data-layout for each of our test cases was as follows.

The 2D-Jacobi used arrays shapes: $\{512\times512\}$, $\{1024\times512\}$ and $\{1024\times1024\}$. These shapes allowed five data-layout choices for AVX512.

The 3D Jacobi used arrays shapes: $\{64\times64\times64\}$, $\{128\times64\times64\}$, and $\{128\times128\times64\}$. These shapes allowed us to evaluate 15 data-layout choices for AVX512.

The WD kernel has a larger data footprint per stencil site, and limits the array shape choices for smaller problem sizes. The following array shapes were used for this stencil: $\{4\times4\times8\times8\}$, $\{4\times8\times8\times8\}$, and $\{4\times8\times16\times16\}$. The reason for selecting these particular array shapes was to mimic the typical array block sizes used in production LQCD applications (Joó *et al.*, 2013). There the usual strategy is to block

the outer array dimensions over multiple threads, and use the inner dimensions for SIMD vectorization. These array shapes limited the potential number of data-layouts that could be evaluated. They allowed a mix of two-, three-, and four-dimensional data-layouts. Our layout selection policy that is discussed in Chapter 8 is independent of the array shape, and applies to selecting the best subset of data-layouts for a given set of data-layout choices. Thus, even though a limited number of data-layout choices are possible, it does not restrict us from evaluating our overall data-layout selection policy.

7.3 Results and Observations

Figure 7.1 presents the execution time distribution for the different data-layout options for each of the stencil kernels. Few trends are immediately apparent from these plots. The SIMD vectorization performance variation across different data-layouts is much more pronounced on KNL than on SKX. In fact, for the WD kernel the variation is negligible on SKX, and we saw only a difference of $\sim 10\%$ in execution time for all problem sizes. This to a certain extent relates to the fact that a relatively smaller number of layouts were tried for this kernel. All experiments on both SKX and KNL showed that most data-layouts performed well, with a small number of outliers. The outliers became prominent only at larger problem sizes. The data-layout choice also had a greater impact for larger array sizes. This was seen on both architectures, and all three kernels.

Analyzing 2D-Jacobi

Table 7.1 presents the performance for each of the five data-layouts for the 2D-Jacobi stencil for SKX and KNL for the $\{1024 \times 512\}$ and $\{1024 \times 1024\}$ array shapes. Both architectures exhibited similar variation in performance of the best and worst data-layout choice. A more interesting observation is how individual data-layouts performed. The results in this regard are almost a complete opposite of each other for each architecture for the smaller problem size. The $\{1, 16\}$ layout was one of the best choices on SKX, but was the worst layout choice on KNL. The $\{16, 1\}$ layout was one of the best choice on KNL for the smaller size, but was the worst option for SKX.

We need to recall how data-layouts are constructed using $\rho\phi$ transformations to explain these results. The $\{16, 1\}$ data-layout completely transposes the innermost array dimension, while constructing the SIMD dimension by reshaping the outer dimension. The $\{1, 16\}$ data-layout is built by only reshaping

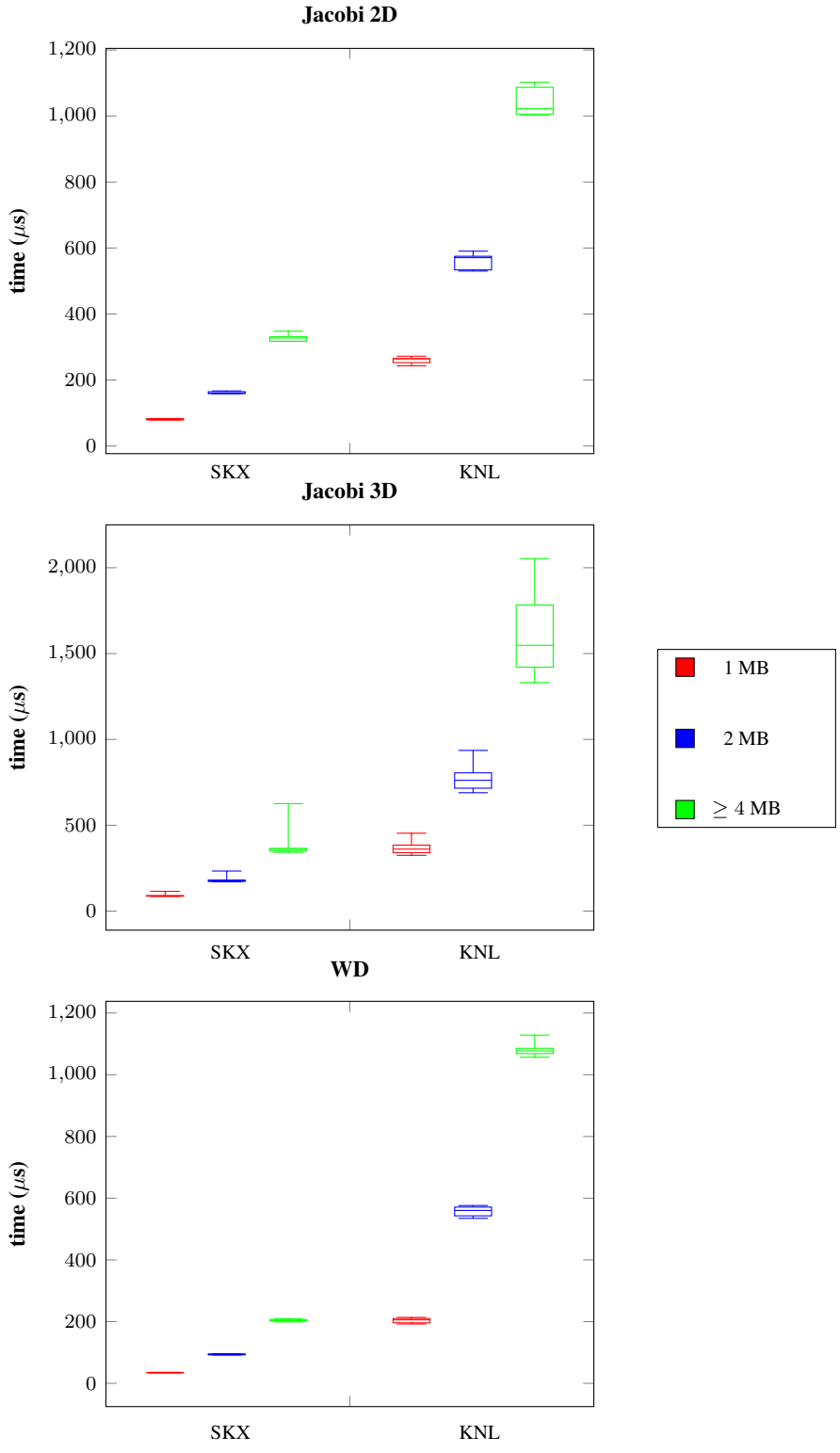


Figure 7.1: An empirical evaluation of data-layout choices based on wall clock execution time. Each boxplot shows the minimum, first quartile, median, third quartile, and maximum execution time. The minimum execution time value indicates the best data-layout choice.

Table 7.1: Evaluating all data-layout choices for 2D-Jacobi

(a) SKX

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
1,16	159	449	65.63	65.57
2,8	159	439	65.65	65.55
4,4	159	433	67.40	65.54
8,2	165	430	131.85	65.38
16,1	167	352	131.86	65.41

$\{1024\times512\}$ (2MB)

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
2,8	317	864	134.10	130.97
1,16	317	875	131.20	130.96
4,4	327	859	263.60	130.63
8,2	331	856	263.76	130.72
16,1	348	701	263.58	130.99

$\{1024\times1024\}$ (4MB)

(b) KNL

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
4,4	530	433	38.39	3.70
16,1	534	351	92.04	4.55
8,2	571	430	90.07	4.08
2,8	575	438	30.41	3.55
1,16	590	449	21.64	3.76

$\{1024\times512\}$ (2MB)

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
2,8	1003	863	76.20	68.16
4,4	1006	859	183.02	65.87
8,2	1022	856	182.12	71.28
16,1	1087	701	182.16	84.77
1,16	1103	875	60.17	63.49

$\{1024\times1024\}$ (4MB)

Table 7.2: Calculated reuse distances and shuffle instructions for the $\{1024\times512\}$ sized 2D-Jacobi

Layout	Y	X	RD_Y	RD_X	S_Y	S_X	Total Shuffles
1,16	1024	32	32	1	0	2048	2048
2,8	512	64	64	1	2	1024	1026
4,4	256	128	128	1	2	512	514
8,2	128	256	256	1	2	256	258
16,1	64	512	512	1	2	0	2

and transposing the inner dimension, without impacting the outer dimension. These transformations impact the strides in each dimension, and therefore the reuse distances. As explained in Section 6.4.4 it also impacts the number of shuffle and blend operations needed while vectorizing over a particular array data-layout.

Table 7.2 shows the reuse distances and the number of shuffles needs for the $\{1024\times512\}$ problem size. The two array dimensions are called 'Y' and 'X', with Y being the outer dimension, and X the inner dimension. RD_Y and RD_X depict the reuse distance in Y and X. Reuse distance is measured in terms of the number of vector register loads that are required between two successive accesses for the same vector. S_Y and S_X are the shuffles needed in Y and X. Note that since each vector tile has two boundaries the number of shuffles is doubled. The total number of shuffle instructions is the sum of S_Y and S_X.

Table 7.3: Evaluating the five best and five worst data-layout choices for 3D-Jacobi kernel

(a) SKX

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
1,2,8	171	682	130.82	65.55
1,4,4	171	790	131.49	65.54
1,8,2	173	755	131.42	65.55
1,16,1	173	702	129.32	65.56
2,8,1	174	743	133.66	65.59
4,2,2	180	690	131.50	65.66
8,2,1	182	730	133.44	66.20
8,1,2	186	802	131.52	66.17
4,1,4	186	850	131.55	65.64
16,1,1	233	703	132.43	95.94

$\{128\times64\times64\}$ (2MB)

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
2,8,1	342	1472	266.27	131.01
1,16,1	344	1366	266.87	131.03
1,2,8	347	1357	262.67	130.97
1,4,4	349	1575	262.93	131.05
1,1,16	351	1773	262.71	131.08
4,4,1	362	1470	266.71	131.44
4,1,4	369	1701	262.94	131.41
8,1,2	445	1604	262.65	182.80
8,2,1	445	1454	264.64	183.49
16,1,1	626	1407	263.93	261.72

$\{128\times128\times64\}$ (4MB)

(b) KNL

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
1,2,8	690	691	68.07	3.72
2,1,8	697	744	51.84	4.59
1,16,1	699	702	79.32	1.82
2,4,2	713	692	69.93	2.47
2,8,1	719	743	72.51	2.24
1,4,4	767	790	44.05	1.33
8,2,1	844	730	70.19	3.53
4,1,4	855	850	62.26	2.38
8,1,2	895	801	64.62	3.08
16,1,1	936	703	72.09	3.58

$\{128\times64\times64\}$ (2MB)

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
1,2,8	1331	1374	134.65	7.50
2,1,8	1389	1485	103.55	10.12
1,1,16	1403	1724	71.79	7.32
1,16,1	1411	1366	168.42	4.24
2,4,2	1431	1377	140.66	5.29
4,1,4	1771	1701	123.05	5.96
4,4,1	1794	1470	133.33	8.86
8,1,2	1935	1604	125.28	6.94
8,2,1	1975	1454	133.30	8.54
16,1,1	2053	1407	142.29	6.94

$\{128\times128\times64\}$ (4MB)

Both $RD_{X|Y}$ and $S_{X|Y}$ are computed based on the reshaped extents of the original array dimension. Reuse distance in a given dimension is the same as the stride in that dimension based on the reshaped extents. The number of shuffle and blends is computed as the *reverse stride* for that dimension, *i.e.*, doing the stride calculation in the opposite direction or inside out. It can be seen that the $\{1, 16\}$ data-layout results in the lowest reuse distance, while having the highest number of shuffles. It is exactly the opposite for the $\{16, 1\}$ data-layout. Our empirical measurements attest these calculations. The $\{1, 16\}$ data-layout has the highest instruction count, and the $\{16, 1\}$ layout has the minimum. Whereas, the $\{1, 16\}$ data-layout has a low L1 cache miss rate, and vice-versa for the $\{16, 1\}$ layout.

Table 7.4: Calculated reuse distances and shuffle instructions for the $\{128\times64\times64\}$ sized 3D-Jacobi

Layout	Z	Y	X	RD_Z	RD_Y	RD_X	S_Z	S_Y	S_X	Total Shuffles
1,2,8	128	32	8	256	8	1	0	256	8192	8448
1,4,4	128	32	16	512	16	1	0	256	8192	8448
1,8,2	128	8	32	256	32	1	0	256	2048	2304
1,16,1	128	4	64	256	64	1	0	256	0	256
2,8,1	64	16	64	1024	64	1	2	128	0	130
4,2,2	32	64	32	2048	32	1	2	64	4096	4162
8,2,1	16	64	64	4096	64	1	2	32	0	34
8,1,2	16	64	32	2048	32	1	2	0	2048	2050
4,1,4	32	64	16	1024	16	1	2	0	4096	4098
16,1,1	8	128	64	8192	64	1	2	0	0	2

Analyzing 3D-Jacobi and WD

Both 3D-Jacobi and WD are higher order stencils, and due to this reason, the effect of data-layouts is slightly different from the 2D-Jacobi case. Table 7.3 presents the best and worst five data-layout choices for 3D-Jacobi for the $\{128\times64\times64\}$, and $\{128\times128\times64\}$ shaped arrays. The worst performing data-layouts are in the rows highlighted in gray. The data shows that on both SKX and KNL data-layouts constructed by reshaping the outermost dimension by a large factor leads to the worst performance. This once again is due to the impact of these data-layouts on the reuse distance. All the worst performing data-layouts have the largest reuse distance in the outermost dimension. The best layouts all have small reuse distances for the outer dimension. This can be seen in Table 7.4 that presents the reuse distances for the data-layouts on SKX for the $\{128\times64\times64\}$ case. The convention followed to name the dimensions is the same as before, with 'Z' introduced to represent the outermost array dimension. The number of shuffles and blend operations has a much lesser impact than in the 2D-Jacobi case.

Along with reuse distance and shuffle operations there is a tertiary factor that impacts data-layout performance for higher dimensional arrays and higher order stencils. We term this as "edge" reuse. The vectors at the edges of every vector tile have two types of reuse. The first comes when the vector is accessed in a subsequent loop iteration with an aligned vector load. This reuse is the basis of our reuse distance calculations. The vectors at the edges of the tiles are reused once more in the shuffle operations needed to handle the internal vector tile boundary. The edges for multi-dimensional vector tiles are longer for higher dimensional arrays. This leads to a performance improvement for cases where the edge vector tile is resident in L2 cache for both its reuses. This is the reason the data-layouts with the Z dimension transformed by a factor of two performed well on both architectures.

Table 7.5: Evaluating the five best and five worst data-layout choices for WD kernel

(a) SKX

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
1,1,4,4	92	366	40.45	20.42
1,4,4,1	92	367	41.40	20.12
2,2,4,1	92	367	41.16	20.18
2,4,2,1	92	363	41.21	19.89
2,2,1,4	93	364	41.35	20.11
2,1,4,2	94	358	42.84	20.23
2,1,2,4	95	363	41.68	20.30
1,2,2,4	96	362	44.31	20.15
2,2,2,2	96	360	46.14	20.22
1,4,2,2	96	359	46.19	20.40

$\{4\times8\times8\times8\}$ (2MB)

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
2,1,1,8	410	1417	165.33	110.98
2,1,8,1	413	1441	164.59	110.60
2,1,4,2	427	1413	190.04	110.61
2,1,2,4	428	1420	183.17	110.63
2,4,2,1	429	1443	164.33	123.75
1,1,8,2	441	1422	187.20	120.91
1,4,1,4	441	1440	185.20	120.85
1,4,2,2	442	1419	189.36	119.54
1,2,2,4	446	1425	195.16	117.80
1,1,4,4	450	1425	195.55	121.15

$\{4\times8\times16\times16\}$ (12MB)

(b) KNL

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
1,2,2,4	535	365	29.58	13.56
1,4,1,4	535	367	27.66	12.98
2,4,2,1	540	362	28.24	12.22
1,4,2,2	545	358	31.30	13.23
1,4,4,1	552	369	28.03	13.29
1,1,4,4	564	364	26.01	13.63
2,2,2,2	568	361	30.99	12.20
2,1,2,4	574	363	27.22	13.37
2,2,4,1	574	363	27.99	12.57
2,2,1,4	576	365	27.69	12.41

$\{4\times8\times8\times8\}$ (2MB)

Layout	Exec. Time (μs)	Total Inst. ($\times 10^3$)	L1 DCM ($\times 10^3$)	L2 DCM ($\times 10^3$)
2,4,2,1	2272	1436	113.47	55.67
2,2,4,1	2351	1429	119.29	58.91
2,1,2,4	2359	1422	121.38	57.03
2,2,2,2	2362	1429	126.33	58.13
1,4,2,2	2403	1418	128.22	62.68
1,4,1,4	2534	1440	123.35	64.85
1,2,1,8	2535	1425	118.19	64.52
1,1,2,8	2563	1413	115.22	63.97
1,1,4,4	2573	1423	128.19	62.94
1,1,8,2	2584	1420	124.07	63.94

$\{4\times8\times16\times16\}$ (12MB)

Table 7.5 presents the performance results for the five best and five worst data-layouts for the WD kernel. The table shows the results for the $\{4\times8\times8\times8\}$ and $\{4\times8\times16\times16\}$ array shapes. The four-dimensional WD stencil has a much larger data footprint than the Jacobi stencils. The edge reuse is highly beneficial for this kernel, especially for larger problem sizes. Each of the best data-layout options for the 12 MB problem size involved transforming the outermost dimension. Within that set of layouts the relative reuse distances of the inner dimension determined the performance of a layout. The smaller problem size also benefits from edge reuse, but not to the same extent. The reuse distance plays a more important role, especially on KNL.

Chapter Review

This performance analysis presented a set of heuristics that allow evaluating $\rho\phi$ data-layout transformations for stencil kernels. Data-layouts formed using these transformations are impacted by three factors: reuse distance, number of shuffle and blend operations, and edge reuse. How these three factors interplay with each other depends on the type of stencil, the problem size and the architectural characteristics. Each SKX core is much faster than a single KNL core. This is the reason the number of shuffles has a lesser impact on SKX than on KNL. However, for most cases cache performance played a much bigger role than the instruction counts.

This analysis helps provide an empirical basis for the intuitions behind the data-layout selections in hand vectorized libraries, such as the QPhiX library (Joó *et al.*, 2013) from LQCD. Due to the complexity of hand vectorization and data-layout transformation, most hand vectorized libraries select a very small set of data-layouts. As an example, QPhiX does not support building three or four-dimensional vector tiles, and is restricted to one- and two-dimensional vector tiles. Our analysis shows that there are other data-layout options that can potentially perform better. Having our results as a reference should enable library writers make better choices regarding data-layouts for SIMD vectorization.

Finally, as a side benefit this analysis demonstrates one of the big advantages of QUARC's automated data-layout based SIMD vectorization. Exploring the complete space of data-layouts by hand is not feasible especially for high-dimensional arrays on architectures with relatively longer SIMD widths. QUARC makes this extremely easy, and can be the used by auto-tuners or even manual analysis. The results of such analysis can be used to define policies, making the data-layout selection and code-generation process as much automated as possible. Chapter 8 describes a possible implementation of such a policy.

CHAPTER 8: DATA-LAYOUT SELECTION POLICY

QOPT's SIMD vectorizer requires arrays to have a data-layout defined using a $\rho\phi$ transformation. The transformation reshapes one or more outer array dimensions, and then transposes them inwards to build an innermost SIMD dimension. Previous chapters provide the specifics of the transformation (Chapter 4), its implementation in ATL (Chapter 5), and the QOPT's ahead of time speculative SIMD vectorization process (Chapter 6). This chapter discusses the policy used to select the set of data-layout choices based on which speculative vectorization is done. The policy currently is implemented outside of QOPT. The chapter focuses only on the policy, and not on its implementation mechanism.

8.1 Performance Effects of Data-Layouts

The data-layout selection policy is tailored specifically for stencil kernels based on the empirical performance evaluation in Chapter 7.

The primary performance effect of a layout transformation is the removal of stream alignment conflicts. Stream alignment conflicts reduce vectorization efficiency by requiring extra unaligned loads, shuffles, blends and permute vector instructions. A data-layout transformation performs a gather-scatter redistribution of an array's elements to ensure that most accesses in a stencil loop are aligned vector loads and do not need shuffles or blends operations. Apart from this primary effect, a data-layout has a second order effect that impacts performance. A $\rho\phi$ data-layout transformation can be equated to multi-dimensional *array tiling*. Tiling impacts the stride of the array in each dimension, and thereby affects the reuse distances for array accesses. The results in Section 7.3 demonstrated the correlation between vector reuse distance and the L1 and L2 cache miss rates. Thus, how a data-layout impacts reuse distance impacts the overall performance.

Although, a $\rho\phi$ data-layout transformation removes most unaligned vector loads, shuffle and blend operations it cannot remove all shuffles operations for stencil kernels. Shuffle operations are required to handle vector loads at the tile boundaries. The number of such shuffle operations is dependent on the

Procedure selectLayout

Input: Architecture A
Input: Block shape B_s
Input: Element size es
Input: Stencil extents S_{exts}
Input: Number of layouts N_l
Output: Set of $\rho\phi$ factors L

$N_d \leftarrow$ number of array dimension;
$D_f \leftarrow$ data footprint of the array block;
Generate set of all data-layout candidates for B_s for the target SIMD width.;
Eliminate all layouts that violate Constraint (6.3);
Compute reuse distance (R_d) and number of shuffle operations (N_s) for each layout;
if *L2 cache size* $< D_f \leq 8\times$ *L2 cache size and* $N_d > 3$ **then**
    Add to L all layouts with outermost reshape factor of 2;
end
if L *is empty* **then**
    Add all viable layouts to L;
end
Sort layouts in L based on smallest R_d starting from the outermost dimension;
Retain top N members in L;
if *A is KNL* **then**
    Sort the layouts based on smallest N_s;
end
return L;

Figure 8.1: Steps to select a set of data-layout candidates for a stencil kernel.

number of array dimensions that are transformed. The number of shuffle operations too has an impact on the SIMD vectorization performance, and is an important third order effect to be consider.

Finally, for a limited number of cases involving four-dimensional arrays there is an effect that is termed as the edge reuse effect. Four-dimensional arrays used for lattice and grid computation typically have larger data footprints, but the extents for each dimension is small. The discussion in Section 7.3 touched upon the two types of reuse for boundary vector tiles. Apart from the reuse for regular aligned loads, boundary tiles are also reused in the shuffle operations. Multi-dimensional vector tiling involving the outermost dimension is useful for such cases, especially if the number of tiles is small. Such cases benefit from improved reuses of the boundary tiles for the shuffle operations.

8.2 Policy Input Parameters

Architecture

The policy is designed for two types of Intel Architectures, server-class Xeon processors and the Intel Knight's Landing (KNL) Xeon-Phi processor. Apart from deciding which data-layouts to select, the architecture also decides the required SIMD width.

Array block shape

The data-layout choices are selected for a specific block shape that specifies the extents of each array dimension.

Element size

The policy requires the size in bytes of each array element to compute the data footprint of the array block.

Stencil extents

The policy requires the stencil extents in every cardinal direction to compute the reuse distances, and number of shuffle operations.

Number of layouts

The number of layouts selected by the policy.

8.3 Policy Execution Steps

Figure 8.1 describes the functioning of the data-layout selection policy. The policy starts by exhaustively generating all possible data-layout candidates based on SIMD register width of the target architecture. This set of data-layouts is then pruned based on Constraint (6.3) defined in Section 6.4.4. The constraint ensures that a data-layout is free of stream alignment conflicts, and the shifted accesses can be translated to the transformed layout without any division or modulo operations.

An additional step is included for four-dimensional arrays specific to lattice and grid based kernels. The policy looks for data-layouts that define the vector tile by factoring the outermost dimension. The search is restricted to the smallest possible reshape factor, and cases where the boundary tiles fit inside L2 cache. The current policy only considers array blocks that are at up to eight times the L2 cache size. Beyond that the vector tiles are large enough that no benefit is seen from edge reuse.

After the initial set of layouts is identified reuse distances and the number of shuffle and blend operations are computed for each layout. The layouts are then ranked based on the ascending order of reuse distances.

An extra step is done to optimize the data-layout selection for KNLs; the selected data-layouts are further sorted based on the number of shuffle instructions. This ensures that the policy gives preferences to data-layouts that have lower total shuffle operations on KNLs.

8.4 Evaluating the Policy

The policy is evaluated by comparing it to the empirical results obtained in Chapter 7. The policy was effective on both architectures for the 2D-Jacobi example. It was always able to select the best three layouts for both problem sizes resident in L2 cache, and larger sizes that did not fully fit in the L2 cache. An accuracy of 66% was achieved for the 3D-Jacobi kernel when selecting the best three options on the test Intel Skylake server. However, the accuracy rate improved to 80% when top five data-layout choices were selected. The policy performed similarly for the same kernel on the KNL server, but there were some outlier array shapes for which the policy accuracy dropped to 60% for best five layouts. It is possible that these particular array shapes led to higher conflict cache misses. The policy proved effective for the Wilson Dslash LQCD stencil as well for both Skylake and KNL. The accuracy rate was again up to 80% on Skylake. On KNL the policy worked best for problem sizes resident in L2 cache, and had an accuracy of 80%. Data-layout selection for large problem sizes was not as effective, and the policy could identify only one out of the top three data-layouts. Even then the data-layouts identified by the policy were within the third quartile of all layout choices.

Overall, using the simple heuristics identified in Chapter 7 the policy proved highly effective in being able to select data-layouts for the tested stencil kernels. It is possible that the policy can be further

fine-tuned for more problem sizes and array shapes, but the main performance effects we highlighted should remain effective for selecting data-layouts using $\rho\phi$ data-layout transformations.

CHAPTER 9: QUICQ: A QUARC-BASED LQCD DSL

We now describe the construction of a prototype EDSL using QUARC. The target application domain for the EDSL is Lattice Quantum Chromodynamics (LQCD), which is one of the original United States government HPC grand challenges (Interagency Working Group on Information Technology Research and Development, 2006). LQCD simulations are part of both high-energy physics and nuclear physics research, and are one of the largest consumers of computation cycles on United States Department of Energy leadership class machines. Section 9.1 introduces the domain of LQCD and its basic operators. The information in Section 9.1 paraphrases discussions with LQCD researchers, Balint Joó (private communication, Joó, 2014) and Carleton DeTar (private communication, DeTar, 2013).

The prototype EDSL, QUICQ Internally Calls QUARC (QUICQ), demonstrates a highly expressive and notation programming interface using LQCD-specific abstractions. Section 9.2 presents QUICQ's various code components including domain-specific array data types, whole-array operators, and mkernels. QUICQ is not yet a complete EDSL for LQCD, but it is powerful enough to write iterative solvers that can be deployed at scale on large cluster computers. This chapter only presents QUICQ's implementation details, performance evaluation results are in Chapter 10.

9.1 Lattice Quantum Chromodynamics (LQCD)

Quantum Chromodynamics (QCD) is a branch of theoretical physics that deals with the study of the strong force, which is one of the four fundamental forces in nature. The strong force, or strong interaction, describes the interactions between the elementary particles: quark and gluon. Quark and gluons make up the nucleons, *i.e.* protons, neutrons, and mesons. QCD is analogous to quantum electrodynamics. Just as photons are the charge carriers in quantum electrodynamics, gluons are the charge carriers in QCD. Instead of carrying an electrical charge, gluons carry what is called a *color charge*. Color charge has no correlation to the visual perception of color, and is merely a notation used to define the quark-gluon interaction.

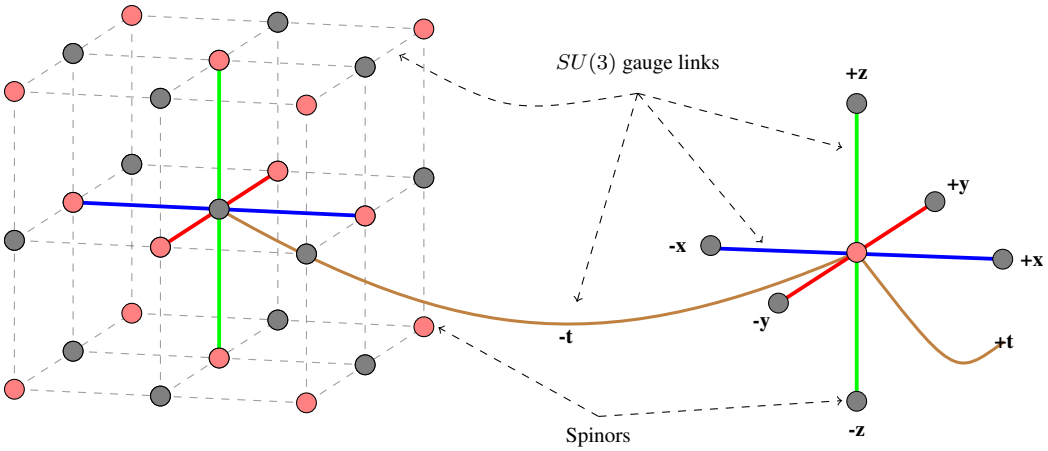


Figure 9.1: An illustrative four-dimensional even-odd preconditioned LQCD lattice.

The strong force is highly non-linear; therefore, it is extremely hard to formulate analytical solutions for low-energy QCD. Instead, an alternative non-perturbative approach, called Lattice QCD (LQCD), is used to simulate the interactions on a discretized four-dimensional space-time lattice. The nodes, or sites, of the lattice represent quarks and the connecting links represent gluons. Figure 9.1 shows an example of a LQCD lattice.

The Dirac equation (Dirac, 1928) is fundamental to all LQCD simulations. Repeated solving of the Dirac equation is the most computationally expensive piece of LQCD simulations. In LQCD, the Dirac equation is discretized as a sparse matrix equation

$$M\psi = \chi, \tag{9.1}$$

where ψ, and χ are quarks, and M is the Dirac matrix. The equation solves ψ for a given χ. The Dirac equation is a partial differential equation, and the M matrix represents the discretized partial differential operators. Solving Equation (9.1) involves repeated inversion of M. The Dirac equation is solved with varying RHS values, and different Dirac matrix configurations. We refer readers to more topical material (Creutz, 1987) for further details.

9.1.1 The Wilson Dslash operator

The Wilson Dslash (WD) operator is one of the discretized differential operators used to solve the Dirac equation. The locality of this operator connects each lattice site with its neighbor in the eight

cardinal directions. The WD operator uses four-dimensional complex vectors to represent quarks. These complex vectors are known as *spinors*. Gluons are represented by three-dimension complex matrices that belong to the $SU(3)$ unitary group. The $SU(3)$ matrices comprise a *gauge field*. The full vector space for WD is the tensor product of four different vector spaces that results in a 12-dimensional complex vector space.

Spin space (S)

S is a four-dimensional complex vector space. The spin space is used to define four 4×4 hermitian γ matrices that are used by the WD operator. The γ matrices are used to project and elongate spinors between the four-dimensional spin space and a two-dimensional subspace.

$$\gamma_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \gamma_2 = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix},$$

$$\gamma_3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \gamma_4 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}$$

Color space (C)

C is a three-dimensional complex vector space that defines the color charge. The three color components of a spinor are usually called as red, blue or green for quark particles. Antiquark particles have the corresponding antired, antiblue, and antigreen charge.

Reality space (R)

R is the two-dimensional vector space of complex numbers used in spinors and gauges.

Lattice space (V)

V defines a regular grid or lattice. Its size is defined by the product of the dimensions of each of the four components of the lattice space, *i.e.* the product of the extents of the lattice in x, y, z and t Cartesian directions. The overall size of V defines the total number of lattice points. Therefore, any generic point on the lattice can be denoted using four coordinates (x, y, z, t).

The full complex vector space of WD is:

$$W \equiv S \otimes C \otimes R \otimes V \equiv S \otimes C \otimes R \otimes X \otimes Y \otimes Z \otimes T \tag{9.2}$$

The sparse matrix M to solve the Dirac equation using WD can be represented as

$$M = (N_d + m) - \frac{1}{2} D_w, \tag{9.3}$$

where m is the quark real mass parameter, and D is the WD operator.

$$D_w = \sum_{\mu=1}^{4} ((I - \gamma_\mu) \otimes U_x^\mu \delta_{x+\hat{\mu},x'} + (I + \gamma_\mu) \otimes U_{x-\hat{\mu}}^{\mu\dagger} \delta_{x-\hat{\mu},x'}). \tag{9.4}$$

Equation (9.4) shows the summation of the $SU(3)$ 3 matrix-vector product of gauges and spinors at each lattice site. U_x^μ represents the gauge emanating from site x in direction μ. The $\delta_{x+\hat{\mu},x'}$ Kronecker delta notation corresponds to the neighboring spinor in the forward or backward μ direction. The $(I \pm \gamma)$ terms are projector operators acting on spin indices of spinors. Spin projection projects out the lower two spin components to reduce the number of spin components to two.

A numerical property of the $SU(3)$ gauge matrices is that

$$U_x^\mu \equiv U_{x'}^{\mu\dagger}, \tag{9.5}$$

i.e., the gauge matrix between the sites x and one of its neighboring sites in a cardinal direction x' is the conjugate transpose of the gauge matrix between x' and x. The property reduces the storage requirements for the gauge matrices, and improving the computational properties of WD.

```
Procedure WilsonDslashEO
    Input: Odd half-lattice
    Output: Even half-lattice
1   foreach each site in the half-lattice do
2       foreach each of the eight neighboring sites do
            // 24 real values
3           Gather neighboring spinor (n_s);
            // 12 flops
4           Project out lower two spin components of n_s;
            // 18 real values
5           Gather gauge matrix for the cardinal direction;
            // 2×66 flops
6           Matrix multiplication of gauge and spinor for both spin components ;
7           Reconstruct the lower two spin components of n_s;
8       end
        // 7×24 flops
9       Aggregate matrix multiplication results at output site;
10  end
```

Figure 9.2: High-level structure of the Wilson Dslash operator

The lattice is preconditioned into even and odd half-lattices before applying WD. Such a preconditioning step splits the lattice into two separately stored half lattices based on the parity of each site. After preconditioning the Schur complement system is solved for one half lattice at a time.

Computational profile of the WD operator

Figure 9.2 shows the high-level structure of the WD operator for one half lattice where the odd half lattice is the input and the even half lattice is the output. The computation at each site needs eight neighboring spinors, and the eight $SU(3)$ gauge matrices. Every spinor is a 12-dimensional complex vector requiring 24 real numbers. An $SU(3)$ matrix for a gauge is a 3×3 complex matrix requiring 18 real numbers. The per-site data footprint constitutes of 8×24 real numbers for the spinors, 8×18 for the gauge matrices, and 24 real numbers for the output spinor. The total requirement is 360 real numbers that in single precision translates into 1440 bytes, and in double precision is 2880 bytes.

The WD operations consists of an initial *spin projection* step for each of the eight neighboring spinor. A projection operation on a spinor involves two complex additions per color component, *i.e.*, six complex additions or 12 floating point operations (flops). Here addition includes both add and subtract operations. Spin projection projects out the lower two spin components of the spinor. After spin projecting, the

residual color components are multiplied with the gauge matrix. The gauge-spinor product consists of three complex inner products with each inner product involving three complex multiplications and two complex additions. Thus, the number of flops for a gauge-spinor product are 66 flops. Since, both color components are multiplied with the gauge the total flops are 2×66, *i.e.*, 132 flops. After the gauge-spinor multiplication, the projected out spinors are reconstructed. Reconstruction of the lower two spin components only involves multiplication by the $-i$, i, 1, or -1, and can be done without expending any flops. The final step is aggregating the intermediate results into the output spinor that involves seven spinor additions, or 7×24 additions. From this breakdown, the total flop rating for WD is 1320 flops.

Arithmetic Intensity (AI) or the flop-to-byte ratio of WD per lattice site is $1320/1440 = 0.92$ in single precision and 0.46 in double precision. The relatively low AI makes WD a memory bound operation.

9.1.2 The Kogut-Susskind Dslash operator

The Kogut-Susskind Dslash (KSD) or the staggered Dslash operator is another formulation of the Dslash operator. KSD uses a different type of spinor that has only one spin component. The locality of KSD connects each site with its immediate neighbors and its third neighbors in each cardinal direction.

The vector space of KSD is a three-dimensional complex space.

$$KS \equiv C \otimes R \otimes V \equiv C \otimes R \otimes X \otimes Y \otimes Z \otimes T \tag{9.6}$$

The notation is the same as the WD operator. As KSD uses only one spin component, the S spin space is not required. The full discretized KSD differential operator is represented as:

$$D_{ks} = \sum_{\mu=1}^{4}((U_x^\mu \delta_{x+\hat{\mu},x'} + U_{x-\hat{\mu}}^{\mu\dagger}\delta_{x-\hat{\mu},x'}) + (U3_x^\mu \delta_{x+3\hat{\mu},x'} + U3_{x-\hat{\mu}}^{\mu\dagger}\delta_{x-3\hat{\mu},x'}). \tag{9.7}$$

As with WD, in Equation (9.7) U_x^μ represents the gauge emanating from site x in direction μ, and the Kronecker delta terms corresponds to the neighboring spinor in the forward or backward μ direction. $U3_x^\mu$ represents the gauge matrices that represent the edge or link between a site and its third neighbor in the direction μ. The gauges, U_x^μ and $U3_x^\mu$ are constructed differently, but the calculations are otherwise

Procedure KSDslashEO

Input: Odd half-lattice

Output: Even half-lattice

```
1  foreach each site in the half-lattice do
2      foreach each of the 16 neighboring sites do
           // 6 real values
3          Gather neighboring spinor (n_s);
           // 18 real value
4          Gather gauge matrix (U) for the first neighbor;
           // 66 flops
5          Matrix multiplication of U and (n_s);
           // 18 real value
6          Gather gauge matrix (U3) for the third neighbor;
           // 66 flops
7          Matrix multiplication of (U3) and (n_s);
8      end
       // 15×6 flops
9      Aggregate matrix multiplication results at output site;
10 end
```

Figure 9.3: High-level structure of the KS Dslash operator

the same. KSD does not involve any spin projection and reconstruction step. Even-odd preconditioning is also used for KSD.

Computational profile of the KSD operator

Figure 9.3 shows the high-level structure of the KSD operator for one half lattice where the odd half lattice is the input and the even half lattice is the output. The gauge basic data types are similar to WD and are 3×3 complex matrix requiring 18 real numbers. KSD uses three-dimensional complex spinors that require six real numbers. Since, KSD's locality spans the first and third neighbors its per-site data footprint is higher. Each site requires 16×6 real numbers for the spinors, 16×18 for the gauge matrices, and six real numbers for the output spinor. The total requirement is 390 real numbers that in single precision translates into 1560 bytes, and in double precision is 3120 bytes.

The numerical operations involve 16 gauge-spinor multiplications. Using the same calculation as WD, this translates into 1056 flops. The final aggregation step requires 15×6 additions, bringing the total flop count to 1146 flops per site.

Arithmetic Intensity (AI) or the flop to byte ratio of KSD is $1146/1560$ in single precision 0.73 and 0.37 in double precision.

9.1.3 Linear Solvers

Non-stationary iterative Krylov solvers, such as conjugate Gradients (CG) (Hestenes and Stiefel, 1952) and biconjugate gradient stabilized method (BiCGStab) (van der Vorst, 1992), are typically used to apply Dslash operators when solving the Dirac equation. Even-odd preconditioning is almost always used to accelerate the solution discovery amd reduce the memory footprint. Additionally, techniques such as compression of $SU(3)$ matrices (Clark *et al.*, 2010) and preconditioning at reduced precision are sometimes also applied to reduce the memory bandwidth requirements. The linear solver accounts for 80-99% of the execution time of most LQCD simulation. Apart from iterative solvers, recent approaches have explored using adaptive geometric multigrid methods with impressive performance gains (Clark *et al.*, 2016).

9.2 The QUICQ DSL

All EDSLs developed using QUARC have to provide three main code components: domain-specific data types created using METAL's array types (Section 5.1.2.2), whole-array operators to build data parallel array expressions, and the elemental functions or mkernels (Section 5.1.2.3). The following subsections present these components for the QUICQ EDSL.

LQCD Data Types

QUICQ supports both the Dslash operators described in Section 9.1. It includes custom nested array types to represent both Wilson and KS spinor types. Listing 9.1 shows the type definitions for these spinor types. Both spinor types are constructed by nesting METAL `sdlarray`.

The `su3matrix` type is for gauge matrices, and is also constructed using `sdlarray`. QUICQ uses an additional data type called `packedGauges` that is a collection of the eight gauge matrices emanating from each lattice site. The collection is used for cases where all neighboring gauges are *pre-gathered* prior to use, removing the need to gather the gauge matrices prior to their use in a Dslash operator. Finally, an `mddarray` type is defined for both spinor type, and the `packedGauges` type. These `mddarray` types are used to define the overall LQCD lattice data types that are shown in Listing 9.2.

Listing 9.2 shows two LQCD lattice data types. The listing defines two even-odd preconditioned lattices of KS, and Wilson spinors. We have skipped the constructor and destructor calls for the classes.

```
using namespace quarc::metal;

/// Complex number
using qcomplex            = sdlarray<TYPE, 2>;
/// KS spinors
using su3vector           = sdlarray<qcomplex, 3>;
/// Wilson spinors
using wilson_vector       = sdlarray<su3vector, 4>;
using half_wilson_vector  = sdlarray<su3vector, 2>;
/// A 3x3 SU(3) matrix formed out of three su3vector
using su3matrix           = sdlarray<su3vector, 3>;
/// A packed array of eight SU3 matrices
using packedGauges        = sdlarray<su3matrix, 8>;
/// Global shape representing a 4D lattice
using Latt4D              = global_shape<4>;
/// Mddarray containers representing different types of lattices
using ksSpinorLattice     = mddarray<su3vector, Latt4D>;
using wilsonSpinorLattice = mddarray<wilson_vector, Latt4D>;
using gaugeLattice        = mddarray<packedGauges, Latt4D>;
```

Listing 9.1: Defining LQCD data types in QUICQ

```
/// Even-odd preconditioned KS lattice
class EvenOddKSLattice
{
  ksSpinorLattice EO_KSSpinors[2];
  /// packged gauges for the first neighboring sites
  gaugeLattice    EO_FatLinks[2];
  /// packged gauges for the third neighboring sites
  gaugeLattice    EO_LongLinks[2];
};

/// Even-odd preconditioned Wilson lattice
class EOWilsonLattice
{
  ksSpinorLattice EO_WilsonSpinors[2];
  gaugeLattice    EO_Links[2];
};
```

Listing 9.2: Lattice data types

Notice that by default QUICQ's uses an SOA layout for a LQCD lattice. As mentioned in Section 9.1, each LQCD lattice site has a spinor, and eight gauges representing the edges between that site and its neighboring sites in each cardinal direction. Conceptually, a lattice is an array of sites, and each site represented as a `struct`. Several LQCD applications, like MILC (MILC collaboration, 1992), use such a representation. In QUICQ, every lattice is stored using at least two arrays an array of spinors and an array of packed gauges. The use of two separate arrays for spinors and packed gauges aids SIMD vectorization, and reduces the need for strided gathers when moving neighboring spinors between MPI ranks. The even and odd halves are kept separately in two different `mddarrays`.

Lattice Operators

Lattice operators operate on lattice data types. These are whole-array operations that each encapsulate an mkernel function. Listing 9.3 shows the four lattice operators that are needed to implement the KS Dslash operator as a METAL array expression. Each operator uses METAL's expression factory functions to define an array expression. C++14's Substitution Failure Is Not An Error (SFINAE) functionality is used to ensure an operator works for only a specified type of operands.

Each lattice operator builds either a binary or a unary array expression. For binary expressions, the operator needs to specify the types of the two input operands, the output type, and the mkernel function that is called back when evaluating the expression. For example, the `operator*` is defined to accept an `mddarray` of `packedGauges` and and an `mddarray` of `ksSpinors`, while returning an `mddarray` of `ksSpinors`. The operator calls the `su3mult_op` mkernel function. The unary operators have similar semantics, but require only one input operand type.

Mkernels

An mkernel function (Section 5.1.2.3) is an elemental operation that gets called for each `mddarray` element. The mkernels provided by QUICQ define the base $SU(3)$ algebra used in LQCD. Figure 9.4 presents the mkernels that are used in QUICQ's KS Dslash implementation. These represent different $SU(3)$ operations, and can be viewed as short matrix-vector or vector-vector operations. During code-generation mkernel functions get fully inlined at their call sites. For this reason, METAL requires that mkernel functions are defined with the C++ `static inline` qualifiers.

KS Dslash Implementation

With the lattice operators defined in Listing 9.3 the KS Dslash operator is implemented as a single array assignment statement. Listing 9.8 presents QUICQ's implementation of the 17-point stencil kernel representing the KS Dslash operator. As evident from this implementation, the kernel is implemented using very few lines of code.

```
/// Builds a binary expression encapsulating SU(3) matrix-vector product.
template < typename T1, typename T2,
           typename std::enable_if <
             std::is_base_of<base_expression, T1>::value &&
             std::is_base_of<base_expression, T2>::value &&
             std::is_same<su3vector, typename T2::value_type>::value
          >::type* = nullptr >
const auto& operator* (const T1 & r1, const T2 & r2)
{
  return
  expr_factory::binary_expr_builder <
    T1, T2, typename T2::value_type, su3mult
  > (r1, r2);
}
/// Builds a binary expression encapsulating addition of two SU(3) vectors.
template < typename T1, typename T2,
           typename std::enable_if <
             std::is_base_of<base_expression, T1>::value &&
             std::is_base_of<base_expression, T2>::value &&
             std::is_same<su3vector, typename T2::value_type>::value
           >::type* = nullptr >
const auto& operator+ (const T1 & r1, const T2 & r2)
{
  return
  expr_factory::binary_expr_builder <
    T1, T2, typename T2::value_type, su3add
  > (r1, r2);
}
/// Builds a binary expression encapsulating subtraction of two SU(3) vectors.
template < typename T1, typename T2,
           typename std::enable_if <
             std::is_base_of<base_expression, T1>::value &&
             std::is_base_of<base_expression, T2>::value &&
             std::is_same<su3vector, typename T2::value_type>::value
           >::type* = nullptr >
const auto& operator- (const T1 & r1, const T2 & r2)
{
  return
  expr_factory::binary_expr_builder <
    T1, T2, typename T2::value_type, su3sub
  > (r1, r2);
}
/// Builds a unary expression for conjugate transpose of SU(3) matrix
template < typename T1,
           typename std::enable_if <
             std::is_base_of<base_expression, T1>::value
           >::type* = nullptr >
const auto& adjoint (const T1 & r1)
{
  return
  expr_factory::unary_expr_builder < T1, su3matrix, adj > (r1);
}
```

Listing 9.3: KS Lattice operators in QUICQ

```
/// Add two SU(3) vectors.
static inline auto
su3add (su3vector a, su3vector b) {
  su3vector r;
  for(auto i = 0ul; i < 3; ++i) {
    r[i][0] = a[i][0] + b[i][0];
    r[i][1] = a[i][1] + b[i][1];
  }
  return r;
}
```

Listing (9.4) Adding two su3vectors

```
/// Subtract two SU(3) vectors.
static inline auto
su3sub (su3vector a, su3vector b) {
  su3vector r;
  for(auto i = 0ul; i < 3; ++i) {
    r[i][0] = a[i][0] - b[i][0];
    r[i][1] = a[i][1] - b[i][1];
  }
  return r;
}
```

Listing (9.5) Subtracting two su3vectors

```
/// SU(3) matrix-vector multiplication.
static inline auto
su3mul (su3matrix m, su3vector v) {
  su3vector r = {0.0};
  for(auto i = 0ul; i < 3; ++i)
  for(auto j = 0ul; j < 3; ++j) {
    r[i][0] += m[i][j][0] * v[j][0];
    r[i][0] -= m[i][j][1] * v[j][1];
    r[i][1] += m[i][j][0] * v[j][1];
    r[i][1] += m[i][j][1] * v[j][0];
  }
  return r;
}
```

Listing (9.6) Multiply su3matrix by su3vector

```
/// Conjugate transpose of an SU(3) matrix.
static inline auto
adj (su3matrix m) {
  su3matrix r;
  for(auto i = 0ul; i < 3; ++i)
    for(auto j = 0ul; j < 3; ++j) {
      r[i][j][0] =  m[j][i][0];
      r[i][j][1] = -m[j][i][1];
  }

  return r;

}
```

Listing (9.7) Conjugate transpose of an su3matrix

Figure 9.4: Mkernel functions needed for the KS Dslash operator

Implementation BLAS Level 1 routines

Building Krylov subspace solvers, such as the conjugate gradient (CG) method, requires various BLAS Level 1 routines. Listing 9.9 shows excerpts from a full CG solver that solves the Dirac equation using the KS Dslash operator. The listing shows various BLAS Level 1 routines applied to `ksSpinorLattice` instances. QUICQ makes it exceedingly easy to write these routines, and all of them can be written as succinct one-line array expressions.

Chapter Review

The implementation of QUICQ demonstrates the productivity advantage accrued by using QUARC. The implementation of the KS Dslash 17-point stencil kernel took less than 400 lines of code including all lattice operators and mkernel functions. A complete conjugate gradient solver on top of the KS Dslash

```
/// KS Dslash operator that updates one half of an even-odd lattice
void
__attribute__((always_inline))
ks_dslash_eo (const ksSpinorLattice &in, ksSpinorLattice &out,
              const gaugeLattice &fl, const gaugeLattice &ll)
{
  out = DRILL<0>(fl)           * in.GSHIFT<1,0,0,0>()
      + DRILL<1>(fl)           * in.GSHIFT<0,1,0,0>()
      + DRILL<2>(fl)           * in.GSHIFT<0,0,1,0>()
      + DRILL<3>(fl)           * IF_EVEN_CHOOSE(in.GSHIFT<0,0,0,1>(), in)
      - adjoint(DRILL<4>(fl)) * in.GSHIFT<-1,0,0,0>()
      - adjoint(DRILL<5>(fl)) * in.GSHIFT<0,-1,0,0>()
      - adjoint(DRILL<6>(fl)) * in.GSHIFT<0,0,-1,0>()
      - adjoint(DRILL<7>(fl)) * IF_EVEN_CHOOSE(in, in.GSHIFT<0,0,0,-1>())
      + DRILL<0>(ll)           * in.GSHIFT<3,0,0,0>()
      + DRILL<1>(ll)           * in.GSHIFT<0,3,0,0>()
      + DRILL<2>(ll)           * in.GSHIFT<0,0,3,0>()
      + DRILL<3>(ll)           * IF_EVEN_CHOOSE(in.GSHIFT<0,0,0,2>(),
                                                in.GSHIFT<0,0,0,1>())
      - adjoint(DRILL<4>(ll)) * in.GSHIFT<-3,0,0,0>()
      - adjoint(DRILL<5>(ll)) * in.GSHIFT<0,-3,0,0>()
      - adjoint(DRILL<6>(ll)) * in.GSHIFT<0,0,-3,0>()
      - adjoint(DRILL<7>(ll)) * IF_EVEN_CHOOSE(in.GSHIFT<0,0,0,-1>(),
                                                in.GSHIFT<0,0,0,-2>());
}
```

Listing 9.8: KS Dslash implemented in QUICQ

```
ksSpinorLattice X,Y,Z;
/// squared magnitude of ksSpinorLattice X into a single su3vector
/// (operator* squares the magnitude of an su3vector)
auto msq_v = REDUCE(X*X, su3add);
/// AXPY (overloaded operator* scales the su3vector by the scalar 'a')
auto a = 4.0;
auto Z = a*X + Y;
/// Dot product
auto pkp = REDUCE(dot(X,Y), scalar_add<float,float>);
/// L2 Norm
auto norm = std::sqrt(REDUCE(X*X, scalar_add<float,float>));
```

Listing 9.9: BLAS Level 1 routines in QUICQ

stencil kernel required another 100 lines of code, and a glue interface to integrate QUICQ's solver into MILC needed approximately 200 lines of code. The equivalent implementation of the KS Dslash kernel itself in the MILC application package is approximately 3500 lines of code long. Thus, programming in QUICQ offers a significant productivity boost in terms of the number of lines of code. The productivity gain is much higher when we consider that programming in QUICQ is free of MPI, OpenMP, or any other explicit parallelization constructs. Another point worth highlighting is that the mkernel functions are near equivalent to their MILC counterparts, and are reused directly in the EDSL.

The main motivation of QUARC is not just to provide a high-productivity programming interface, rather to combine high-productivity with high computational efficiency. To make that case, Chapter 10 presents the performance evaluation of the QUICQ kernels presented here, and compares the performance to the baseline application performance.

CHAPTER 10: EVALUATION AND PERFORMANCE ANALYSIS

This chapter provides an evaluation of the QUARC framework. It looks at the effectiveness of the data-layout transformation scheme introduced in Chapter 7. The performance of the stencil kernels in Chapter 7 is compared to a baseline performance obtained by the Intel compiler's auto-vectorizer. After that we present a detailed evaluation and performance analysis of QUICQ, the prototype lattice QCD DSL. QUICQ's performance and productivity gains are compared to two existing lattice QCD application frameworks: Chroma (Edwards and Joó, 2005), and MILC (MILC collaboration, 1992). Our evaluation compares the two Dslash kernels introduced in Chapter 9. In addition, for MILC we developed a complete conjugate gradient solver that was incorporated into an existing hybrid Monte Carlo molecular dynamics application. The results present both the solver-level performance improvement, and shows how that gain translates to whole application performance. We also provide a performance model to evaluate the Dslash performance against the machine capabilities. The chapter concludes by discussing two different code generation approaches that were tried when designing QUARC.

Test Platforms

All experiments for these set of evaluations were done on the following three Intel servers: A 32-core dual-socket Intel®Xeon®E5-2698 v3 (Haswell) at 2.3 GHz server with 32 KB L1d, 256 KB L2 caches and a shared 40 MB L3 cache per socket. A 68-core Intel®Xeon Phi ™7250 (KNL) at 1.4 GHz per core with 32 KB L1d, 1 MB L2 and a 16 GB high-bandwidth MCDRAM configured as a L3 cache. A 48-core dual-socket Intel®Xeon®Platinum 8160 (Skylake) server at 2.10 GHz with 32 KB L1d, 1 MB L2 and 33 MB of shared L3 per socket.

10.1 Stencil SIMD vectorization

This section compares QOPT's SIMD vectorization for the 2D and 3D Jacobi stencils in Chapter 7 against ICC 18.0.1 autovectorization at `-O3` optimization levels. Table 10.1 presents the execution

	2D Jacobi			3D Jacobi		
	4MB	2MB	1MB	4MB	2MB	1MB
QOPT best exec. time (μs)	317	159	79	342	171	86
QOPT median exec. time (μs)	327	159	79	360	174	88
QOPT worst exec. time (μs)	348	167	83	626	233	115
ICC exec. time (μs)	504	247	124	572	279	141
% Gain in best case	37%	36%	36%	39%	40%	39%

Table 10.1: ICC vectorization v/s QOPT vectorization on Intel Skylake (AVX512)

	2D Jacobi			3D Jacobi		
	4MB	2MB	1MB	4MB	2MB	1MB
QOPT best exec. time (μs)	1003	530	243	1331	690	325
QOPT median exec. time (μs)	1022	571	264	1548	762	361
QOPT worst exec. time (μs)	1103	590	272	2053	936	454
ICC exec. time (μs)	1127	723	338	1884	987	489
% Gain in best case	11%	27%	28%	29%	30%	34%

Table 10.2: ICC vectorization v/s QOPT vectorization on Intel KNL (AVX512)

times for the best, median, and worst performing QUARC data-layouts on the Intel Skylake server. It also shows the execution time for a reference C++ implementation. We used the `-qrestrict` flag along with the `restrict` keyword to aid ICC autovectorization. These options inform the compiler that the pointers used in our stencil loop do not alias. Additionally, we used the `xCORE-AVX512` and `qopt-zmm-usage=high` flags to ensure ICC used AVX512 vectorization on the Skylake server. Both 2D and 3D scalar Jacobi are relatively simple kernels, and we were only considering single threaded execution with periodic boundary conditions. Even then, on the Skylake server QOPT's SIMD vectorization in the best-case out performed ICC auto vectorization by up to 40% for the 3D example, and up to 37% for the 2D example.

Table 10.2 presents the evaluation of the stencil kernels on the KNL platform used for the performance study. On KNL the best performance was obtained for the smaller problem sizes. For the smallest 2D Jacobi example, QOPT's best data-layout outperformed ICC autovectorization by 28%. The gap closed for larger problem sizes, where the performance was limited by the memory bandwidth. Even for such cases, QOPT was able to achieve an 11% performance gain. The 3D Jacobi case has a comparatively higher arithmetic intensity, and thus gains more from SIMD vectorization. Here, QOPT out performed ICC by 34% for the smallest problem size. The larger problem size yielded a 29% performance gain.

10.2 Performance Evaluation of QUICQ

Figure of Merit (FOM) Both the Wilson Dslash and KS Dslash kernels from Chroma and MILC are evaluated based on the following figure of merit. The reported numbers calculate the number of floating point operations as follows:

$$FOM = \frac{iterations \times FlopCount \times lattice\ volume}{execution\ time(s)}(flops), \tag{10.1}$$

where *lattice volume* is total number of sites in the lattice. The *FlopCount* for both Wilson and KS Dslash was calculated in Section 9.1.

10.2.1 Chroma

The Chroma LQCD application is primarily developed using QDP++, with some of the performance critical sections re-implemented as C/C++ libraries. QDP++ is a C++ DSL based on expression templates. It uses the PETE expression template library (S. Haney J. Crotinger and Smith, 1999), and supports MPI, OpenMP and IA SIMD vectorization. QDP++ also provides an optional JIT compiler to generate code for Nvidia GPGPUs.

We evaluated the single node performance of Chroma's default Wilson Dslash kernel implemented in QDP++, and a hand-optimized version from the QPhiX library (Joó *et al.*, 2013). QPhiX provides tuned versions of the kernel for various generations of Intel architectures, and is currently the best performing implementation of the Wilson Dslash kernel on x86_64 architectures. Chroma by default uses the QPhiX implementation wherever available. We built both QDP++ and QPhiX using the Intel compiler 18.01, and use only OpenMP parallelization. For QDP++, the IA SIMD specializations were not used. The IA SIMD specialization only supports Intel SSE3 vector extensions, and was found to offer no performance gain on our test platforms. QUICQ used LLVM 6.0 and OpenMPI 4.0.0. The evaluation was done on two Intel servers, a 32-core Haswell with AVX2 vectorization, and a 68-core KNL supporting AVX512 vectorization. Experiments used all available cores and hyperthreads. The Haswell server provided two hyperthreads per core, and the KNL server offered four hyperthreads per core. In addition, the KNL server was configured to group the cores as four quadrants and the attached high-bandwidth memory

(HBM) was configured to run as a direct-mapped L3 cache. All results presented here use single-precision floating point numbers. The results do not include data-layout conversion timings.

Figure 10.1 shows the comparative performance of QDP++ and QUICQ on the Haswell server. The graph plots the lattice size against the calculated GFLOPs rate. We could not build the QPhiX implementation for AVX2 vectorization, and it was omitted from these results.

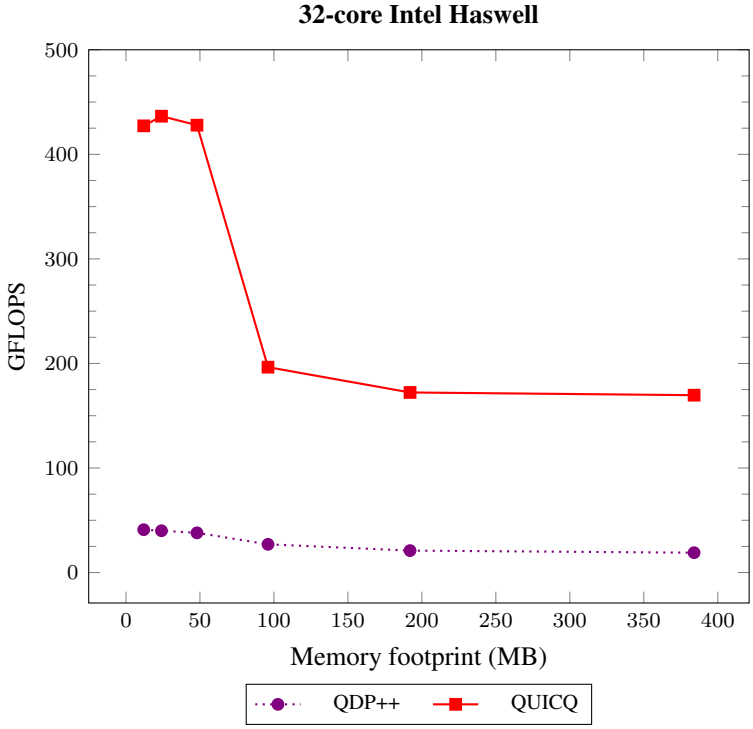


Figure 10.1: Single node Haswell comparison of QDP++ and QUICQ.

On the Haswell server, when the lattice size fits within L3 cache QUICQ outperformed QDP++ by 10×. There was a slight performance drop for problem sizes outside of L3 cache, but even for this case QUICQ consistently performed 8-9× better than QDP++. For the QUICQ evaluations we blocked the outermost lattice dimension, and allocated each block to an MPI rank. The data-layout for SIMD vectorization was constructed using the simd-rtf value of $\{1,2,4,1\}$. The innermost dimension was fully unrolled and the vector tile constructed out of the two inner dimensions.

The KNL evaluation included QPhiX as well. The problem sizes evaluated ranged from 51 MB to 1.6GB lattices. QPhiX and QUICQ blocked the lattice and bound each block to a hyperthread. When evaluating the smallest problem size of a 51MB lattice, the blocks were small enough to be fully L2 cache resident. QDP++ provides no provision for blocking the lattice, and the default OpenMP static

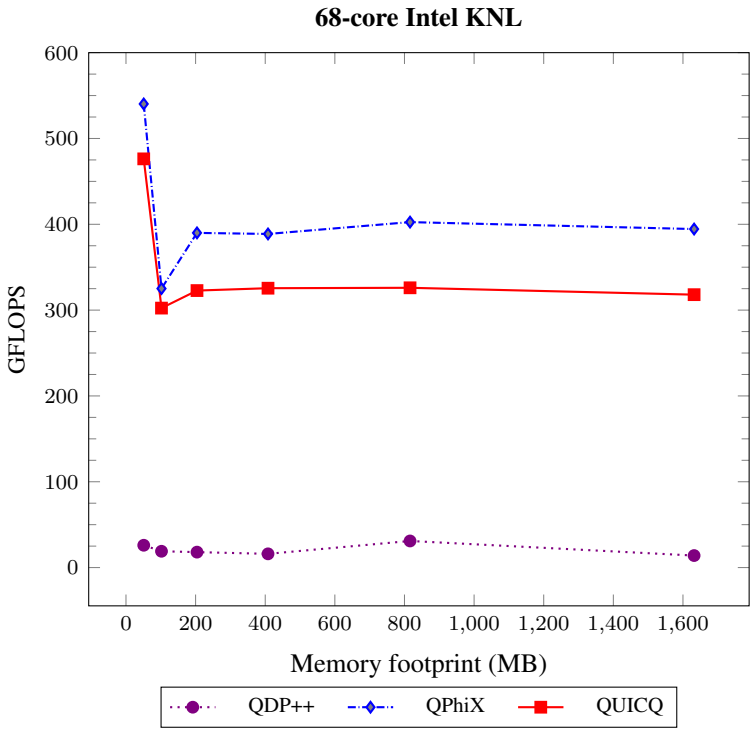


Figure 10.2: Single node KNL comparison of QDP++, QUICQ and QPhiX.

blocking and scheduling was applied. QUICQ's block distribution was similar to what was used on the Haswell test server, the lattice was block distributed along the outermost dimension. The data-layout strategy involved building a three-dimensional vector tile by transforming the three outer dimensions. As before the innermost dimension was fully unrolled. The data-layout was specified using a simd-rtf value of $\{2,2,4,1\}$. This allowed building a 16 wide SIMD dimension to accommodate AVX512 vectorization.

QUICQ's WD implementation outperformed QDP++ by up to $18\times$ for the smallest lattice, and was still $10\times$ faster for the larger lattices. QPHiX's KNL implementation performed $\sim 17\%$ faster than QUICQ. To understand the performance gap between QUICQ and QPhiX on KNLs, we measured and analyzed performance-counter data, and found that QPhiX had a L2 cache miss rate of $\sim 15\%$, whereas QUICQ had a miss rate of $\sim 25\%$. This difference only showed up when using hyperthreading. QPhiX uses OpenMP based fine-grained thread affinity settings to pin OpenMP threads to hardware threads. Due to this when QPhiX allocated two neighboring lattice blocks to different OpenMP threads, it could ensure that the threads are co-located on the same core. QUICQ uses OpenMPI as its MPI back end. OpenMPI only allows binding MPI ranks to individual cores. When using hyperthreads, OpenMPI does what is known as over-provisioning of MPI ranks per node. It does not offer any easy features to pin

ranks to hyperthreads. The only way to do so would be to write MPI *rankfiles*, something we have not explored at this point.

10.2.2 MILC

MILC is another production LQCD application mostly written in C, and that uses MPI for parallelism. Some kernels within MILC provide OpenMP support to hybrid MPI+OpenMP parallelism. There is limited support for explicit SIMD vectorization, but this application too uses the QPhiX library for optimized implementation of the Wilson and KS Dslash kernels and iterative solvers. We evaluated the performance of MILC's KS Dslash and an iterative conjugate gradient solver to corresponding implementations using QUICQ. Performance evaluation results for MILC use its default MPI+OpenMP execution mode. We used the MILC version distributed as part of the Trinity NERSC benchmark suite (National Energy Research Scientific Computing Center, 2013). The MILC application was compiled using ICC 18.0.1 and used the Intel MPI library.

We evaluate MILC's KS Dslash kernel against an equivalent version implemented in QUICQ, and also compare a hand-tuned version released as part of the QPhiX library. This is followed by evaluating a conjugate gradient solver that uses the KS Dslash kernel, and finally we analyze the whole program performance after integrating our solver.

KS Dslash Performance

The KS Dslash kernel was described in Section 9.1.2. This kernel is used in staggered fermion configuration of LQCD simulations. It is a 17-point stencil over a complex vector field.Table 10.3 shows the respective lines of code (*loc*) metric for the KS Dslash implementations in MILC, QPhiX and QUICQ. These counts were generated using Linux's *cloc* utility. The *loc* metric considered only functions that implement the Dslash operator, MPI communication, and other helper routines such as vector and prefetch intrinsics. The pseudo-code *loc* value represents the minimum number of lines needed to write the 17-point Dslash stencil and its elemental $SU(3)$ operators in C. The numbers show that going from 65 lines of pseudo-code to a real HPC application is a huge expansion in *loc*. More importantly, with this inflation of code size comes a large increase code complexity. MILC or QPHiX implementations include MPI, OpenMP and vector intrinsics interleaved with the application logic. This makes the code hard to read and maintain. It also makes it harder to improve and port them to newer architectures.

Table 10.3: Lines of code for KS Dslash MILC v/s QPhiX v/s QUICQ

Pseudo-code	MILC	QPhiX	QUICQ
65	3,486	2,358/1,726[a]	387

[a]Vector intrinsics headers implementing K-S Dslash for one of the several layouts that QPhiX can use. QPhiX uses a small code-generator to generate separate headers for each ISA and data-layout.

In this regard, QUICQ offers tremendous productivity gain. The implementation of the KS Dslash kernel and the solver routines in QUICQ takes almost one tenth the *loc* as MILC. The implementation was presented in Section 9.2. The DSL-level code does not require any explicit parallelization constructs. The code retains an expressiveness that comes closest to the pseudo code implementation. This gives application developers better readability, and allows them to focus on the algorithmic aspects of their applications. QUARC's speculative vectorizer and decoupled runtime data-layout specification means that porting the abstract application to a new IA involves recompilation with correct data-layout flags and architecture specifier. This offers a significant boost in productivity, reducing the time to develop or port algorithms to newer generations of architectures.

Figure 10.3 compares the performance of the single precision performance of the KS Dslash kernel. On all architectures QUICQ significantly outperforms MILC by over a factor of two. QUICQ's better cache blocking strategy and SIMD vectorization achieves a much greater performance, even when the problem falls out of last-level cache. QUICQ is also able to closely match QPhiX's performance on the 32-core Haswell server. The only variation is for the smallest lattice configuration on this server. This was a small $32\times8\times8\times16$ lattice for which QUICQ could not use a data-layout wide enough for AVX2 vectorization. The reported number are based on AVX vectorization and comparatively slower than QPhiX.

QUICQ was $1.8\times$ faster than MILC on KNLs as well. As with the Chroma experiments, there was a small performance gap between QUICQ and QPhiX on KNLs, still QUICQ's performance was within 10% of QPhiX performance. The KS Dslash implementation within the QPhiX library was not publicly available for the Skylake architecture at the time of this evaluation, so our evaluation on the Skylake server does not include QPhiX.

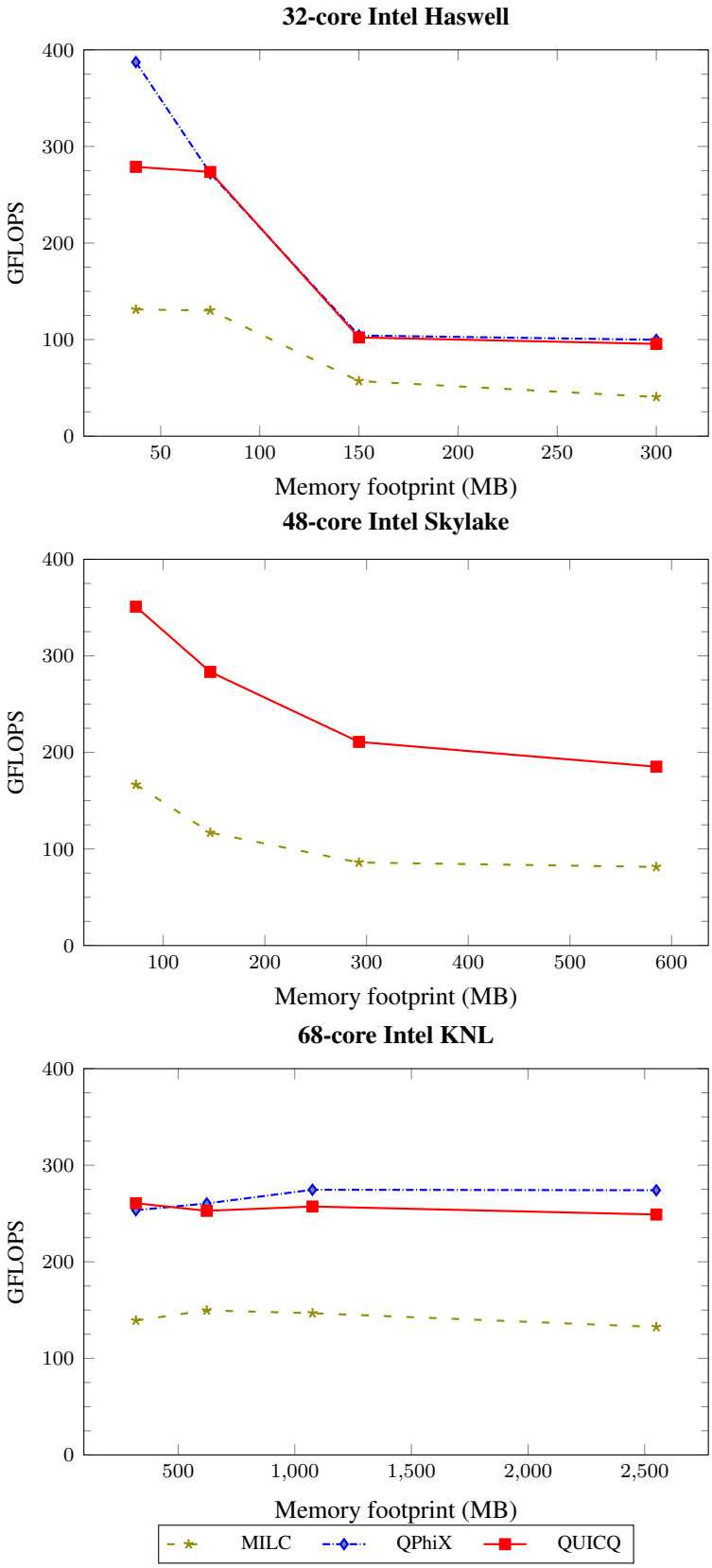


Figure 10.3: Single node comparison of MILC, QUICQ and QPhiX.

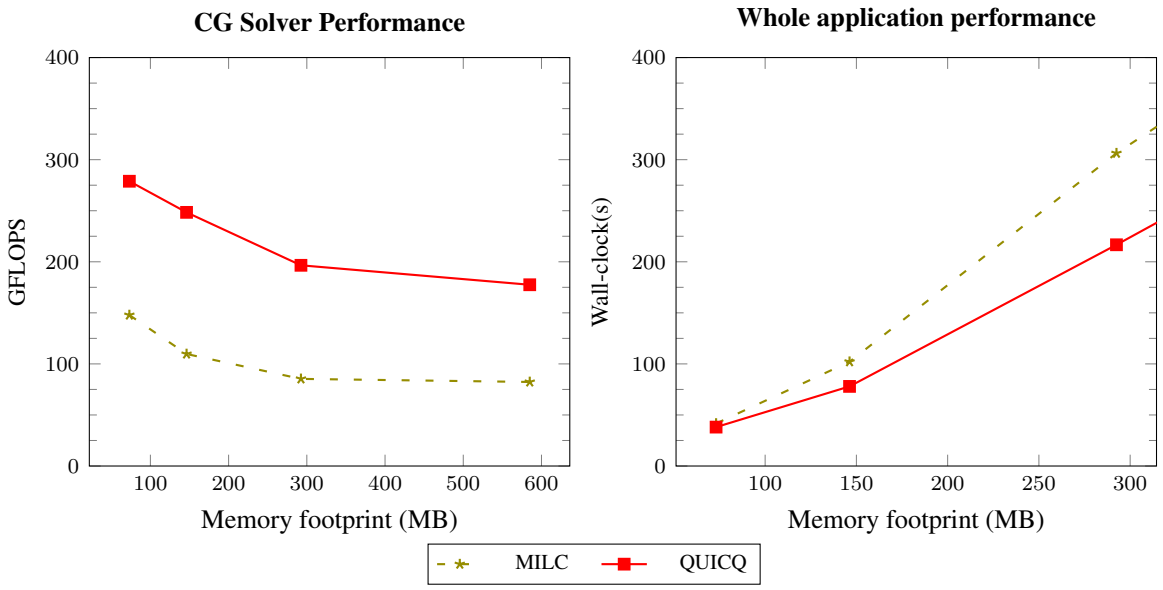


Figure 10.4: Evaluation of MILC and QUICQ's conjugate gradient solvers and the complete `su3_rmd` simulation on a single 48-core Skylake server.

Conjugate Gradient Solver and Whole Application Performance

The final step of our experimental evaluation of MILC was analyzing the full conjugate gradient solver built on top of the KS Dslash kernel, and to use that solver in the MILC benchmark (National Energy Research Scientific Computing Center, 2013). The benchmark is a simplified version of MILC's `su3_rmd` hybrid Monte Carlo molecular dynamics simulation. The solver constitutes around 70-80% of the total wall-clock time of each simulation. In addition to the Dslash kernel, the solver incorporates various level-one BLAS kernels. This causes the whole solver to have a slightly greater flop count than the Dslash. Using the methodology used in Chapter 9 for the KS Dslash kernel, the per site flop count for the solver is derived as 1187. The whole application and solver experiments were done only on the Skylake server. We updated the existing MILC `su3_rmd` benchmark application to use the solver written in QUICQ.

Figure 10.4 shows the performance of QUICQ for the conjugate gradient solver on the Skylake server. Here too QUICQ outperforms MILC by a factor of two. The whole application level performance reflects a much smaller gain than just the solver. The best performance improvement was 30% for a $16\times16\times16\times48$ size lattice.

This lower whole application performance gain is attributable to the cost of doing the data-layout transformations between MILC and QUICQ before each solver run. However, the benchmark application does significantly lower number of iterations per solve compared to a full-scale production simulation. A production simulation would do order of magnitude higher number of iterations. The net application-level gain should be higher for actual production loads. This proof-of-concept design shows the viability of the DSL approach to rewrite a portion of the existing application without wholesale re-implementation.

Performance Model to Evaluate Dslash Memory Performance

Section 9.1.2 provided the computational profile of the KS Dslash operator. This kernel has a large data footprint of 1560 bytes per lattice site when using single precision floating point numbers. The flop-to-byte ratio was calculated as 0.73 for every Dslash iteration. These figures are now used to develop a basic performance mode to evaluate the measured floating-point operation rate.

The measured flop rate can be written in terms of memory bandwidth as follows:

$$Flops = \frac{1147}{\frac{16\times G}{B_r} + \frac{16\times S}{B_r} + \frac{1\times S}{B_w}}, \tag{10.2}$$

where G is the size of a $SU(3)$ matrix representing a gauge link, S is the size of a $SU(3)$ vector representing a spinor, and B_r and B_w are the read and write memory bandwidths. The read spinors can be reused across iterations, but the gauge links do not have reuse and have to be reread each time. The result of each iteration needs to be written back to memory. The theoretical memory bandwidth requirement for the KS Dslash kernel can be computed based on these characteristics.

The $\frac{16\times S}{B_r}$ term can be disregarded if it is assumed that the read spinors are perfectly reused from last-level cache. Also, for simplification of the model B_r and B_w are considered the same. With these assumptions, the required memory bandwidth is computed as:

$$B = \frac{MeasuredFlops * (16 \times G + 1 \times S)}{1147}. \tag{10.3}$$

For the results obtained on the Skylake server for KS Dslash, the $32\times16\times16\times16$ lattice is used as a representative problem size that does not fully fit in the L3 cache on this server. The measured flop rate of the Dslash kernel for this lattice was 185 GFLOPS. Equation (10.3) provides the peak theoretical bandwidth requirement for this problem size as 189 GB/s. If this figure is compared to the published

memory bandwidth numbers (Gómez-Iglesias *et al.*, 2017) for this server, we notice that our measured results come within 97% of the peak Stream (McCalpin, 2007) bandwidth.

The performance model shows that the overall performance of this kernel is constrained by memory bandwidth, and that QUICQ obtains close to the maximum possible Stream memory bandwidth. Thus, based on the roofline model (Williams *et al.*, 2009) we argue that the performance attained by QUICQ is close to the peak possible flop-rate for this kernel on this particular server. Thus, lending credence to one of our core arguments that it is possible to achieve uncompromised performance using a DSL approach.

10.3 Comparing Two Approaches for Code Generation

The initial implementation of QUARC (quarc v0.4) required a static problem size for all `mddarrays`. The problem size and data-layout needed to be specified as a compiler flags to QOPT. The advantage of this approach was that it allowed us to do the various checks regarding applicability of data-layouts at compile time. Additionally, the polyhedral code generation process was much simplified, as all loop bounds were statically defined. The disadvantage to this approach was it made integrating QUARC-generated kernels with existing code cumbersome. A change of problem size or data-layout required recompilation of the whole program. Linking QUARC-generated binaries to existing applications also proved hard, since the glue interface needed to be specialized for the problem sizes and data-layouts that were previously generated using QUARC.

To get around these software engineering issues, we did a re-implementation of the code generator (quarc v0.5). The requirement for a compile time known problem size was removed. The speculative SIMD vectorizer was introduced to allow the code generation to be specialized for a subset of applicable data-layout choices. The layout selection and most of the checks to ensure legality were pushed into the QUARC-RT runtime library. These changes meant that the interoperability of QUARC with existing application became significantly easier. This was evident when we could use the solver routine implemented in QUICQ as a drop-in replacement for MILC's default solver.

A disadvantage of the second code generation method was it slightly increased compilation time, because multiple versions are generated for each kernel. Figure 10.5 presents the comparison of the KS Dslash kernel on the Skylake server for both approaches. These numbers are based on evaluating just the KS Dslash as a micro-kernel with synthetic data. As evidenced from these numbers, we noticed an

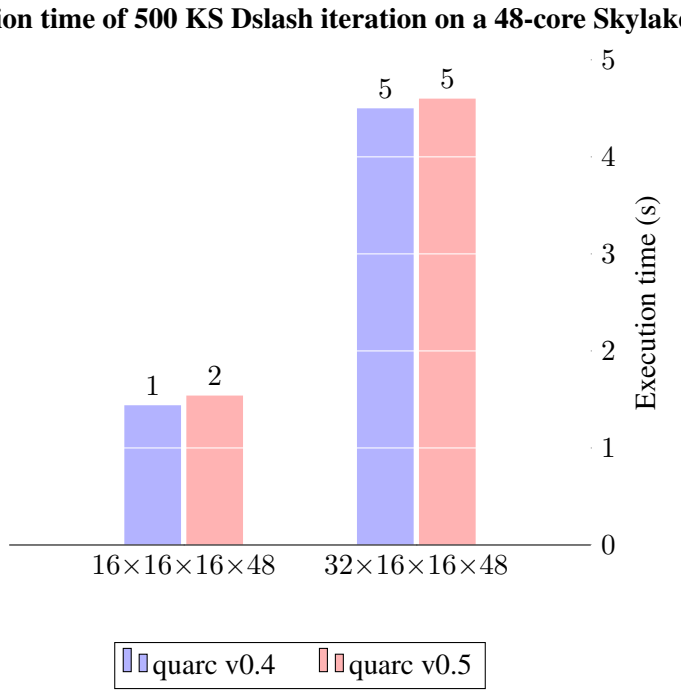


Figure 10.5: Comparing the two code generation approaches for QUARC for the KS Dslash lattice.

overhead of 8% for the best-case scenario when the problem fits in L3 cache. Once the problem becomes memory bound the overhead between the two methods is negligible.

Chapter Review

The results presented in this chapter underscore the effectiveness of data-layout transformations in improving SIMD vectorization for stencil kernels. The performance gain is especially pronounced for HPC applications such as lattice QCD that involve large number of iterative steps. The performance results of the QUICQ DSL show that our approach is well suited to offer large productivity gains, while being highly competitive with the best hand-tuned implementations for the type of kernels currently targeted by QUARC.

From a software architecture perspective, our design shows how a DSL approach is usable with existing production applications. Our framework allows existing hot spots to be rewritten in a high-productivity DSL, without needing a wholesale application rewrite. We feel this approach is better suited to wide adoption of DSLs.

CHAPTER 11: RELATED WORK

This chapter reviews related work and prior art that connect to aspects of QUARC's design and implementation. QUARC builds on ideas introduced in diverse domains like array-programming, metaprogramming, data parallelism, polyhedral compilation, and DSL design. We compare QUARC to other C++-based approaches and to previous DSLs in the domain of LQCD, as well as to recent HPC DSL designs and their implementation. The final section of the chapter evaluates data-layout transformation methods, contrasting them to QUARC's approach.

11.1 C++ Array-Programming Techniques

C++ does not directly support multi-dimensional dynamic arrays as first-class objects. Techniques such as C++ ETs (Veldhuizen, 1995; Vandevoorde and Josuttis, 2002) exist to get around this limitation. C++ ETs are based on C++ template recursion and lazy evaluation of C++ templates. The key concept is to evaluate a C++ template-based array expression only when it is needed, eliminating temporary arrays that otherwise would be created. Dedicated C++ ETs frameworks, such as Boost.Proto (Niebler, 2007), build on this basic design and include further optimizations using C++ template specialization. C++ ETs have been used widely to build BLAS libraries (Veldhuizen, 2006; Iglberger *et al.*, 2012; Walter and Koch, 2012), data-parallel EDSLs (Edwards and Joó, 2005; Parsons and Quinlan, 1994; Reynders and Cummings, 1998), and high-level abstractions to generate x86_64 SIMD code (Estérie *et al.*, 2012), and GPGPUs code (Wiemann *et al.*, 2011; Breglia *et al.*, 2013; Winter *et al.*, 2014).

The C++ ETs design generates loops and linearized array accesses in the C++ TMP-layer. Due to this design, scalar optimizations such as common sub-expression elimination and lazy code motion cannot directly be applied to array expressions written using C++ ETs. Such optimizations require additional loop analysis and optimization. Often, C++ template-generated obfuscation makes it nearly impossible for any compiler to detect any such loop-optimization opportunity. As an example, each C++ ET expands into a separate function call that has a scalarized loop nest. This implies that a prerequisite

to any loop optimization across the loop nests generated for multiple C++ ETs is inlining of each C++ ETs function call. The problem gets harder when C++ ETs generate parallel code. In such cases, the scalarized loop nests may include OpenMP or MPI library calls. Such library calls are opaque to most C++ compilers, and make it harder to apply function inlining and loop optimizations such as loop fusion in a deterministic fashion.

There have been attempts to mitigate some of the limitations of C++ ETs. The ROSE source-to-source compiler (Quinlan *et al.*, 2003) uses abstract syntax-tree rewrite rules to optimize code-generation out of C++ ETs. Winter *et al*. (Winter *et al.*, 2014) used JIT compilation of C++ ETs to compile for NVIDIA GPGPUs, and to generate SIMD vector code on x86_64. Despite much sterling effort, it has not been possible to overcome the design flaws associated with C++ ETs.

METAL's approach solves many of the issues of C++ ETs. Rather than generate low-level code out of array expressions in the C++ template-layer, METAL generates a domain-specific IR that encodes the array expressions directly into a compiler IR. QUARC's code-generation stages apply standard as well as domain-specific compiler passes to optimize and to auto-parallelize array expressions.

There have been other C++ array-programming techniques apart from C++ ETs. C++ Extension for Array Notation (CEAN) (Robison, 2013), Intel Array Building Blocks (ArBB) (Newburn *et al.*, 2011), and the Kokkos library (Edwards and Trott, 2013) incorporated array-based abstractions and data parallelism. CEAN expressions were inherently data-parallel and served as compiler hints to generate x86_64 SIMD vector code. ArBB used JIT compilation to scalarize loops for automatic thread parallelism and SIMD vectorization. Both CEAN and ArBB are now retired and are no longer extant. The Kokkos library supports array data-layouts. Kokkos does not support array expressions directly, but has *foreach* loop constructs. It is possible to specialize the *foreach* constructs both for GPGPUs and for x86_64 CPU parallelism from the same high-level interface.

11.2 C++ Parallel Skeleton Library

A parallel skeleton is an abstraction for a parallel computation pattern. Parallel skeletons, either as language extensions or as library functions, support programmers in writing parallel programs, thus improving programmer productivity. Parallel skeletons often are combined with parallel containers. Parallel containers abstract the partitioning and distribution of data across a compute environment.

Implementations of skeletons and parallel containers can abstract aspects of parallelism, data movement, and synchronization. In C++, the library function route has been an efficient way to implement parallel skeletons (Bischof *et al.*, 2004). Often, such implementations use generative programming using C++ templates. There are several production-quality C++ parallel skeleton libraries.

STAPL (Buss *et al.*, 2010) has parallel containers whose basic functionalities are equivalent to the sequential containers in the C++ Standard Template Library (STL). STAPL implements parallel versions of STL's algorithm functions, supporting both shared and distributed memory parallelism. STAPL's underlying programming model is task parallelism. It relies on a its own runtime system.

Thrust (NVIDIA Corporation, 2016) is another C++ header-only library with a C++ STL-like interface for parallel skeletons. The primary focus of Thrust is to accelerate code on NVIDIA GPGPUs. The library has memory-locality specifiers to control where data is stored. Thrust has optimized the NVIDIA CUDA (NVIDIA Corporation, 2010) implementation of standard C++ STL algorithms.

OpenCL skeletons are parallel programming skeletons for GPGPUs using OpenCL (The Khronos Group, 2015) directives. There are several libraries that abstract OpenCL code-generation. VexCL (Demidov *et al.*, 2013) is a C++ ETs libraries that generates OpenCL/CUDA code for several parallel patterns and BLAS kernels. Boost.Compute (Lutz, 2015) is a C++ template library for OpenCL programming. Boost.Compute parallel containers manage GPU-device memory allocation as well as automate data movement between the CPU and the GPU. SYCL (Group, 2018) is a cross-platform abstraction layer for OpenCL developed by the Khronos Group, an industry consortium which maintains the OpenCL standard. Sycl allows programs to be written in a "single-source" style, *i.e.*, as opposed to separate sources for host and device code, a single source file can include both. SYCL's interface is like C++ STL, and includes built-in parallel patterns.

C++17 Parallel STL The library-based parallel skeleton implementations stem from the C++ standard's lack of support for parallel skeletons and algorithms. The revamped C++17 standard aims to mitigate this discrepancy. The new C++17 standard has included a set of parallel algorithms in the newly added C++17 Parallel STL standard library. The scope for C++17 Parallel STL is restricted to multi-core and SIMD parallelization. The initial set of algorithms include only three parallel patterns: *foreach*, *reduce* and *scan*. Present discussions in the C++ community are exploring ways to expand on this initial set of patterns. There are also discussions around supporting GPGPUs and accelerator programming in an updated C++ standard.

QUARC's current design incorporates the *foreach*, *shift*, and *reduce* parallel skeletons. All three skeletons are implemented inside the METAL frontend. QUARC's parallel skeletons act on the `mddarray` global view array container. The uniqueness in QUARC's design is its support for shared-memory parallelism, distributed-memory parallelism, and SIMD vectorization using the same set of abstractions. This makes QUARC's parallel skeletons powerful and wellsuited for large-scale HPC application domains.

11.3 LQCD DSLs

LQCD has existing production DSLs based on C++ ETs. QDP++ (Edwards and Joó, 2005) is a legacy DSL that uses the PETE (S. Haney J. Crotinger and Smith, 1999) C++ ET library. Although QDP++ is a highly expressive programming language for LQCD applications, its performance does not match those of hand-tuned libraries. Performance-critical portions of LQCD applications usually are coded outside of QDP++. The goal of improving QDP++ performance automatically was the main motivation for QUARC.

QDP-JIT (Winter *et al.*, 2014) is a JIT compiler framework to optimizes QDP++. It embeds a JIT compiler in the C++ ET template interface. Runtime evaluation of the ETs generates native code *via* the JIT compiler. The initial goal of QDP-JIT was to generate code for NVIDIA GPGPUs. Further extensions to QDP-JIT have added code-generation capabilities for x86 and MIC architectures. QDP-JIT offers a performance advantage over QDP++, but its performance does not compare to hand-tuned libraries such as QPhiX (Joó *et al.*, 2013) or QUDA (Babich *et al.*, 2011).

Grid (Boyle *et al.*, 2015) is a more modern C++ template-based LQCD DSL. Grid uses abstractions for architecture-specific SIMD data types and supports a set of fixed data-layout choices based on architecture-specific SIMD register width. Despite having a modern C++-based design, Grid too suffers from limitations that impact a library-only EDSL design. Grid's interface has to be updated for every SIMD ISA generation, and optimizations such as cache blocking require separate template specialization.

11.4 DSLs and DSL Frameworks

The need for DSLs and DSL frameworks in HPC has been apparent for a long time. The early 2000's saw several projects aimed at DSLs and at domain-specific compiler implementations. The telescoping languages (Kennedy *et al.*, 2005) design was an influential early proposal seeking to address

property discovery and transformation of high-level code. The telescoping languages aimed at reducing the performance gap between scripting languages and code written in C or in FORTRAN. The design presented, among other things, a concept called a "library-aware" optimizer. Such an optimizer could be capable of substituting DSL function calls with calls to specialized library function implementations. It was also proposed that the creation of a library-aware optimizer could be automated. The Tensor Contraction Engine (TCE) (Baumgartner *et al.*, 2005) was a domain-specific compiler developed to compile a Mathematica-style DSL into FORTRAN code. TCE targeted two levels of optimizations. At the highest level, domain-specific algebraic substitutions were done to reduce the computational complexity. Then, the DSL code was source-to-source translated to FORTRAN, and loop fusion transformations were explored. TCE primarily was used for quantum chemistry code. Finally, POOMA (Reynders and Cummings, 1998) was a C++ ET based data parallel framework. POOMA was a precursor to LQCD's QDP++ DSL. POOMA suffered from the same issues afflicting C++ ETs.

There has been a lot of advancement in DSL technologies since these early frameworks. We look next at some of the most successful DSL frameworks that have prevailed in the past decade.

Delite (Sujeeth *et al.*, 2014) is a Scala-based DSL framework. It provides several data-parallel patterns and DSL data types that form the basic blocks for HPC DSLs. The framework uses a modified version of the standard Scala compiler called *Scala-virtualized* (Moors *et al.*, 2012) to construct a domain-specific IR from Scala's JVM byte code. To identify the DSL constructs embedded inside a standard Scala program, Delite uses a version of MSP (Chapter 2) called Lightweight Modular Staging (LMS). The domain-specific IR is optimized using a standalone Delite compiler. The Delite compiler provides several optimizations, such as common sub-expression elimination, dead code elimination, and code motion, that are applied directly to high-level DSL constructs and operators. In addition, Delite supports AOS-to-SOA data-layout transformations and has some support for fusion of data-parallel operators based on producer-consumer dependence.

The Delite approach bears similarities to that of QUARC, but the overall engineering and design of the frameworks are very different. Unlike Delite, QUARC stages its domain-specific IR inside LLVM's SSA CFG IR using only a small set of DSL intrinsic function calls. QUARC's compiler is fully integrated with the standard LLVM compiler and is implemented as a set of custom LLVM compiler passes. This allows us to use existing data-flow optimizations, polyhedral code-generation, and loop optimizations directly by utilizing the LLVM infrastructure. Although Delite supports AOS-to-SOA data-layout

transformations, QUARC's data-placement optimizations for numeric nested array types are much more general and support a larger space of data-layouts.

SPIRAL (Püschel *et al.*, 2005) is a digital signal-processing (DSP) DSL with a split-programming language interface. Algorithms for a DSP transform are written as formula in a language called SPL. SPL uses domain-specific algebraic simplification rules, or rewrite rules, to optimize the high-level formulae. The process incorporates feedback-driven autotuning. Low-level code-generation is driven by a separate specification layer. SPIRAL uses two types of code-generation specifications that it calls *tags* and *templates*. Tags are directly attached to SPL rules and define options such as loop unroll factors. Templates define a rich loop-generation algebra and control most aspects of low-level loop generation. Templates allow decoupling of the high-level algorithms from platform-specific code-generation and ease evaluation of different code-generation strategies for the same high-level algorithm.

Halide (Ragan-Kelley *et al.*, 2013) is an image-processing DSL with a two-level split-programming interface. Halide programs have an algorithm section and a separate code-generation specification that is called a *schedule*. The schedule guides loop optimizations and parallel code-generation. Halide uses an autotuning approach to optimize schedules for a given algorithm. Although Halide was primarily designed for image processing, the DSL has other potential application. There is ongoing effort to use Halide in designing DSLs for LQCD.

QUARC's METAL and ATL approach drew on SPIRAL and on Halide's split-language design.

Terra (DeVito *et al.*, 2013) is a statically typed language for building EDSLs in the Lua programming language (LabLua, 2015). Terra and Lua share the same lexical scope. The design allows applications to be prototyped rapidly in Lua before portions of the code are staged and re-implemented in Terra. Terra uses dynamic staging and allows for runtime feedback-based autotuning.

11.5 Data-Layout Transformations

Data-layout transformations, and specifically, data-placement abstractions, have been discussed elsewhere in this dissertation. This section summarizes and highlights some of the key prior work. Anderson *et al*. (Anderson *et al.*, 1995) used a data-layout transformation based on the strip-mine and interchange loop transformation to address false sharing on cache-coherent architectures. Their technique is expressed easily in our $\rho\phi$ algebra as a one-dimensional array reshape-and-transpose operation. A

very similar technique was introduced by Lu *et al*. (Lu *et al.*, 2009) to improve data locality on chip-multiprocessors with non-uniform caches. So *et al*. (So *et al.*, 2004) used a data-layout transformation to improve memory access on FPGA-based systems. Their method transformed arrays for placement on multiple memory banks to enhance memory bandwidth by paralleling memory accesses. Sung *et al*. used a matrix transpose-based technique (Sung *et al.*, 2010) to change array data-layouts on GPGPUs, thus improving memory-level parallelism for structured grid applications.

FORTRAN variants like FORTRAN D (Hiranandani *et al.*, 1992) and High Performance FORTRAN (HPF) (Loveman, 1993) offered directive-based data-placement abstractions for multi-dimensional arrays. The directives were used primarily to define hypercubic blocking for arrays over distributed memory systems. Similar block directives were used in ZPL (Chamberlain *et al.*, 1998), and in the HPCS programming language family (Chamberlain *et al.*, 2007; Charles *et al.*, 2005; Allen *et al.*, 2008).

Recent compiler frameworks like Intel's ispc (Pharr and Mark, 2012) and a Habenero-C based compiler framework from Majeti *et al*. (Majeti *et al.*, 2016) have AOS and SOA-type data-layout in the code frontend. A stencil compiler prototype developed by Henretty *et al*. (Henretty *et al.*, 2013) used the previously discussed dimension-lift and transpose data-layout transformations to optimize stencil kernels.

CHAPTER 12: CONCLUSION AND FUTURE WORK

The issue of combining high programmer productivity with commensurate computational efficiency is a long-standing challenge in computer science. The challenge is greater within the space of HPC where there is the requirement of cluster-level parallelism. When compared to serial programming, writing parallel programs for a cluster is much harder as cluster-level parallel programming has limited compiler and programming language support. Such programs are typically written using external libraries and APIs such as MPI. The already hard problem of cluster-level parallel programming has gained a new dimension due to the onset of chip multi-processing and large-sale parallelism with a single compute node of a cluster computer. Getting a large fraction of peak single-node performance of an application by compiling it for a newer, faster architecture generation is no longer a viable option.

To mitigate aspects of the challenge, our thesis argued for a data-placement-based approach where code-generation and optimization is driven by data-placement. We argued that an EDSL-based design is well-suited for such a data-placement-based code-generation approach. The QUARC framework is a proof-by-example in support of this thesis. The data-parallel abstractions in QUARC was the basis of the prototyped LQCD DSL, QUICQ. LQCD kernel implementations in QUICQ (Section 9.2) were simpler than comparable existing production code. QUICQ not only took less than one-tenth the lines of code, but required no explicit parallel programming. The performance results in Chapter 10 show that the kernels implemented in QUICQ are up to twice as fast in comparison to existing production code written in C. The performance of QUICQ was also competitive with those of the very best hand-optimized library implementation. In addition, QUICQ outperformed an existing LQCD EDSL by a factor of ten. The evaluation results, both in terms of the productivity gain and in terms of performance gain, make a strong case in support of our thesis.

The design and implementation of QUARC addressed two important issues regarding EDSL development for HPC domains.

The first issue relates to data placement abstractions and the need for multiple layers of such abstractions. Our $\rho\phi$ index-space transformation algebra (Chapter 4) addressed this issue. Using this algebra, QUARC's ATL specifications can define data-placement at multiple levels. Our work demonstrates the use of the same set of operators to define both on node data-layouts and global data-distributions. The current implementation is limited to these two types of data-placement abstractions, but the algebra is extensible to other architectures. Specific to shared-memory systems, we implemented a new SIMD auto-vectorization method that was solely based on data-layouts defined using the $\rho\phi$ algebra. The new auto-vectorization method performs up to 40% better than default auto-vectorization for scalar stencil kernels. The performance gain is much higher for the complex stencil kernels in LQCD. There exists prior art that used hand-vectorization after similar data-layout transformations, but the automated technique implemented in QUARC makes this kind of vectorization robust and portable. Another key feature of QUARC's data-placement abstractions is they work with polyhedral code-generation. Chapter 6 explained in detail the use of polyhedral code-generation methods in combination with custom data-placement abstractions. To the best of our knowledge, QUARC is the first system to demonstrate this type of code-generation approach.

The second issue addressed by QUARC's design is the issue of making a compiler aware of the programmer's intent. Traditionally, low-level languages such as C and C++ make a number of code-generation decisions in the frontend. Loop and access linearization are examples of such decisions. Losing these information makes it hard for any compiler to recover the programmer's intent, and thwarts subsequent compiler analyses and optimizations. Existing EDSLs address some of these issues by constructing domain-specific IRs and using domain-specific AST-level rewrites before passing the code to a lower-level compiler. Our approach expanded on these ideas. Using a new metaprogramming technique called ACTs (Section 5.2), we presented a new design for domain-specific IRs. ACTs are a way to encode a domain-specific IR into a general-purpose compiler's IR. By encoding a domain-specific IR into a general-purpose compiler's IR, QUARC lowers the engineering effort of domain-specific code-generation and optimizations. The viability of this design was demonstrated by QOPT's speculative SIMD vectorization optimization. In Section 12.1, we discussed some other potential benefits of QUARC's ACTs-based IR design.

In conclusion, the design and implementation of QUARC introduced new ways of constructing EDSLs using C++14 and LLVM. Our work showed the importance of data-layouts on modern x86_64-

based architectures, and offered a way of automating several of the code-generation decisions that are dependent on data-layouts. The value of these innovations is demonstrated by the implementation of QUICQ, an EDSL for a real-world HPC application domain. There is significant room for further expasion of the ideas introduced in our work, and targeting the development for other HPC EDSLs using QUARC. The next section concludes this dissertation by presenting a list of such potential future work.

12.1 Future Work

The current design of QUARC is general, and it offers the potential of extensions to support large-scale, real-world application domains. The implementation for the purpose of this dissertation only serves as a technology demonstrator in support of our main thesis around data-placement abstractions. There are several other areas that present opportunities for further development of QUARC. Following are some of the areas that need to be incorporated into QUARC's implementation to quantify its applicability in a real-world environment.

METAL's design, using ACTs and DSL intrinsics, also makes it possible to develop other patterns such as operators required for algebraic multigrid methods. Such techniques are becoming increasingly useful, but programming them by hand is still difficult. An EDSL-based approach could help simplify the conceptual and programming challenges.

The design of late scalarization of array expressions preserves high-level constructs late into the code-generation process. This opens up the possibility of applying high-level optimizations directly to array expressions prior to lowering them into low-level loops and elemental accesses. QOPT presented a design that explored the viability of this method. To do this well for general cases will require data-dependence analysis. Chapter 6 discussed how data-dependence analysis can be incorporated into QUARC. Incorporating data-dependence-based analyses would enable targeting much wider range of kernels.

The current focus of QUARC was restricted to Intel architectures, but both QUARC's EDSL design methodology and data-layout transformation techniques should map well to other architectures such as GPGPUs. The emergence of very large accelerator-based systems makes this a priority.

Finally, techniques for profiling and debugging EDSL generated code are a necessity. This area presents both practical design and implementation challenges and open research questions. Such challenges need to be solved prior to wide adoption of EDSL frameworks like QUARC.

BIBLIOGRAPHY

Acharya, A. and Bondhugula, U. (2015). PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–64.

Adve, V. and Mellor-Crummey, J. (1998). Using Integer Sets for Data-parallel Program Analysis and Optimization. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 186–198, New York, NY, USA. ACM.

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G. L., and Tobin-Hochstadt, S. (2008). The Fortress Language Specification Version 1.0. `http://homes.soic.indiana.edu/samth/fortress-spec.pdf`.

Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J. (1983). Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 177–189, New York, NY, USA. ACM.

Ancourt, C. and Irigoin, F. (1991). Scanning Polyhedra with DO Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 39–50, New York, NY, USA. ACM.

Anderson, J. M., Amarasinghe, S. P., and Lam, M. S. (1995). Data and Computation Transformations for Multiprocessors. *ACM SIGPLAN Notices*, 30(8):166–178.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67.

Babich, R., Clark, M. A., Joó, B., Shi, G., Brower, R. C., and Gottlieb, S. (2011). Scaling Lattice QCD Beyond 100 GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 70:1–70:11, New York, NY, USA. ACM.

Backus, J. (1978). The History of FORTRAN I, II, and III. *SIGPLAN Not.*, 13(8):165–180.

Barua, R., Lee, W., Amarasinghe, S., and Agarwal, A. (1999). Maps: a compiler-managed memory system for Raw machines. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pages 4–15.

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA. IEEE Computer Society Press.

Baumgartner, G., Auer, A., Bernholdt, D. E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R. J., Hirata, S., Krishnamoorthy, S., Krishnan, S., chung Lam, C., Lu, Q., Nooijen, M., Pitzer, R. M., Ramanujam, J., Sadayappan, P., and Sibiryakov, A. (2005). Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models. *Proceedings of the IEEE*, 93(2):276–292.

Benabderrahmane, M. W., Pouchet, L. N., Cohen, A., and Bastoul, C. (2010). The Polyhedral Model Is More Widely Applicable Than You Think. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6011 LNCS:283–303.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98.

Bhaskaracharya, S. G., Bondhugula, U., and Cohen, A. (2016). SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 526–538, New York, NY, USA. ACM.

Bischof, H., Gorlatch, S., and Leshchinskiy, R. (2004). Generic Parallel Programming Using C++ Templates and Skeletons. In Lengauer, C., Batory, D., Consel, C., and Odersky, M., editors, *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, pages 107–126, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bondhugula, U. (2013). Compiling Affine Loop Nests for Distributed-memory Parallel Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 33:1–33:12, New York, NY, USA. ACM.

Boyle, P., Yamaguchi, A., Cossu, G., and Portelli, A. (2015). Grid: A Next Generation Data-Parallel C++ QCD Library. `https://github.com/paboyle`.

Breglia, A., Capozzoli, A., Curcio, C., and Liseno, A. (2013). CUDA Expression Templates for Electromagnetic Applications on GPUs [EM Programmer's Notebook]. *IEEE Antennas and Propagation Magazine*, 55(5):156–166.

Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N. M., and Rauchwerger, L. (2010). STAPL: Standard Template Adaptive Parallel Library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA. ACM.

Carter Edwards, H., Trott, C. R., and Sunderland, D. (2014). Kokkos. *J. Parallel Distrib. Comput.*, 74(12):3202–3216.

Chamberlain, B. L., Callahan, D., and Zima, H. P. (2007). Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312.

Chamberlain, B. L., Choi, S.-E., Lewis, E. C., Lin, C., Snyder, L., and Weathersby, W. D. (1998). The Case for High Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40(10):519–538.

Clark, M., Babich, R., Barros, K., Brower, R., and Rebbi, C. (2010). Solving Lattice QCD Systems of Equations Using Mixed Precision Solvers on GPUs. *Computer Physics Communications*, 181(9):1517 – 1528.

Clark, M. A., Joó, B., Strelchenko, A., Cheng, M., Gambhir, A. S., and Brower, R. C. (2016). Accelerating Lattice QCD Multigrid on GPUs Using Fine-Grained Parallelization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 795–806.

Creutz, M. (1987). Quarks, Gluons and Lattices. Cambridge University Press 1983. ISBN 0-521-31535-2. (Cambridge Monogr. on Mathematical Physics) . *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 67(1):16.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. (2008). Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA. IEEE Press.

Deb, D., Fowler, R. J., and Porterfield, A. (2016). QUARC: An Array Programming Approach to High Performance Computing. *Proceedings of the 29th International Workshop on Languages and Compilers for Parallel Computing, LCPC'16*.

Deb, D., Fowler, R. J., and Porterfield, A. (2017). QUARC: An Optimized DSL Framework Using LLVM. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC'17, pages 9:1–9:11, New York, NY, USA. ACM.

Demidov, D., Ahnert, K., Rupp, K., and Gottschling, P. (2013). Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries. *SIAM Journal on Scientific Computing*, 35(5):C453–C472.

Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. *IEEE J. Solid-State Circuits*, 9(5):256–268.

Denning, P. J. and Schwartz, S. C. (1972). Properties of the Working-set Model. *Commun. ACM*, 15(3):191–198.

Department of Defense (2001). DoD Research and Development Agenda For High Productivity Computing Systems. `https://www.nitrd.gov/nitrdgroups/images/6/64/DoD_Research_and_Development_Agenda_For_HPCS.pdf`.

DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., and Vitek, J. (2013). Terra: A Multi-stage Language for High-performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA. ACM.

Dirac, P. A. M. (1928). The Quantum Theory of the Electron. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 117(778):610–624.

Dongarra, J. J., Graybill, R., Harrod, W., Lucas, R. F., Lusk, E. L., Luszczek, P., McMahon, J., Snavely, A., Vetter, J. S., Yelick, K. A., Alam, S. R., Campbell, R. L., Carrington, L., Chen, T.-Y., Khalili, O., Meredith, J. S., and Tikir, M. M. (2008). DARPA's HPCS Program- History, Models, Tools, Languages. *Advances in Computers*, 72:1–100.

Edwards, H. C. and Trott, C. R. (2013). Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Proceedings of the 2013 Extreme Scaling Workshop (Xsw 2013)*, XSW '13, pages 18–24, Washington, DC, USA. IEEE Computer Society.

Edwards, R. G. and Joó, B. (2005). The Chroma Software System for Lattice QCD. *Nucl. Phys. Proc. Suppl.*, 140:832. [,832(2004)].

Eisenecker, U. W. (1997). Generative Programming (GP) With C++. In Mössenböck, H., editor, *Modular Programming Languages: Joint Modular Languages Conference, JMLC'97 Linz, Austria, March 19–21, 1997 Proceedings*, pages 351–365, Berlin, Heidelberg. Springer Berlin Heidelberg.

El-Ghazawi, T. and Smith, L. (2006). UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA. ACM.

Estérie, P., Gaunard, M., Falcou, J., Lapresté, J.-T., and Rozoy, B. (2012). Boost.SIMD: Generic Programming for Portable SIMDization. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 431–432, New York, NY, USA. ACM.

Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., Park, J. Y., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. (2006). Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*.

Free Software Foundation (2018). GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`.

Gannon, D., Jalby, W., and Gallivan, K. (1988). Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5(5):587 – 616.

Garcia, R., Siek, J., and Lumsdaine, A. (2001). Boost.MultiArray Library. `https://www.boost.org/doc/libs/1_66_0/libs/multi_array/doc/index.html`.

Giles, M. B., Mudalige, G. R., Spencer, B., Bertolli, C., and Reguly, I. Z. (2013). Designing OP2 for GPU architectures. *Journal of Parallel and Distributed Computing*, 73(11):1451 – 1460. Novel architectures for high-performance computing.

Gómez-Iglesias, A., Chen, F., Huang, L., Liu, H., Liu, S., and Rosales, C. (2017). Benchmarking the Intel®Xeon®Platinum 8160 Processor. In *TACC Technical Report TR-17-01*.

Gropp, W., Hoefler, T., Thakur, R., and Lusk, E. (2014). *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press.

Grosser, T. (2011). Enabling Polyhedral Optimizations in LLVM. *Diploma Thesis*.

Group, K. (2018). Sycl: C++ Single-source Heterogeneous Programming for OpenCL. `https://www.khronos.org/sycl`.

Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., and Sadayappan, P. (2011). Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *Compiler Construction: 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*, pages 225–245, Berlin, Heidelberg. Springer Berlin Heidelberg.

Henretty, T., Veras, R., Franchetti, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. (2013). A Stencil Compiler for Short-vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA. ACM.

Hestenes, M. R. and Stiefel, E. (1952). Methods of Conjugate Gradients for Solving Linear Systems. *Journal of research of the National Bureau of Standards*, 49:409–436.

Hiranandani, S., Kennedy, K., and Tseng, C.-W. (1992). Compiling Fortran D for MIMD Distributed-memory Machines. *Commun. ACM*, 35(8):66–80.

Hoshino, T., Maruyama, N., and Matsuoka, S. (2014). An OpenACC Extension for Data Layout Transformation. In *2014 First Workshop on Accelerator Programming using Directives*, pages 12–18.

IBM Corporation (2015). IBM XL C/C++ for Linux, V13.1.2. `https://ibm.co/2ruJgoE`.

Iglberger, K., Hager, G., Treibig, J., and Rude, U. (2012). High Performance Smart Expression Template Math Libraries. In *2012 International Conference on High Performance Computing and Simulation (HPCS)*. IEEE.

Interagency Working Group on Information Technology Research and Development (2006). GRAND CHALLENGES: Science, Engineering, and Societal Advances Requiring Networking and Informational Technology Research and Development . `https://www.nitrd.gov/pubs/200311_grand_challenges.pdf`.

Iverson, K. E. (1962). *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA.

Joó, B., Kalamkar, D. D., Vaidyanathan, K., Smelyanskiy, M., Pamnany, K., Lee, V. W., Dubey, P., and Watson, W. (2013). Lattice QCD on Intel®Xeon Phi™ Coprocessors. In *Supercomputing*, pages 40–54. Springer Science + Business Media.

Kamil, S., Husbands, P., Oliker, L., Shalf, J., and Yelick, K. (2005). Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 36–43, New York, NY, USA. ACM.

Kennedy, K. and Allen, J. R. (2002). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Kennedy, K., Broom, B., Chauhan, A., Fowler, R. J., Garvin, J., Koelbel, C., Mccosh, C., and Mellor-Crummey, J. (2005). Telescoping Languages: A System for Automatic Generation of Domain Languages. *Proceedings of the IEEE*, 93(2):387–408.

Kennedy, K., Koelbel, C., and Schreiber, R. (2004). Defining and Measuring the Productivity of Programming Languages. *The International Journal of High Performance Computing Applications, (18)4, Winter*, 2004:441–448.

LabLua (2015). The Programming Language Lua. `https://www.lua.org/`.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA. ACM.

Loveman, D. B. (1993). High Performance Fortran. *IEEE Parallel Distrib. Technol.*, 1(1):25–42.

Lu, Q., Alias, C., Bondhugula, U., Henretty, T., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P., Chen, Y., Lin, H., and f. Ngai, T. (2009). Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 348–357.

Lutz, K. (2015). Boost.Compute. `https://bit.ly/2UTshdk`.

Majeti, D., Barik, R., Zhao, J., Grossman, M., and Sarkar, V. (2014). Compiler-Driven Data Layout Transformation for Heterogeneous Platforms. In *Euro-Par 2013: Parallel Processing Workshops*, pages 188–197. Springer Science + Business Media.

Majeti, D., Meel, K. S., Barik, R., and Sarkar, V. (2016). Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 240–250, New York, NY, USA. ACM.

MATLAB (2010). *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts.

McCalpin, J. D. (2007). STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia. A continually updated technical report. http://www.cs.virginia.edu/stream/.

Mellor-Crummey, J., Adve, V., Broom, B., Chavarría-Miranda, D., Fowler, R., Jin, G., Kennedy, K., and Yi, Q. (2002). Advanced Optimization Strategies in the Rice dHPF Compiler. *Concurrency Computation Practice and Experience*, 14(8-9):741–767.

MILC collaboration (1992). MILC Collaboration Code for Lattice QCD Calculations.

Moors, A., Rompf, T., Haller, P., and Odersky, M. (2012). Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 117–120, New York, NY, USA. ACM.

More, T. (1973). Axioms and Theorems for a Theory of Arrays. *IBM J. Res. Dev.*, 17(2):135–175.

Mullin, L. (1988). A Mathematics of Arrays. *PhD thesis, Syracuse University, December 1988*.

Myers, G. J. (1979). Review of Advances in Computer Architecture by Glenford J. Myers. Wiley-Interscience Division of John Wiley and Sons 1978. *SIGARCH Comput. Archit. News*, 7(7):25–26.

National Energy Research Scientific Computing Center (2013). NERSC-8/Trinity Benchmarks. `https://bit.ly/2R8i0Y8`.

Newburn, C. J., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S. D., Wang, Z. G., Du, Z. H., Chen, Y., Wu, G., Guo, P., Liu, Z., and Zhang, D. (2011). Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA. IEEE Computer Society.

Niebler, E. (2007). Proto: A Compiler Construction Toolkit for DSELs. *ACM SIGPLAN Symposium on Library-Centric Software Design*.

Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Aprà, E. (2006). Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231.

Numrich, R. W. and Reid, J. (1998). Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31.

NVIDIA Corporation (2010). NVIDIA CUDA C Programming Guide. Version 3.2.

NVIDIA Corporation (2016). NVIDIA Thrust. `https://developer.nvidia.com/thrust/`.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition.

Odlyzko, A. M. (1995). Handbook of Combinatorics (Vol. 2). pages 1063–1229. MIT Press, Cambridge, MA, USA.

OpenACC.org (2013). OpenACC 2.0 Specifications. `https://www.openacc.org/specification`.

OpenMP Architecture Review Board (2015). OpenMP 4.5 Specifications. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`.

Oren Ben-Kiki, Clark Evans, Brian Ingerson (2009). YAML: YAML Ain't Markup Language. `http://yaml.org/`.

Parsons, R. and Quinlan, D. (1994). A++/P++ Array Classes for Architecture Independent Finite Difference Computations. *In Proc. 2nd Annual Object-Oriented Numerics Conf. (OON-SKI'94)*, pages 408–418.

Pharr, M. and Mark, W. R. (2012). ispc: A SPMD Compiler for High-performance CPU Programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13.

Püschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. (2005). SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275.

Quinlan, D. J., Schordan, M., Philip, B., and Kowarschik, M. (2003). The Specification of Source-to-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications. In Dietz, H. G., editor, *Languages and Compilers for Parallel Computing*, pages 383–394, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. (2017). Halide: Decoupling Algorithms from Schedules for High-performance Image Processing. *Commun. ACM*, 61(1):106–115.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA. ACM.

Rawat, P., Kong, M., Henretty, T., Holewinski, J., Stock, K., Pouchet, L.-N., Ramanujam, J., Rountev, A., and Sadayappan, P. (2015). SDSLc: A Multi-target Domain-specific Compiler for Stencil Computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, pages 6:1–6:10, New York, NY, USA. ACM.

Reynders, J. V. W. and Cummings, J. C. (1998). The POOMA framework. *in Comput. Phys.*, 12:453–459.

Rich Hickey (2007). The Clojure Programming Language. `https://clojure.org/`.

Robison, A. D. (2013). Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science Engineering*, 15(2):66–71.

Rosales, C., Cazes, J., Milfeld, K., Gómez-Iglesias, A., Koesterke, L., Huang, L., and Vienne, J. (2016). A Comparative Study of Application Performance and Scalability on the Intel Knights Landing Processor. In *Lecture Notes in Computer Science*, pages 307–318. Springer International Publishing.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA. ACM.

Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M. (2018). Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR*, abs/1805.00907.

Roth, G., Mellor-Crummey, J., Kennedy, K., and Brickner, R. G. (1997). Compiling Stencils in High Performance Fortran. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, pages 1–20, New York, NY, USA. ACM.

S. Haney J. Crotinger, S. K. and Smith, S. (1999). Easy Expression Templates Using PETE, the Portable Expression Template Engine. *Technical Report LA-UR-99-777*.

Sheard, T. and Peyton Jones, S. (2002). Template Meta-programming for Haskell. pages 1–16.

So, B., Hall, M. W., and Ziegler, H. E. (2004). Custom Data Layout for Memory Parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 291–, Washington, DC, USA. IEEE Computer Society.

Stone, J. E., Gohara, D., and Shi, G. (2010). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73.

Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. (2014). Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25.

Sujeeth, A. K., Gibbons, A., Brown, K. J., Lee, H., Rompf, T., Odersky, M., and Olukotun, K. (2013). Forge: Generating a High Performance DSL Implementation from a Declarative Specification. *SIGPLAN Not.*, 49(3):145–154.

Sung, I.-J., Stratton, J. A., and Hwu, W.-M. W. (2010). Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 513–522, New York, NY, USA. ACM.

Taha, W. and Sheard, T. (1997). Multi-stage Programming with Explicit Annotations. *SIGPLAN Not.*, 32(12):203–217.

Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. (2011). The Pochoir Stencil Compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA. ACM.

The Khronos Group (2015). The OpenCL Specification v2.1. `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`.

The LLVM Foundation (2018). The LLVM Compiler Infrastructure. `llvm.org`.

The R Foundation (2018). The R Project for Statistical Computing. `https://www.r-project.org/`.

van der Vorst, H. (1992). Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644.

van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, 13(2):22–30.

Vandevoorde, D. and Josuttis, N. M. (2002). *C++ Templates: The Complete Guide*. Addison-Wesley Professional.

Veldhuizen, T. (1995). Expression Templates. *C++ Report*, 7:26–31.

Veldhuizen, T. (2006). Blitz++ library. `http://blitz.sourceforge.net/resources/blitz-0.9.pdf`.

Veldhuizen, T. L. and Gannon, D. (1998). Active Libraries: Rethinking the Roles of Compilers and Libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, OO'98*. SIAM Press.

Verdoolaege, S. (2010). *isl: An Integer Set Library for the Polyhedral Model*, pages 299–302. Springer Berlin Heidelberg, Berlin, Heidelberg.

Walter, J. and Koch, M. (2012). uBLAS BOOST library. `http://www.boost.org/doc/libs/1_57_0`.

Wiemann, P., Wenger, S., and Magnor, M. (2011). CUDA Expression Templates. In *WSCG Communication Papers Proceedings*, pages 185–192. ISBN 978-80-86943-82-4.

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline. *Commun. ACM*, 52(4):65.

Winter, F. T., Clark, M. A., Edwards, R. G., and Joó, B. (2014). A Framework for Lattice QCD Calculations on GPUs. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE.

Wolf, M. E. and Lam, M. S. (1991). A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA. ACM.

Xu, S. and Gregg, D. (2014). Semi-automatic Composition of Data Layout Transformations for Loop Vectorization. In *Network and Parallel Computing*, pages 485–496. Springer Science + Business Media.

Yelick, K. A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. (1998). Titanium: A High-Performance Java Dialect. In *In ACM*, pages 10–11.

Yount, C. (2015). Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 865–870.