POLYHEDRAL+DATAFLOW GRAPHS

by

Eddie C. Davis

A dissertation

submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Computing

Boise State University

May 2020

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the dissertation submitted by

Eddie C. Davis

Dissertation Title: Polyhedral+Dataflow Graphs

Date of Final Oral Examination: 6th December 2019

The following individuals read and discussed the dissertation submitted by student Eddie C. Davis, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Catherine RM. Olschanowsky, Ph.D.	Chair, Supervisory Committee
Elena Sherman, Ph.D.	Member, Supervisory Committee
Steven Cutchin, Ph.D.	Member, Supervisory Committee
Donna Calhoun, Ph.D.	Member, Supervisory Committee

The final reading approval of the dissertation was granted by Catherine RM. Olschanowsky, Ph.D., Chair of the Supervisory Committee. The dissertation was approved by the Graduate College.

Dedicated to Elaina, Arianna, and Zora

ACKNOWLEDGMENTS

The author wishes to express gratitude to his advisor, Catherine Olschanowsky, for her unwavering patience, encouragement, and support.

This work has been partially supported by the National Science Foundation under Grant Numbers 1422725, 1563818, and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program under Award Number DE-SC-04030, and contract number DE-AC02-05-CH11231. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Department of Energy.

ABSTRACT

This research presents an intermediate compiler representation that is designed for optimization, and emphasizes the temporary storage requirements and execution schedule of a given computation to guide optimization decisions. The representation is expressed as a dataflow graph that describes computational statements and data mappings within the polyhedral compilation model. The targeted applications include both the regular and irregular scientific domains.

The intermediate representation can be integrated into existing compiler infrastructures. A specification language implemented as a domain specific language in C++ describes the graph components and the transformations that can be applied. The visual representation allows users to reason about optimizations. Graph variants can be translated into source code or other representation. The language, intermediate representation, and associated transformations have been applied to improve the performance of differential equation solvers, or sparse matrix operations, tensor decomposition, and structured multigrid methods.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

NSCI – National Strategic Computing Initiatitive

ECP – Exascale Computing Project

DOE – Department Of Energy

NNSA – National Nuclear Security Administration

FLOP – Floating Point Operation

AMR – Adaptive Mesh Refinement

SAMR – Structured Adaptive Mesh Refinement

AST – Abstract Syntax Tree

IR – Intermediate Representation

COO – Coordinate Format

CSR – Compressed Sparse Row

CSC – Compressed Sparse Column

CSF – Compressed Sparse Fiber

ELL – ELLPack Format

BSR – Block-compressed Sparse Row

CSB – Compressed Sparse Block

HiCOO – Hierarchical Coordinate Format

SPF – Sparse Polyhedral Framework

PDFL – Polyhedral+Dataflow Language

PDFG – Polyhedral+Dataflow Graph

GeMM – Generalized Matrix Multiplication

LUD – Lower-Upper Decomposition

SVD – Singular Value Decomposition

MPP – Moore-Penrose Pseudoinverse

SOR – Successive Over Relaxation

ODE – Ordinary Differential Equation

PDE – Partial Differential Equation

SGD – Steepest Gradient Descent

SPD – Symmetric Positive Definite

CG – Conjugate Gradient

CPD – Canonical Polyadic Decomposition

PCA – Principal Component Analysis

GMG – Geometric Multigrid

AMG – Algebraic Multigrid

ILP – Integer Linear Programming

GCC – Gnu Compiler Collection

ICC – Intel Compiler Collection

TBB – Thread Building Blocks

RTL	– Register Transfer Language
DAG	– Directed Acyclic Graph
CFG	– Control Flow Graph
DFG	– Dataflow Graph
DSA	– Dynamic Single Assignment
SSA	– Static Single Assignment
DSL	– Domain Specific Language
eDSL	– Embedded Domain Specific Language
PDFG	– Polyhedral+Dataflow Graph
PDFL	– Polyhedral+Dataflow Language
PEP	– Polyhedral Expression Propagation
SCoP	– Static Control Part
CFD	– Computational Fluid Dynamics
FFT	– Fast Fourier Transforms

LIST OF SYMBOLS

α	Step length in conjugate gradient computation
β	Solution improvement in conjugate gradient computation
γ	Conserved to primitive factor in Euler equations
Δ	Laplace scalar operator
λ	Normalization vector in Kruskal tensor
∇	Del operator for gradients
ρ	Density in fluid dynamics applications
$\mathbb{R}$	Set of all real numbers
$\mathbb{Z}$	Set of all integers
Σ	Summation operator
$\prod$	Product operator
$\otimes$	Kronecker product operator
$\odot$	Hadamard product operator
$\dagger$	Moore-Penrose pseudoinverse
$\mathcal{X}$	General tensor

CHAPTER 1

INTRODUCTION

The solutions to many important scientific, engineering, and national security challenges require improvements in the software stack to achieve the computational efficiency necessary for large scale modeling and simulation applications. The National Strategic Computing Initiative (NSCI) prioritizes fields such as molecular dynamics, material science, advanced manufacturing, and precision medicine [1]. The Exascale Computing Project (ECP) is an associated effort to build computational tools that support advances in these fields. Computational efficiency is determined by the number of resources required by an application. More efficient computation means that more data can be processed in less time or with fewer resources. This work aims to improve computational efficiency in scientific applications.

Compilers are crucial components of the software stack, and are responsible for translating code implemented in high-level programming languages to architecture-specific assembly code. During this translation, the computational efficiency of the application can be improved by performing the appropriate set of code optimizations and transformations. The best sequence of optimizations depends on the target architecture. There has been an increase in architecture variability and complexity in recent years. CPU memory hierarchies have become more complex, and scientific applications often target alternative architectures including graphics processors and field-programmable gate arrays. More compiler internal representations are required to select and apply effective code transformations.

Dataflow optimizations are especially beneficial for memory bound applications, which are those that move relatively large quantities of data per each unit of arithmetic computation. This low computational intensity means that processors spend significant amounts of time waiting for data to become available for computation. By contrast, compute bound applications are limited by the rate at which processors can perform arithmetic operations on data that are already available [2].

Significant performance gains in memory bound applications can be achieved by dataflow optimizations that reorganize computations and reduce storage requirements. Scientific applications contain common computational patterns that enable these types of optimizations, such as linear algebra operations, or stencil computations. The calculations are typically implemented as a series of nested loops over the data, with each loop nest computing a portion of the solution. To take advantage of these patterns, computations performed across large data domains are often distributed across many compute nodes in a network or cluster. The performance of shared memory programs is crucial for scalability since the time and energy lost to poor single node performance is multiplied when the code is distributed.

Memory access patterns are critical to application performance and scalability. Applications with predictable data accesses and control flow patterns can be statically analyzed and optimized at compile time. These applications have regular patterns and so are considered regular. Other applications that rely on pointers or other indirect memory access patterns cannot always be statically analyzed. Such applications are referred to as irregular. These computations may require run time information or domain specific knowledge to be successfully optimized [3]. Both classifications of memory access patterns are considered in this work.

Transforming readable and maintainable application source code into fast and energy-efficient machine code is challenging for compilers, due in part to existing programming language designs. A unified representation grants programmers control over memory interactions and execution schedules, but burdens them with the responsibility to write efficient code. This is particularly difficult for domain experts with no background in computer science or software engineering. These users may instead wish to convert mathematical expressions into algorithmic representations without being concerned with programming abstractions or hardware performance. This separation of concerns enables domain experts to write algorithms and performance engineers to apply optimizations, resulting in performance portability. Achieving this portability requires trade-offs between memory storage and computation, which is important for computationally intensive science and engineering applications.

The effectiveness of dataflow optimizations greatly depends on the abstraction level at which the code is analyzed. Higher abstraction levels such as the source code better capture the intention of the computation. However, the source code representation of scientific computations should not be altered to enable optimizations, improve performance, or target different hardware architectures. Such changes can make the source code difficult to understand, maintain, and update by domain scientists or engineers. Performing compiler optimizations at the instruction level can lead to ineffective dataflow optimizations. This leaves the programmer to perform dataflow optimizations at the source code level. Instead, transformations should be applied to a higher level intermediate representation (IR) within the compiler, or via an abstraction layer that enables a performance expert to tune the applications. These goals can be achieved by decoupling the algorithmic specification from the execution schedule and data layout [4–6].

This research targets compiler transformations focused on dataflow optimizations for memory bound applications. A compiler intermediate representation was designed and developed to enable code transformations in existing applications using a combination of program analysis, performance modeling, and programmer feedback. Domain experts can implement computations in the provided specification language, while performance engineers can transform code by manipulating the intermediate representation. The dataflow optimizations were applied to improve the performance of finite difference and structured grid solvers, and sparse linear algebra applications. The specific research contributions are described in the following section.

1.1 Contributions

1. Development of the polyhedral+dataflow graph intermediate representation (PDFG-IR) that expresses execution schedules, dataflow, memory interactions, and program statements in a manner that expands the set of automated transformations available to optimizing compilers. The polyhedral model is combined with macro-dataflow graphs to explicitly represent data requirements, including data type, domain, and size. The graphs encapsulate code with execution schedules and data mappings for both persistent and temporary storage spaces.

2. Definition and implementation of compiler transformations to modify the execution schedules and storage mappings of the specified computation. These operations include statement rescheduling, producer-consumer and read-reduce loop fusion, and other loop transformations, such as unrolling, splitting, and tiling. Storage reductions are determined using reuse distance and reachability

analyses. A memory allocation algorithm based on liveness analysis [7] is described that allocates sufficient space for those data that are live at each point in the computation.

3. Extension of the IR to support irregular applications using the inspector/executor approach [8]. The inspector-executor method is applied when code or data transformations require run time support, including run time dependence analysis or data transformations. Both inspectors and executor components can be represented and optimized, by transforming both data and iteration spaces. Non-affine, data dependent loop bounds are represented by *uninterpreted* functions [9] and converted into *explicit* represetations at run time.

4. Development of an embedded domain specific language to construct the IR. Numerical algorithms are expressed in C++ using a combination of iterators, functions, constraints, spaces, and executable statements. An iteration space is composed of an iterator set and their corresponding boundary constraints. Data spaces are derived from access functions in the statements. A computation consists of an iteration space, execution schedule, and statement list. The PDFG-IR is generated from the eDSL specifications.

5. Generation of an internal performance model for each graph variant. Many different graph variants can be generated from an initial graph specification by applying the supported transformations. A performance model is generated for each variant that include estimates of floating point operations (FLOPs), memory throughput, and arithmetic intensity. The model can be used to reason about the performance of a given variant on a target architecture.

1.2 Dissertation Structure

The remainder of this dissertation is organized as follows. Chapter 2 provides background on the numerical methods targeted in this work, the polyhedral model and associated compiler technologies, sparse matrix and tensor representations, the inspector-executor approach, dataflow languages, and memory optimizations. The polyhedral+dataflow IR and its application to regular applications is described in Chapter 3. The polyhedral+dataflow language (PDFL) and PDFG-IR extensions to support irregular applications are detailed in Chapter 4.

The integration of PDFG-IR with a structured grid adaptive mesh refinement solver (SAMR) eDSL called `Proto` is described in Chapter 5. Case studies including conjugate gradient (CG) and canonical polyadic tensor decomposition (CPD) implementations are performed in Chapter 6. A survey of related work is provided in Chapter 7. Finally, topics for future work are discussed, and conclusions are drawn in Chapter 8.

CHAPTER 2

BACKGROUND

This chapter provides an overview of the concepts applied in this work. The research has built on recent developments in polyhedral compiler frameworks [10–12], loop chains [13–15], domain specific languages [4–6], dataflow programming languages [16–18], memory access optimizations [19–21], stencil-based partial differential equation (PDE) solvers [22–25], inspector/executor applications [26–28], code generation with non-affine or data-dependent loop bounds [8, 29, 30], and sparse matrix or tensor optimizations [31–33]. The applications targeted for optimization by this work include numerical methods for mathematical solvers, scientific and engineering simulations, or data analytics.

2.1 Compiler Optimizations

A typical optimizing compiler consists of at least three stages. The source code in the chosen programming language is parsed and converted into an initial intermediate representation (IR), such as an abstract syntax tree (AST). The IR may undergo many transformations, and several different representations as the code progresses through a series of optimization passes. The final IR is passed to the code generator that then emits machine instructions for the target architecture. An example is the register transfer language (RTL) of the Gnu Compiler Collection (GCC). Finally, the compiled code is linked with any external libraries to produce an executable binary file. An overview of this process is given in Figure 2.1.

The correctness of every transformation performed by a compiler must be guaranteed. The soundness of an optimization assures that the runtime behavior of the program is not modified. This requirement forces conservative decisions to be made during static analysis passes that occur at compile time. These passes are applied by traversing the various internal representations in the middle-end of the compiler infrastructure. A trade-off between precision and compile time overhead exists for each optimization.

Compiler optimizations can be limited by the high level language semantics. The use of pointers in C, for example, presents a significant challenge to data dependence analyses. If the compiler cannot guarantee that two pointers do not address overlapping memory spaces within a reasonable amount of time, it must assume that the pointers are aliased. This assumption limits possible transformations, and therefore the run time performance.

Programmers may conservatively allocate more space than necessary, rather than formally analyzing the space requirements of an algorithm. A compiler that supports

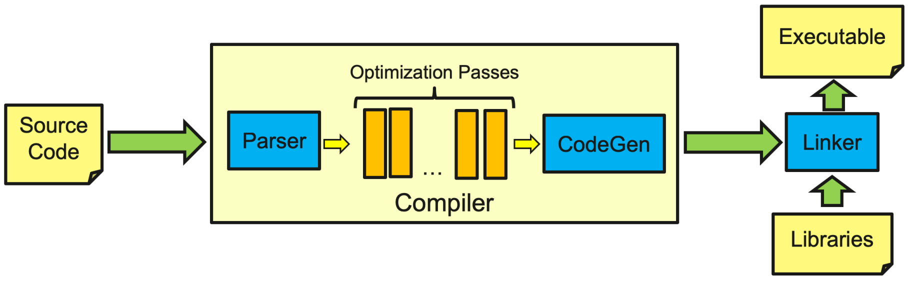


Figure 2.1: Process flow of an optimizing compiler with parser (front), optimization passes (middle), code generator, and linker (back-end components).

data transformations can help overcome such challenges by automatically reducing the amount of temporary storage allocated, or reordering the data into a form that is more efficient for the given computation and target architecture. The IR described in this work is designed to apply such transformations.

Unnecessary data movement at the application level consumes large quantities of time and energy. Memory interactions at this level are difficult to understand, reason about, and therefore optimize. Irregular applications are characterized by non-sequential memory access patterns or sparse data structures that cannot be statically analyzed in a straightforward manner. The problem is compounded by the difficulty of communicating dataflow information to the middle-end of an optimizing compiler. The methods to provide information such as data access mappings or read/write patterns are limited in existing programming languages.

Compiler IRs are data structures that represent a series of machine instructions coded in a programming language [34]. ASTs are recursive data structures that represent the syntactic structure and content of a program. A basic block is a sequence of instructions with no control flow (branches) except the entry and exit points. A control flow graph (CFG) describes execution order, and can be obtained by traversing the AST. Each CFG node represents a basic block of the program, and each edge indicates the transfer of control from one block to another. The CFG is translated into a static single assignment (SSA) form [35], such as GIMPLE in GCC or LLVM-IR in Clang.

Dataflow analysis is performed on the CFG to produce a dataflow graph (DFG). The flow of data blocks during program execution are described by the DFG structure. The SSA form enables more efficient data dependence analysis. Dataflow analysis is limited even in SSA form, and much of the data movement is left to the programmer to define.

Programming languages such as C and Fortran do not provide the necessary dataflow information to the compiler middle-end. Two potential soultions to this problem are code annotations and embedded DSLs. The programmer can annotate the source code with pragmas or decorators, for example, to indicate vectorization opportunities, identify static control parts of programs (SCoPs), or provide dataflow information [36,37]. These annotations are ignored by the general purpose compilers.

Irregular applications that require sparse or unstructured computations are important in scientific simulations and analytics. These applications reduce data storage by only storing nonzero data elements [38]. The element locations are stored in index data structures, requiring indirect memory accesses. The resulting code contains data dependent loop bounds that cannot be statically analyzed by an optimizing compiler.

The inspector/executor approach addresses this problem by enabling compiler transformations at run time. An *inspector* can observe data access patterns, perform dependece analysis, or apply run time data transformations. The *executor* performs the computationally intensive computation on the data transformed by the inspector [27]. Statically optimized inspectors and efficient executors can be produced at compile time [39].

2.2 Polyhedral Model

The polyhedral compilation model [40] is a mathematical framework for describing complex applications with multiple operations and loop nests in a compact form. An affine transformation is a linear mapping that preserves points, lines, and planes [41]. Loop iterations are represented as lattice points within a polyhedron. Affine transformations can be applied to the polyhedra, enabling loop optimizations such as fusion and tiling. The model provides a means of applying loop transformations based on affine spaces defined by integer sets. An iteration space that describes a loop nest can be considered an affine space, an integer set of tuples $(i_1,...,i_n) \in \mathbb{Z}^n$.

A loop nest can be represented with the following components:

1. *Iteration Space*: the set of statements in the section, and the loop iterations where instances of the statement are executed. These are specified with named union sets. An integer set, *I*, is defined as

$$I = \{ \ [i_1, \ldots, i_n] \mid c_1 \ \wedge \ \ldots \ \wedge c_m \ \} \tag{2.1}$$

   Where $i_1, \ldots, i_n$ are indices, or iterators, in the n dimensions of the set, and $c_1, \ldots, c_m$ are the affine inequalities, or constraints, that bound the integers in the set. Integer sets are typically expressed as Presburger formulae [42].

2. *Access Relations*: The set of reads, writes, and may-writes that relate statement instances in the iteration space to data locations. These are represented by mapping functions or relations. An integer relation is denoted by the mapping

$$R = \{ \ [i_1, \ldots, i_n] \ \rightarrow \ [j_1, \ldots, j_k] \mid c_1 \ \wedge \ \ldots \ \wedge c_m \ \} \tag{2.2}$$

   Where $(j_1, \ldots, j_k)$ is the integer tuple in the destination set. A mapping from a dense matrix, A, accessed at indices $(i,\ j)$, to a sparse matrix A', for example, is defined as

$$R_{A\to A'} = \{ \ [i,j] \ \to \ [i',j'] \mid 0 \leq m < M \ \wedge \tag{2.3}$$

$$i' = row(m) \wedge \ j' = col(m) \ \} \tag{2.4}$$

Where M is the number of nonzero values, and *row*, *col* are the respective row and column indices.

3. *Dependences*: The set of data dependences that impose restrictions on the execution order, e.g., producer- consumer relationships. Dependences can be modeled with maps or edges in a dataflow graph. An array A, read at each point $(t,\ i,\ j)$ of an iteration space I, would be represented by the mapping

$$R_{I\to A} = \{ \ I[t,i,j] \to A[i,j] \ \} \tag{2.5}$$

4. *Schedule*: The execution order of each statement instance can be represented by a lexicographically ordered set of tuples in a multidimensional space [43]. Lexicographic ordering ($\prec$) is defined as

$$(a_1, \ldots a_n) \prec (b_1, \ldots b_m) \iff \exists i \mid 1 \leq i \leq min(n,m) \text{ s.t.} \tag{2.6}$$
$$(a_1, \ldots a_{i-1}) = (b_1, \ldots, b_{i-1}) \text{ and } a_i < b_i$$

A statement, S executed at every point in iteration space, I, would have the scheduling function

$$T_{I\to S} = \{ \ I[t,i,j] \to [t,0,i,0,j] \ \} \tag{2.7}$$

The polyhedral model provides a separation of concerns between the statement instances and the corresponding execution order. Polyhedral optimizations change the execution schedule without affecting the set of statements that are executed [44]. Transformations that involve statement reordering include fission, fusion, skewing, interchange, reversal, and tiling. Polyhedral representations can be extracted from source code by analyzing loop bounds and array subscript expressions [45].

Polyhedral code generators such as CLooG [46,47] or Omega [48] apply algorithms that can construct an AST from polyhedra by combining *if* nodes for conditional statements, *for* nodes for loop nests, *block* nodes representing compound statements or basic blocks for loop bodies [43]. Transformations that can be applied outside of the polyhedral framework include loop unrolling, skewing, and tiling. Polyhedra are converted back into ASTs using quantifier elimination techniques for linear inequalities such as Fourier-Motzkin or Chernikova's algorithm [49].

Compiler frameworks such as CHiLL [50] or PLuTo [10] have been built using the polyhedral model. The loop chain abstraction [13, 51] applies transformations to series of loops referred to as loop chains, and demonstrates the potential impact of these transformations on both regular and irregular applications. The loop chain compiler was built on the integer set library (ISL) [52]. The Polly [53, 54] interface for LLVM [55] has demonstrated the applicability of the polyhedral model within compiler optimization passes.

Domain specific languages (DSLs) target a particular problem space. Halide [4,56] and PolyMage [5, 57], for example, are DSLs built specifically for image processing pipelines. Constructing an entirely new compiler is a considerable software engineering challenge, so DSLs are often embedded within existing languages such as C++ or Python (eDSLs).

2.3 Dataflow Languages

The dataflow programming paradigm was motivated by the need to expose parallelism [58]. Early dataflow architectures exhibited poor performance in cases of fine-grained parallelism. A study by Sterling et al. [59] indicated that balancing task granularity was the critical factor in the performance of dataflow programs. A hybrid dataflow / von Neumann approach has since emerged, allowing developers to benefit from both coarse-grained dataflow parallelism at the macro-level and fine-grained instruction level parallelism. Dataflow programming is similar to functional programming in that the code is free of side effects and variables can only be assigned once. In this research, the execution schedule as described in subsection 2.2, is determined by the data dependences from the dataflow graph.

A dataflow graph is an intermediate representation that follows the flow of data through a function or procedure to identify dependences. Statement level dataflow graphs are directed acyclic graphs with nodes at the iteration granularity. Macro dataflow graphs were introduced to coarsen the granularity by grouping iterations into a single node [60]. Functional representations of an application can be translated into macro dataflow graphs. This representation can be traversed to exploit parallelism and assist the associated code generation [61].

Prasanna et al. [62] take a hierarchical approach. Each macro node is scheduled for parallel execution on a machine, unlike the previous work that assumed sequential execution. The entire graph is then partitioned and scheduled for distributed memory execution.

2.4 Memory Optimizations

The clock speeds of microprocessors have increased exponentially since the advent of Moore's law [63], however, off-chip memory performance has not achieved the same rate of improvement. Deeper memory hierarchies have been introduced to bridge the gap. The memory levels between the CPU registers and main memory, collectively referred to as cache, are larger but slower near the bottom, and faster but smaller toward the top. Cache replacement protocols are responsible for transferring blocks of data between cache and DRAM in an effort to ensure that the most frequently accessed data can be retrieved quickly. This leads to the concept of data locality [64].

Compilers are responsible for generating code that reduces both the number of cache misses and the impact of unavoidable misses. This can be accomplished by maximizing data reuse, and ensuring that data are stored contiguously for each process. Techniques to achieve these goals include automatic data layout [65–67], affine partitioning, and loop blocking or tiling. Lattice-based memory optimization [19, 68, 69] is an approach to affine partitioning. Polyhedral data reuse [20] and associative reordering [70] are other methods to improve data reuse. Contiguous data transformation techniques include permutation, strip-mining, and compiler-directed page coloring [71, 72].

2.5 Target Computational Patterns

This work targets a subset of scientific computing methods that share common computational patterns. Colella identified the *seven motifs* [73] of scientific computing as structured and unstructured grids, dense and sparse linear algebra, fast Fourier transforms (FFT), particle interactions, and Monte Carlo simulations. This work focuses on stencil computations within structured grids, and linear algebra applications.

2.5.1 Iterative Methods

Many mathematical problems cannot be solved analytically. Numerical methods are algorithms designed to solve systems of equations with arbitrary precision using successive approximations [74]. Iterative methods consist of an acceptable error threshold for convergence, and a maximum number of steps or iterations. An initial estimate can be provided, based on domain knowledge or assigned randomly. Open methods find the roots, or zeros, of a function within a fixed interval given an initial estimate. Open methods include bisection, Newton-Raphson, the secant, or Brent's method [75], also known as the *zeroin* algorithm [76]. Open methods can be applied to linear or nonlinear systems. An example implementation of the Newton-Raphson method in C is given in Figure 2.2.

```c
#define T 500

double tol = 1e-5;
double err = 1.0;
double x = x0; // Initial guess

for (t = 1; t <= T && fabs(err) >= tol; t++) {
  // x(i+1) = x(i) - f(x) / f'(x)
  err = func(x) / deriv(x);
  x -= err;
}
```

Figure 2.2: Source code for Newton-Raphson method.

2.5.2 Gradient Methods

Gradient methods compute derivatives to find local optima [77]. The gradient descent algorithm can be used to find minima, or the steepest ascent (hill-climbing) for maxima. Powell's conjugate direction method [78] is an algorithm for finding local minima that is applicable when the function is discontinuous, non-differentiable, or no information about the derivative is available. It is an efficient, quadratic method ($O(n^2)$ convergence) that is often paired with Brent's method as its search technique due to its linear time complexity.

The conjugate-gradient method is composed of three fundamental operations, scalar multiplication, sparse-matrix vector multiplication, and the inner product. The *inner product* of two vectors, denoted $x^T y$, is computed as the scalar sum $\sum_{i=1}^{N} x_i y_i$. The matrix, A, is symmetric positive definite (SPD), if $x^T A x > 0$ for every nonzero vector, x. The domain of possible solutions to the unknown vector, x, can be expressed in the quadratic form, $\frac{1}{2}x^T A x - b^T x + c$, where c is a scalar constant. The gradient, or first derivative, is $f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b$. Since A is symmetric, this reduces to $f'(x) = Ax - b$. Therefore, the system is solved by finding the vector, x, that sets the gradient to zero. In other words, $f(x)$ is minimized when the gradient $f'(x) = 0$, or $Ax = b$.

Gradient methods find this critical point by selecting an arbitrary initial solution, x_0, and making a series of steps, $x_1, x_2, ..., x_n$, computing the residual after each, and stopping at some maximum number of iterations or until the error is within some acceptable tolerance with respect to the actual x. Each new search direction is constructed from the residual at each step of the CG algorithm. Successive directions are orthonormal to all previous search directions, ensuring that the same direction will not be followed more than once, and thus accelerating convergence.

Conjugate gradient is among the most popular methods for solving large systems of linear equations in the form, $Ax = b$, where x is an unknown vector to be solved, b is known, and A is a square $(N \times N)$, SPD matrix of known values. [79]. Iterative methods like CG or Jacobi are best-suited to systems involving sparse matrices. Dense matrices can be solved more efficiently using direct methods such as factorization and backsubstitution, e.g., lower-upper (LU) decomposition [80]. Multifrontal methods are an efficient approach to LU factorization in sparse systems [81].

The CG algorithm is summarized in equations 2.8–7, where the zero subscript represents the initialization step, i is the current iteration, and n the maximum number of iterations. The vectors, d, r, s, x represent the search direction, residual, step, and approximate solution, respectively. The scalar α is the step length, and β the improvement in the solution over the previous iteration. The process continues until the residual error falls below a given threshold, or n iterations have been performed.

$$d_0 = r_0 = b - Ax_0 \tag{2.8}$$

$$s_i = Ad_{i-1} \tag{2.9}$$

$$\alpha_i = \frac{r_{i-1}^T r_{i-1}}{d_{i-1}^T s_i} \tag{2.10}$$

$$x_i = x_{i-1} + \alpha d_{i-1} \tag{2.11}$$

$$r_i = r_{i-1} - \alpha s_i \tag{2.12}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}} \tag{2.13}$$

$$d_i = r_i + \beta d_{i-1} \tag{2.14}$$

Given an initial guess of zero, $x_0 = \vec{0}$, equation 2.8 simplifies to $d_0 = r_0 = b$. The remainder of the algorithm consists of one matrix-vector product (equation 6.2), five dot products, three vector additions, and three scalar-vector products per iteration.

2.5.3 Finite Difference and Volume Methods

Conservation problems in physics are often expressed as partial differential equations, that must be solved with computational methods when analytical solutions are unavailable. The Navier-Stokes equations describe the flow of viscous fluids, including the conservation of mass, energy, and momentum [82]. The general form is given as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}(\mathbf{U}) = 0 \tag{2.15}$$

where $\mathbf{U}$ is the vector of conserved unknowns, t is time, ∇ is the differential operator *nabla*, and $\vec{\mathbf{F}}$ is the flux dyad tensor in each spatial direction.

These equations can be solved numerically by approximating a sequence of algebraic equations at discrete locations on a structured grid over the spatial domain. The point-wise approximations are known as *finite-differences*, and are derived from Taylor-series expansion,

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_i = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x} + O(\Delta x^2) \tag{2.16}$$

where i is a discrete grid location in one dimension, and $O(\Delta x^2)$ is the truncation error introduced by a continuous equation approximation in a discretized algebraic form. The error magnitude is a quadratic function of the grid spacing, Δx.

The *finite-volume* method is an alternative to *finite-difference*, where the solution is approximated by integrating over a small control volume, V_i defined by the grid.

Integrals are defined as

$$\frac{\partial}{\partial t}\int_{V_i}\mathbf{U}\mathrm{d}\boldsymbol{x} + \int_{V_i}\nabla\cdot\vec{\mathbf{F}}\mathrm{d}\boldsymbol{x} = \frac{\partial}{\partial t}\int_{V_i}\mathbf{U}\mathrm{d}\boldsymbol{x} + \int_{\partial V_i}\vec{\mathbf{F}}\cdot\hat{n}\,\mathrm{d}\mathcal{S} = 0 \qquad (2.17)$$

with the equation on the right obtained by applying Gauss' divergence theorem. The $\nabla\cdot\vec{\mathbf{F}}$ integral is converted into the normal component integral of the $\vec{\mathbf{F}}$ vector over the control volume surface, where $\hat{n}$ is a unit normal vector pointing outward from the volume. The change of the unknown vector, $\mathbf{U}$ over time is equal to the total flux of $\vec{\mathbf{F}}$ crossing the surfaces in the same time span.

Equations are solved in the *finite-volume* approach by discretizing the volume and approximating $\vec{\mathbf{F}}$ on the control volume faces from the values stored in adjacent cells. The *finite-volume* method incurs a truncation error and implied stencil width. The advantage is that a local conservation property ensures discrete conservation across the entire domain as in the actual PDEs. Figure 2.3 depicts this process in two dimensions (x,y).

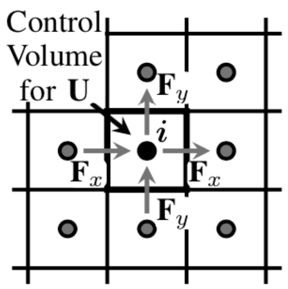


Figure 2.3: Cell fluxes across surface faces of control volume.

PDEs are solved in practice by dividing the grid into discrete cells, iterating over them, and evaluating the algebraic equations for a specified number of iterations. Large domains can be partitioned into smaller pieces referred to as *boxes*. These boxes are padded with layers of *ghost* cells to reduce communication overhead and enable parallel execution.

2.5.4 Structured Grid Methods

Iterative methods can provide an effective means for solving large systems of equations. However, convergence can be slow, requiring $O(N^2)$ iterations, which can be unacceptable for some problems. Multigrid methods allow iterations to change from a fine grid to a coarse grid, with the benefits of reducing convergence to $O(N)$ iterations and improving performance. Multigrid is particularly effective on sparse, symmetric systems [83]. A structured grid solver for the Euler equations is described in Chapter 5.

The Jacobi, Gauss-Seidel, or successive over-relaxation (SOR) stencil computations can be applied at each step to solve a linear system $A_h x = b_h$ to obtain x_h, where h corresponds to the current grid size [74]. These iterative methods are derived from the discretized Taylor series of the function, f, represented by the matrix, A.

$$\begin{aligned}
A^t_{h_{i,j}} &= \frac{1}{4}\left[A^{t-1}_{h_{i+1,j}} + A^{t-1}_{h_{i-1,j}} + A^{t-1}_{h_{i,j+1}} + A^{t-1}_{h_{i,j-1}} - h^2 A_{i,j}\right] && \text{(a)}\\
A^t_{h_{i,j}} &= \frac{1}{4}\left[A^{t-1}_{h_{i+1,j}} + A^{t}_{h_{i-1,j}} + A^{t-1}_{h_{i,j+1}} + A^{t}_{h_{i,j-1}} - h^2 A_{i,j}\right] && \text{(b)}\\
A^t_{h_{i,j}} &= \frac{w}{4}\left[A^{t-1}_{h_{i+1,j}} + A^{t}_{h_{i-1,j}} + A^{t-1}_{h_{i,j+1}} + A^{t}_{h_{i,j-1}} - h^2 A_{i,j}\right] + (1 - wA^{t-1}_{h_{i,j}}) && \text{(c)} \quad (2.18)
\end{aligned}$$

Jacobi is an iterative method that updates the current matrix using only values from the previous time step as seen in Equation 2.18(a). Gauss-Seidel (b) takes advantage of the fact that the values for previous spatial iterations (i-1 and j-1) have already been computed and can perform in place updates, reducing the amount of storage that must be allocated and accelerating convergence. Successive over-relaxation (c) is a Gauss-Seidel refinement that includes a weight term, $1 < w < 2$ that moves the approximation further in the relaxation direction to reduce the number of iterations required for convergence. The $h^2 A_{i,j}$ term in all three equations is the quadratic error.

The use of recently updated values from the same time step in Gauss-Seidel and SOR introduces a race condition resulting in code that is difficult to parallelize. This is resolved by applying *red-black* ordering. A grid point (i, j) is marked red if the sum $(i + j)$ is even and black if it is odd. The red points are updated in the first pass and the black points by reading the red values. Implementations of these smoothing stencils in the C language are shown in Figure 2.4.

```
for (t = 1; t <= T; t++) {
  for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
      A[t,i,j] = (A[t-1,i+1,j] +
        A[t-1,i-1,j] + A[t-1,i,j
            +1] +
        A[t-1,i,j-1]) * 0.25;
} } }
```

(a) Jacobi

```
for (t = 1; t <= T; t++) {
  for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
      A[t,i,j] = (A[t-1,i+1,j] +
        A[t,i-1,j] + A[t-1,i,j+1]
            +
        A[t,i,j-1]) * 0.25;
} } }
```

(b) Gauss-Seidel

```
for (t = 1; t <= T; t++) {
  for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
      A[t,i,j] = (A[t-1,i+1,j] +
        A[t,i-1,j] + A[t-1,i,j+1] +
           A[t,i,j-1]) * w * 0.25;
      A[t,i,j] += (1 - w * A[t-1][i][j]);
} } }
```

(c) Successive over relaxation

Figure 2.4: Source code for 2D smoothing stencils.

2.5.5 Tensor Decomposition

Higher dimensional problems that cannot be represented by matrices are stored in tensors. The rank R, of a tensor $\mathcal{X}$, is the minimum number of indices required to uniquely identify every element in the tensor. The order, d is the number of modes, or dimensions in the tensor. Tensors are a generalization of matrices to higher orders, where scalars are zero order, vectors are first order, and matrices are second order. Tensors can be simplified by decomposing them into a sequence of operations on lower order structures. Tensor decomposition is a generalization of matrix decomposition techniques such as singular value decomposition (SVD) or principal component analysis (PCA) [84].

A tensor can be matricized, for example by taking a slice, $\mathcal{X}(:,j,k)$. In general, a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ can be matricized into a matrix $A \in \mathbb{R}^{N_1 \times N_2} | N_1 N_2 = n_1 \ldots n_d$. A tensor can be decomposed into the sum of rank one tensors (vectors), $\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{x_1}_r \circ \ldots \circ \mathbf{x_d}_r$. The corresponding factor matrices, $\mathbf{X_{(1)}} = [\mathbf{x_1}_1 \ldots \mathbf{x_1}_r], \ldots, \mathbf{X_{(d)}} = [\mathbf{x_d}_1 \ldots \mathbf{x_d}_r]$ are formed from the component vectors, where $\mathbf{X_{(n)}}$ is the mode-n matricization. Given a third order tensor, $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, let $\mathbf{A}_{I \times R}, \mathbf{B}_{J \times R}, \mathbf{C}_{K \times R}$ denote the factor matrices and $\mathcal{X}(i,j,k)$ each tensor element.

The Kronecker product, $\mathbf{A} \otimes \mathbf{B}$ is a generalization of the outer, or Hadamard product, resulting in a block matrix of size $IJ \times R^2$. The Khatri-Rhao product is the column-wise Kronecker product, $\mathbf{A} \odot \mathbf{B} = [\mathbf{A}(:,1) \otimes \mathbf{B}(:,1) \ldots \mathbf{A}(:,R) \otimes \mathbf{B}(:,R)]$ producing a block matrix with dimensions $IJ \times R$. The CPD produces one factor matrix per mode.

$$\mathbf{X}_{(1)} \approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\intercal \rightarrow \mathbf{A} \approx \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \tag{2.19}$$

$$\mathbf{X}_{(2)} \approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^\intercal \rightarrow \mathbf{B} \approx \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A}) \tag{2.20}$$

$$\mathbf{X}_{(3)} \approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^\intercal \rightarrow \mathbf{C} \approx \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A}) \tag{2.21}$$

This computation requires three applications of the matricized tensor times Khatri-Rhao product (MTTKRP), the bottleneck in many tensor decomposition applications. The columns of the factor matrices are often normalized to a length of one, with weights stored in a vector $\lambda \in \mathbb{R}^R$, where the matrix $\mathbf{\Lambda} = diag(\lambda)$. Factor matrices are held constant while a new one is computed for the current mode. This reduces the problem to linear least-squares, leading to the alternating least squares (ALS) algorithm for computing CPD. The minimization problem can be expressed in the form $\min_{\tilde{\mathbf{A}}} \|\mathbf{X}_{(\mathbf{1})} - \tilde{\mathbf{A}}(\mathbf{C} \odot \mathbf{B})^\top\|_F$ where $\tilde{\mathbf{A}} = \mathbf{A} \cdot \mathbf{\Lambda}$ and $\| \cdot \|_F$ is the Frobenius norm. The optimal solution is given by

$$\tilde{\mathbf{A}} = \mathbf{X}_{(\mathbf{1})}[(\mathbf{C} \odot \mathbf{B})^\intercal]^\dagger = \mathbf{X}_{(\mathbf{1})}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^\dagger \tag{2.22}$$

where $\dagger$ denotes the Moore-Penrose pseudoinverse. The pseudoinverse reduces the complexity since only an $R \times R$ matrix is needed, rather than $JK \times R$.

The pseudoinverse is computed by applying the SVD to decompose the matrix A into two unitary matrices U, V, and a diagonal matrix Σ, such that $A = U\Sigma V^*$, where * represents the conjugate transpose, or simply the transpose for real valued matrices. The values in Σ are singular, and the columns of U, V are the left and right singular vectors, respectively. These are approximations of the eigenvalues and eigenvectors. The Moore-Penrose pseudoinverse is defined as $A^\dagger = V\Sigma^\dagger U^*$ [85].

The normalization step is computed as $\lambda_r = \|\tilde{\mathbf{a}}\|$ and $\mathbf{a}_r = \tilde{\mathbf{a}}_r/\lambda_r$. The factor matrices can be initialized to zero, randomly, or to the R leading left singular vectors of $\mathbf{X}_{(n)}$. The large quantities of data produced by the Khatri-Rhao products are known as the intermediate data explosion problem [86]. The source code for a possible implementation of the CPD algorithm focused on the MTTKRP kernel is given in Figure 2.5. The code is for a four dimensional tensor, $\mathbf{X}$, of shape (I,J,K,L) and rank, R, with factor matrices $A,\ B,\ C,\ D$. The tensor and factor matrices are all dense in this example.

The generalized CP-ALS decomposition algorithm into rank, R, components, for an Nth order tensor, $\mathbf{X}$, of shape $(I_1,I_2,\ldots,I_{\mathrm{N}})$, with maximum number of iterations, T, is given in Figure 2.6. The algorithm produces the normalization vector, λ, and factor matrices, $A_1,\ A_2,\ldots,\ A_N$.

```
for (i = 0; i < I; i++)
 for (j = 0; j < J; j++)
  for (k = 0; k < K; k++)
   for (l = 0; l < L; l++)
    for (r = 0; r < R; r++)
     A[i,r] += X[i,j,k,l]*B[j,r]*C[k,r]*D[l,r];
```

Figure 2.5: Source code for Matricized Tensor Times Khatri-Rhao product for fourth order tensor.

2.6 Sparse Matrix Formats

The particular structure of a matrix can be exploited to apply more efficient solving techniques to improve computational performance. Diagonal, or banded matrices are sparse, except for bands around the main diagonal. The distance from the main diagonal of the most distant value is known as the bandwidth. Tridiagonal

```
function cp_als($\mathbf{X}, R$) {
  for ($n = 1, \ldots, N$) {
    $A_n = \mathbf{rand}(I_n \times R)$
  }
  for ($t = 1, \ldots, T \wedge \epsilon > \tau$) {
    for ($n = 1, \ldots, N$) {
      $V \leftarrow A_1^\top A_1 * \ldots * A_{n-1}^\top A_{n-1} * A_{n+1}^\top A_{n+1} * \ldots * A_N^\top A_N$
      $A_n \leftarrow \mathbf{X}_{(n)}(A_1 \odot \ldots \odot A_{n-1} \odot A_{n+1} \odot \ldots \odot A_N)V^\dagger$
      $\lambda \leftarrow \|A_n\|_F$
      $A_n \leftarrow A_n / \lambda$
    }
    $\epsilon = \mathbf{fit}(t) - \mathbf{fit}(t-1)$
  }
  return $\lambda, A_1, A_2, \ldots, A_N$
}
```

Figure 2.6: CP-ALS decomposition algorithm with R components for Nth order tensor, $\mathbf{X}$.

matrices, for example, have a bandwidth of 3 and occur frequently in engineering applications. The Thomas algorithm [87] can efficiently solve tridiagonal matricxes.

Symmetric matrices require only half of the elements to be stored, since each value $a_{ij} = a_{ji}$, and can be efficiently solved using Cholesky decomposition with elimination trees [88]. The Cholesky algorithm has the additional constraint that the matrix be positive definite, meaning the scalar value x^*Ax is strictly positive for all positive column vectors, x.

Storing only the nonzero values of a matrix can save sigificant memory space. Let N denote the number of rows and columns in the matrix, A, and M the number of nonzero values. The simplest way to represent a sparse matrix is to store the nonzero values, and the coordinates (i,j) of each nonzero. This is known as the coordinate (COO) format [89]. This format is common in many popular repositories, such as the Matrix Market format in the SuiteSparse matrix collection [90] or the FROSTT sparse tensor repository [91].

The COO format reduces the storage requirements from $O(N^2)$ to $3 \times O(M) = O(M)$. The COO format stores the same rows and columns multiple times, resulting in wasted space. The *row* array can be compressed into a row pointer (rp), or the columns can be compressed into a column pointer. The corresponding formats are known as compressed sparse row (CSR) and compressed sparse column (CSC), respectively [92].

The CSR and CSC formats can be compressed further if the matrix contains many rows or columns that contain only zeros. Only the indices of the rows or columns that contain nonzeros need to be stored for such matrices. In the case of CSR, the row pointer (`rp`) is compressed and a new array of compressed row indices is stored. This format is referred to as doubly-compressed sparse row (DSR) [93]. The choice of where to compress rows or or columns depends on which will yield a better compression.

Blocked sparse matrices can be efficiently represented by the blocked compressed formats. A matrix is block sparse when the nonzero values are clustered together in adjacent rows and columns. The matrix is divided into small dense blocks containing at least one nonzero element, and padded with zeros. The array `A_prime` consists of all such nonzero blocks. The `bcol` array stores the column of the upper left element of each nonzero block, and the `brow` auxiliary vector has one element per block row, indicating the first element index of the row in the original matrix `A`.

The compressed sparse block format (CSB) [94] shares the data locality benefits of the BSR and BSC formats, but without the need to store dense sub-blocks that are padded with zeros. The nonzero values are instead rearranged so that they can be traversed in a block order, such as the Z-morton sorting used in octrees [95]. The format requires five auxillary arrays, one for the block pointers (analogous to the row pointer in CSR), and two each for the block row and column indices, and the element row and column indices within each block.

The diagonal sparse matrix format (DIA) [96] is suitable for matrices with nonzero values near the main diagonal, such as the banded matrices in the previously described tridiagonal computations. The offsets from the main diagonal are stored in an auxiliary array. The ELLPACK format (ELL) [97] uses a 2-dimensional matrix with the maximum number of nonzero elements per row, and rows with fewer nonzero elements are padded with zeros. An auxiliary column matrix stores the column indices for the nonzeros. When most rows have a similar number of nonzero values, the ELL format is more efficient because of a fixed number of iterations and lack of indirect memory accesses. An example illustrating several of these matrix formats is given in Figure 2.7.

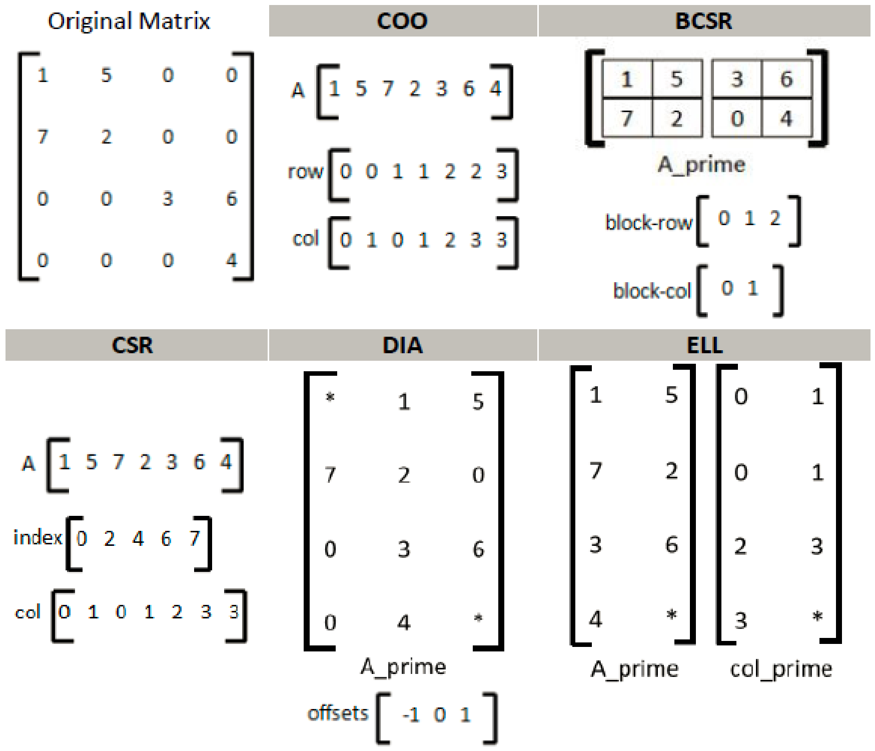
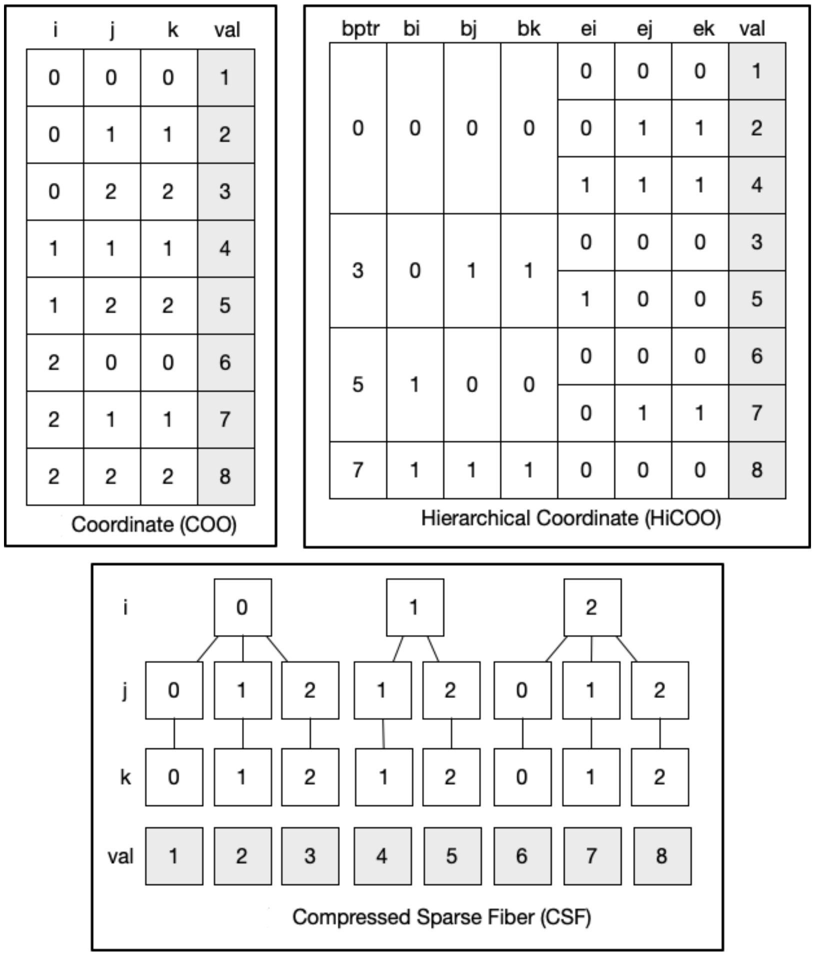


Figure 2.7: Sparse Matrix Formats

Tensor storage can be optimized by considering the data sparsity structure. The coordinate (COO) format can be generalized to sparse tensors by storing the coordinates for each mode. The compressed sparse fiber (CSF) format is a generalization of the CSR and CSC formats for matrices [98]. The HiCOO format is a generalization of the CSB format applied to tensors [33]. CSF is a mode-specific format, meaning that the resulting data structure is different depending on the mode compression order. The COO and HiCOO formats are mode-generic, so the nonzero values can be accessed in any order. Sparse tensor formats are summarized in Figure 2.8.



i	j	k	val
0	0	0	1
0	1	1	2
0	2	2	3
1	1	1	4
1	2	2	5
2	0	0	6
2	1	1	7
2	2	2	8

Coordinate (COO)

bptr	bi	bj	bk	ei	ej	ek	val
0	0	0	0	0	0	0	1
				0	1	1	2
				1	1	1	4
3	0	1	1	0	0	0	3
				1	0	0	5
5	1	0	0	0	0	0	6
				0	1	1	7
7	1	1	1	0	0	0	8

Hierarchical Coordinate (HiCOO)

Figure 2.8: Sparse Tensor Formats

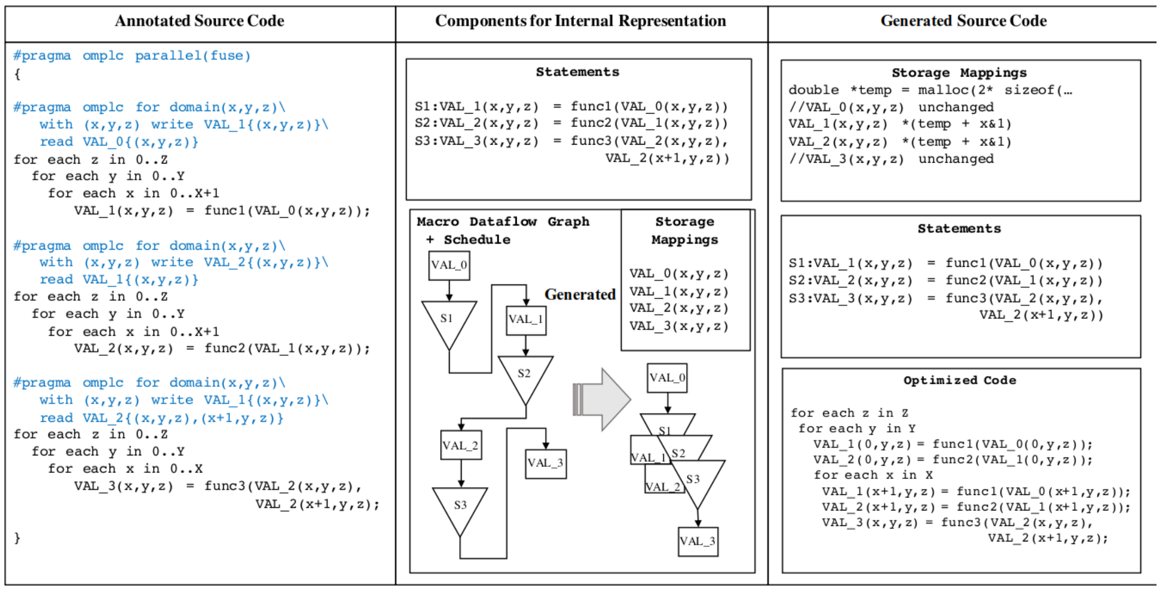
CHAPTER 3

POLYHEDRAL+DATAFLOW GRAPH INTERMEDIATE REPRESENTATION

This chapter presents an intermediate representation for loop chain schedules, and data mappings, a methodology for minimizing temporary storage requirements, and a cost model for comparing different schedules and mappings.

The approach provides a visual interface to aid the performance expert in guiding polyhedral code transformations paired with storage mapping optimizations. In this work we explore the concept of a polyhedral+dataflow graph (PDFG). Based on macro dataflow graphs, PDFGs express dataflow at a high-level using sets of statements, include information about the data being passed between nodes, and use layout to express the execution schedule. This approach is unique in that it includes fine-grained information about memory interactions, while the graph itself remains coarse-grained. Cost models have been used to compare the anticipated performance of macro dataflow graphs consistently since their inception. Given that the goal of many of these graphs is to identify parallelism opportunities, most of the cost models focus on execution cost of computation nodes [60], and the communication costs associated with the adjacent edges.

The novel contributions described in this chapter include (1) a procedure to generate PDFGs given annotated source code, (2) a set of scheduling and data



```
#pragma omplc parallel(fuse)
{

#pragma omplc for domain(x,y,z)\
   with (x,y,z) write VAL_1{(x,y,z)}\
   read VAL_0{(x,y,z)}
for each z in 0..Z
  for each y in 0..Y
    for each x in 0..X+1
       VAL_1(x,y,z) = func1(VAL_0(x,y,z));

#pragma omplc for domain(x,y,z)\
   with (x,y,z) write VAL_2{(x,y,z)}\
   read VAL_1{(x,y,z)}
for each z in 0..Z
  for each y in 0..Y
    for each x in 0..X+1
       VAL_2(x,y,z) = func2(VAL_1(x,y,z));

#pragma omplc for domain(x,y,z)\
   with (x,y,z) write VAL_1{(x,y,z)}\
   read VAL_2{(x,y,z),(x+1,y,z)}
for each z in 0..Z
  for each y in 0..Y
    for each x in 0..X
       VAL_3(x,y,z) = func3(VAL_2(x,y,z),
                            VAL_2(x+1,y,z);

}
```

Statements

```
S1:VAL_1(x,y,z)  = func1(VAL_0(x,y,z))
S2:VAL_2(x,y,z)  = func2(VAL_1(x,y,z))
S3:VAL_3(x,y,z)  = func3(VAL_2(x,y,z),
                         VAL_2(x+1,y,z))
```

Storage Mappings

```
VAL_0(x,y,z)
VAL_1(x,y,z)
VAL_2(x,y,z)
VAL_3(x,y,z)
```

Storage Mappings

```
double *temp = malloc(2* sizeof(…
//VAL_0(x,y,z) unchanged
VAL_1(x,y,z) *(temp + x&1)
VAL_2(x,y,z) *(temp + x&1)
//VAL_3(x,y,z) unchanged
```

Statements

```
S1:VAL_1(x,y,z)  = func1(VAL_0(x,y,z))
S2:VAL_2(x,y,z)  = func2(VAL_1(x,y,z))
S3:VAL_3(x,y,z)  = func3(VAL_2(x,y,z),
                         VAL_2(x+1,y,z))
```

Optimized Code

```
for each z in Z
 for each y in Y
   VAL_1(0,y,z) = func1(VAL_0(0,y,z));
   VAL_2(0,y,z) = func2(VAL_1(0,y,z));
   for each x in X
    VAL_1(x+1,y,z) = func1(VAL_0(x+1,y,z));
    VAL_2(x+1,y,z) = func2(VAL_1(x+1,y,z));
    VAL_3(x,y,z) = func3(VAL_2(x,y,z),
                         VAL_2(x+1,y,z);
```

Figure 3.1: Overview of the three code generation phases using loop chain pragmas and the polyhedral+dataflow graph method and associated cost model.

transformations for PDFGs, (3) a systematic approach to minimizing temporary storage requirements within graph nodes after fusion, (4) an approach for reducing storage allocations in the entire PDFG using liveness analysis, (5) a high-level cost model useful for comparing different graphs and execution schedules, and (6) a comparison of two overlapped tiling approaches.

3.1 Polyhedral+Dataflow Graphs

A PDFG is a visual representation of a computation highlighting data dependences. It can be considered a type of macro-dataflow graph [60]. Traditional dataflow graphs represent data dependences at a fine-grained level, typically per statement or machine instruction. PDFGs differ from existing dataflow graphs in three primary ways:

1. all iterations of a given loop nest are grouped into a single macro node,
2. data is explicitly represented as a node or set of nodes, and
3. the execution schedule is expressed as part of the graph layout.

This section describes the individual components of a PDFG, the expression of the execution schedule using graph layout, and a cost model to compare the potential performance of different graph variants. PDFGs form an intermediate representation that expresses both the execution schedule and dataflow requirements of an application kernel.

This chapter presents the graph IR, methods to transform them by manipulating the underlying polyhedral model, techniques to perform storage reductions, and code generation to produce optimized code. Optimization plans can be visualized, displaying intermediate steps, and the impact of transformations are clearly visible.

3.1.1 Graph Components

A polyhedral dataflow graph (PDFG) represents both the execution schedule and the dataflow requirements of a computation. A PDFG is defined as $G = (V, E)$, where $V = (S, D, T)$ and E the directed edges. S is the set of statement nodes, D the data nodes, and T the transformation nodes. The source and destination node types incident to an edge determine the operation being represented. For example, an edge from a data node to a statement node indicates reading data, and an edge from a transformation node to a statement node indicates that the iteration space will be transformed by applying the corresponding relation. The edges indicate the flow of data between statement nodes, and therefore the coarse-grained execution schedule.

Statement nodes, inverted triangles in the graph, represent ordered sets of statements. They encapsulate the iteration domain, statements within the block, location within the global schedule as an iteration vector or tuple, and the data mapping, referencing the data spaces that are read and written during statement execution. The iteration domain of a statement node is represented using the polyhedral model and the location within the global schedule is maintained using a scattering function. This function determines the fine-grained execution schedule. Each node, s in S, corresponds to a basic lock or loop nest in the code.

The data nodes, depicted as rectangles, abstract storage spaces and consist of the type, range of values, the domain of indices that access it, and the size. The latter can be inferred from the domain of the statement node that writes the data. The space described in the graph corresponds to local space requirements and not actual memory allocations. The memory allocation and associated mapping are created during code generation. Each node, d in D, represents a data space in the program that will be mapped to memory by a storage mapping function.

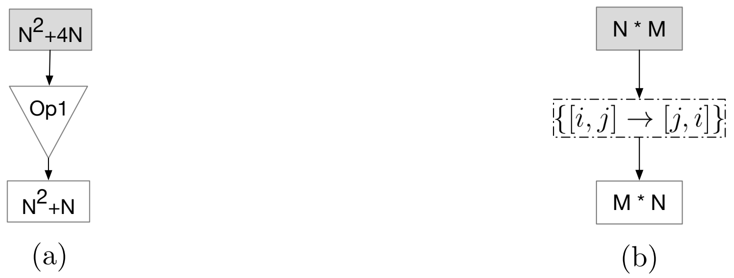


Figure 3.2: Summary of graph components, including edges, data, statement, and transformation nodes.

The graph components are summarized in Figure 3.2. The node labeled $N^2 + 4N$ represents a data space with that cardinality. There are two classes of data nodes, persistent and temporary. Persistent data are accessed outside of the function represented by the graph, either as inputs or outputs, and therefore have a fixed storage mapping. Temporary values are allocated and accessed only within the scope of the graph or loop chain. Persistent data nodes are shaded gray, and temporary or local data retain a white background. The $N^2 + N$ space is temporary.

The statement node, *Op1* in Figure 3.2(a) represents a code block that performs some operation. The incoming edge indicates that the block reads the persistent data, computes the results, and writes them to temporary storage. The contents of statement nodes are retrieved from loop bodies.

Arbitrary transformation functions can be introduced with transformation nodes, denoted in the graph as dashed boxes. Iteration space transformations are specified beginning with a T and data transformations with an R by convention. The transformations are expressed as relations using Presburger arithmetic. The node in Figure 3.2(b) transposes the persistent data space, *N*M* to one of *M*N*. Note that if that transformation were placed between two statement nodes, the result would be a loop interchange of the corresponding iteration spaces.

This polyhedral+dataflow representation support sparse data structures as well. Sparse data structures are important in many applications, including scientific computing, graph analysis (e.g., data science or social media networks), and machine learning. The code for these applications often results in irregular memory access patterns caused by multiple levels of indirection, for example with index arrays such as `A[col[i]]`). These patterns can result in poor performance due to reduced data locality. Data-dependent loop bounds and indirect memory accesses rely on data that are unknown until run-time, making static analysis difficult.

Uninterpreted functions are realized as explicit functions that satisfy the associated constraints at run time. The computational kernel that requires the explicit function is known as an executor, and the kernel that generates the data is an inspector. Inspectors are often hand-written, but both executors and inspectors can be generated by optimizing compilers, including the polyhedral model. The performance of many applications, including inspectors, can be significantly improved by decreasing temporary storage and thereby reducing memory traffic.

3.1.2 MiniFluxDiv Benchmark

The MiniFluxDiv benchmark [99] is used as an application exemplar to demonstrate this approach. The benchmark was chosen because it captures some of the complexity of full-scale simulation-based applications. MiniFluxDiv has been annotated using loop chain pragmas [14]. The loop chain annotations provide the information required to achieve a separation of concerns among statements, schedule, and storage mappings. MiniFluxDiv is modeled after finite difference applications such as those written with the Chombo framework [100]. The benchmark focuses on the shared-memory portion of a single time step in an iterative solve. The input is a 3D, immutable data structure padded with a layer of ghost cells (2 deep). The domain is broken into a set of independent subdomains called boxes. Boxes are decomposed into cells; 16^3 cells is a typical box size, but larger box sizes are desirable to reduce the space required for ghost cells. We explore box sizes of 16^3 cells and 128^3 cells in this work. Each cell represents a vector of five components, including density (ρ), energy (e), and the velocity in each direction (u, v, w).

The original implementation is a series of parallel loops. There are three loops for each dimension of the problem. The first loop performs a partial flux. This calculation results in face values, meaning that when the partial flux is calculated in the x-direction, a value is required for each border between cells. The second loop completes the flux calculation using data from the corresponding velocity components in each direction to produce partial fluxes. These steps are referred to as `Fx1` and `Fx2`. The fluxes in the y- and z- directions are referred to as `Fy1`, `Fy`, `Fz1`, and `Fz2`, respectively. The third loop calculates the differences between flux values and saves a cell-centered value at each point.

Each of the operations is applied to all five components. A naive implementation results in a series of 45 parallel loop nests. The performance baseline is hand-optimized to reduce the number of loops.

Loop Chains

A loop chain is a series of loop nests that perform operations on shared data [51]. The loop chain abstraction captures this pattern and promotes decoupling of the execution schedule from the algorithmic primary expression. The abstraction can be implemented in a variety of ways: domain specific languages, libraries, or code annotations. A loop chain pragma language has been developed and a restricted version of it is used in this chapter [14].

The first column in Figure 3.1 demonstrates how the pragmas are added. The outermost pragma, `omplc parallel(fuse)`, indicates the start of a loop chain and the schedule that should be applied, i.e., fuse. Each loop nest within the chain is labeled with a pragma, indicating its domain. The pragma `domain(0:X+1,0:Y,0:Z)`, for example, indicates that iterator x has domain 0 through $X+1$ (inclusive). Data read and write patterns are specified in the pragma following the *with* clause.

A loop chain compiler has been implemented by Bertolacci et. al [14] that uses the pragma specifications to apply a variety of transformations to the original application code, including shifts/skews, fusion, tiling, and wavefront. In the existing tool, the data access patterns help the compiler to ensure the legality of transformations, but is not used to optimize data accesses or temporary storage.

3.1.3 Execution Schedule

The edges of the graph indicate a partial execution schedule based on data dependences. The graph layout expresses the execution schedule. Graphs are executed from left to right, and top to bottom. Statements within the nodes are executed over the domain in lexicographical order. An exception is made after fusion operations. In this case any shifting will be automatically applied to ensure legal execution.

The original `MiniFluxDiv` schedule over a 2D domain is represented by the PDFG provided in Figure 3.3. The graph is organized into four columns, one for each component in the 2D space. The persistent data nodes in the top row labeled ρ_0, u_0, etc. represent the initial input data for each box. Similarly, the persistent data nodes along the bottom, e.g., ρ_1, u_1, etc., contain the resulting output data. The input nodes are of size $N^2 + 4N$, and the output nodes N^2, the difference is due to ghost cells.

The first face-centered flux loop is represented by the statement nodes labeled `Fx1`. Note that the velocity component of the `Fx1` statement node, u, is read by the `Fx2` statement nodes for all components. The same is true for `Fy1`(v). This dependence pattern is common in CFD applications, and is necessary to obtain realistic performance results.

3.1.4 Cost Model

A cost model is derived using the data nodes in PDFGs. Two primary metrics are calculated: the total amount of data read (S_R), and the maximum number of streams

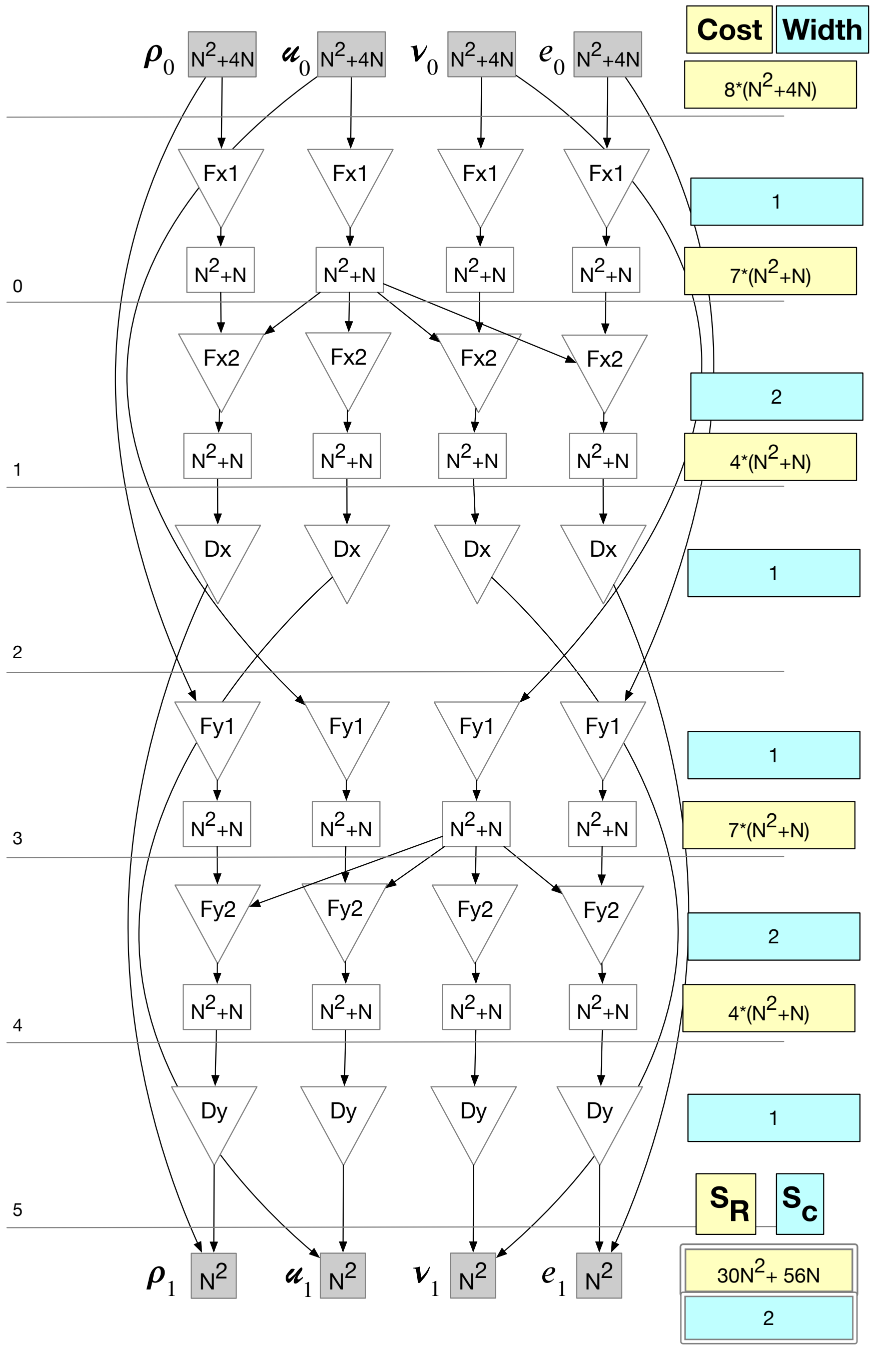


Figure 3.3: A graph representation of the series of loops implementation of the `MiniFluxdiv` benchmark. This schedule uses static single assignment for all values produced within the represented computation.

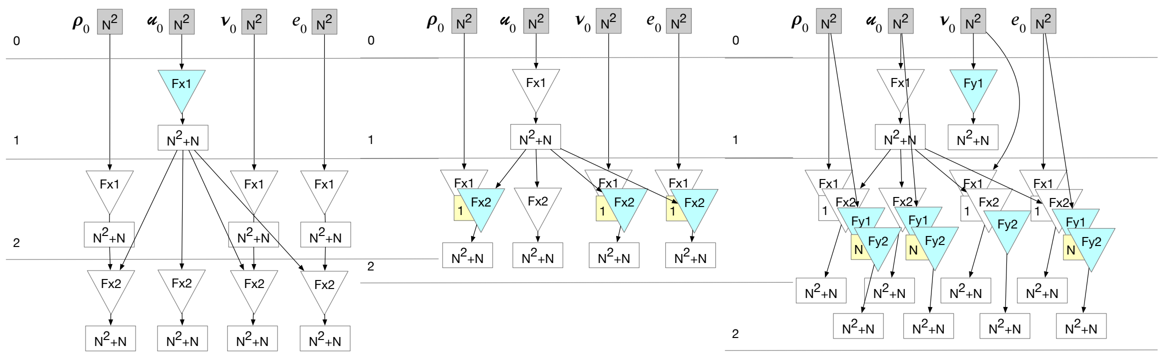

a) Reschedule: The cyan node has been rescheduled in preparation for rows 2 and 3 to be fused.
b) Producer-Consumer Fusion: The cyan nodes have been fused with the Fx1 nodes. This enables the yellow value sets' memory footprint to be reduced.
c) Read Reduction Fusion: The cyan nodes have been fused with the previously fused nodes. While the nodes are executed in a fused iteration space they each produce distinct value sets.

Figure 3.4: The set of operations that are defined for dataflow graphs.

being accessed simultaneously (S_c).

The total amount of data read for each data space is the number of outgoing edges multiplied by the size of the data space. The total for the entire graph is the sum of those values. For example, in Figure 3.3 the total amount of data read in each row is summed in the yellow boxes at the right. The total is on the yellow box at the bottom right labeled S_R.

The maximum number of streams being accessed simultaneously (S_c) determines whether or not the prefetching capabilities of the target architecture have been exceeded. This metric is calculated by taking the maximum incoming degree among all of the statement sets. The maximum number of streams being simultaneously accessed can be improved in a case that there are wide multi-dimensional stencils in the statement node. This pattern type needs to be detected and additional edges included if the prefetch distance for the target machine is exceeded.

The number of simultaneously read data streams, or width, is given in the blue boxes. The total number of streams read in this case is $S_c = 2$. The graph operations described in the following section are intended to reduce S_R, and keep S_c below a threshold to avoid exceeding the capabilities of the prefetcher.

3.2 Graph Operations

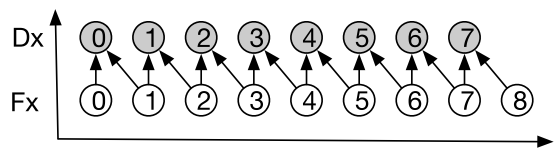
There are three operations defined for the PDFGs, each corresponds to a transformation in the generated code. Figure 3.4 provides visualizations to describe the operations, as detailed in the following subsections. These include the `reschedule` operation, and two types of `fuse` operations, producer-consumer and read reduction. Tiling transformations are considered separately from the reschedule and fuse operations. A tiling approach is defined and applied to the entire graph. Overlapped tiling as we implemented it is described in this section.

3.2.1 `reschedule` Operation

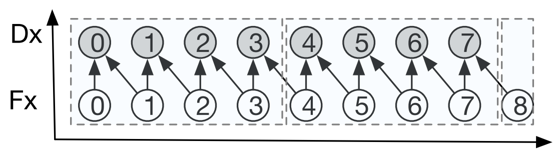
The `reschedule` operation moves a node from one row to another within the graph layout, effectively changing the execution schedule. For example, Figure 3.4(a) demonstrates relocating the velocity component (u) of the `Fx1` operation so that it will be executed before the other components. Rescheduling is provided as a convenience operation to enable subsequent optimizations, or to allow easier interpretation of the graph for code generation.

3.2.2 `fuse` Operations

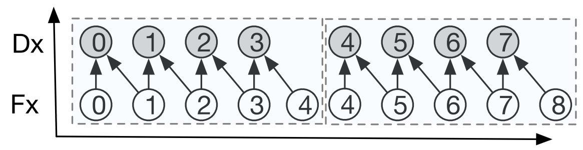
Fusing nodes in the graph directly corresponds to loop fusion. Producer-consumer fusion results in a single, more complex statement node. The benefit is a temporary data storage requirement reduction.



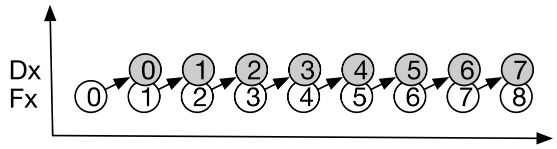
(a) Original execution schedule schedule.



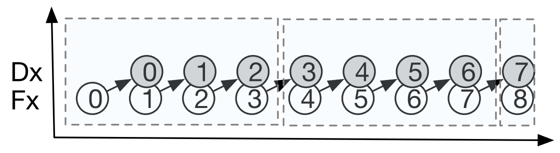
(b) Classic Tiling.



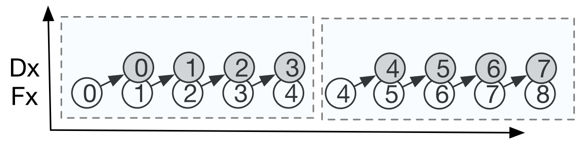
(c) Overlapped Tiling as applied in Halide and PolyMage.



(d) Shifted and fused schedule.



(e) Classic tiling applied to the shifted and fused schedule.



(f) Overlapped tiling applied to the shifted and fused schedule.

Figure 3.5: Illustrations of the transformations that create the two overlapped tiling variants.

A read reduction fusion occurs when two statement nodes read data from the same data node. Each reader still produces its own value space, so there is no storage reduction. However, it provides an opportunity to reduce the number of times the same data are read.

The compiler transformations for the two fusion types are the same. However, the differences affect the cost model. The producer-consumer fusion of `Fx1` and `Fx2` is given in Figure 3.4(b). The subsequent read reduction fusion of the various operations is given in Figure 3.4(c). Fusion of statement nodes is indicated by the overlapping triangles.

3.2.3 Overlapped Tiling

The operations presented previously focus on the execution schedule among nodes of the graph. Global operations, like tiling, are applied to the graph as a whole. Tiling transformations divide a problem domain into smaller subdomains called tiles. In stencil-based applications, this leads to improved temporal locality and decreased data movement. This approach supports two types of overlapped tiling, and we provide a comparison.

In classical tiling, each iteration in the original space is executed by exactly one tile. This translates to each statement node in the graph being tiled separately. In overlapped tiling, an iteration can be executed in multiple tiles. This results in redundant computation overhead, but improved parallelism [99, 101].

Consider the two statement nodes, `Fx2` and `Dx`, as introduced in Figure 3.3. Each iteration of `Dx` reads two values produced by `Fx2`. This is illustrated in Figure 3.5(a).The arrows indicate dataflow. Classical tiling with a tile size of four results in three tiles, Figure 3.11(b). The dependences between tiles require a barrier to be placed after the `Fx2` statements finish execution and before `Dx` can begin.

Overlapped tiling involves redundant computation within tiles to alleviate dependences. Figure 3.5(c) demonstrates overlapped tiling as it is applied in Halide [4], hierarchical overlapped tiling [101], and others. The tile size of four is only applied to the final statement set (Dx). The previous statement sets in the execution schedule are expanded to satisfy the dependences. In this case, the `Fx2` statement set is expanded by one in the positive direction for each tile, and the fourth iteration is executed by two tiles.

A second approach to overlapped tiling fuses producer/consumer loop nests before tiling, Figure 3.11(d). In this example, the loop nest must be shifted for legal fusion. Classic tiling after fusion forces serial execution, Figure 3.11(f). The domains of the previous statement sets expanded to create overlapped tiles. This approach is illustrated in Figure 3.5(f).

Each overlapped tiling approach has distinct advantages. The first preserves the parallelism available in the inner loop, and enables vectorization. The second reduces the temporary storage required per tile. In this case, the first approach requires space for as many iterations as are in the tile. In the second, only two scalars are required. The preferred approach depends on the application and the target hardware. According to the performance results demonstrated in Figure 3.11, sacrificing vectorization for reduced memory traffic is advantageous to this benchmark.

3.2.4 Mapping Data to Memory

Each data space representing temporary data expresses its space requirements in its label. A map is generated differently depending on whether the data node is standalone, or if it has been pulled into a statement node through fusion. Standalone nodes use a one-to-one mapping between the iterator of the writing statement node and memory locations. Each of these maps are relative, meaning that the actual address to the space in memory is a parameter.

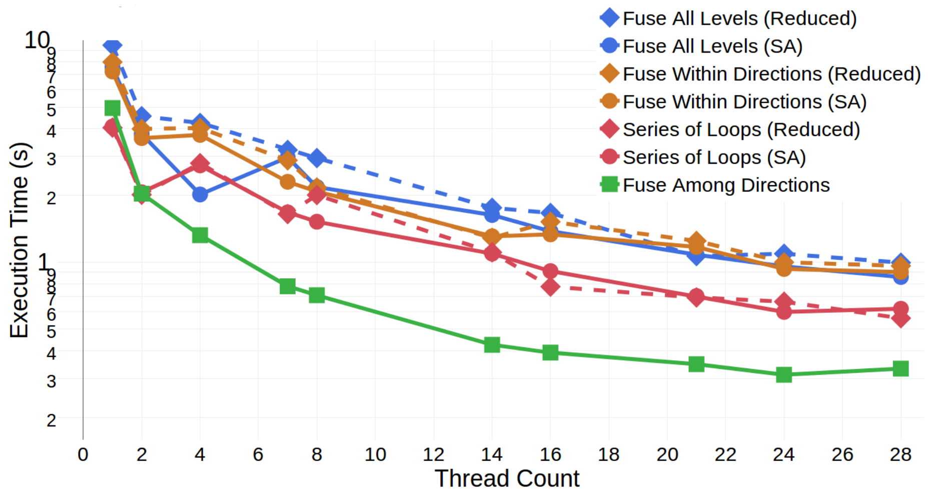
The map for a node subject to producer-consumer fusion is calculated from the data access patterns defined in the loop chain pragmas, along with the reuse distance in the transformed schedule. The distance is 1 in Figure 3.4(b), and only one value is read, therefore, the required space can reduced to a single scalar value. Fusing an operation with a stencil reading pattern will result in greater space requirements. For instance, fusing a `Dx` operation from Figure 3.3 produces a reuse distance of only

1, but two values need to be maintained. The data dependence for a stencil in the y-direction requires even more space to satisfy. Fusing a `Dy` node with a `Fy1` would require saving two values for each operation. The reuse distance is the domain length in the x-direction (N). In this case, the dependences can be satisfied with a buffer of size $2N$.

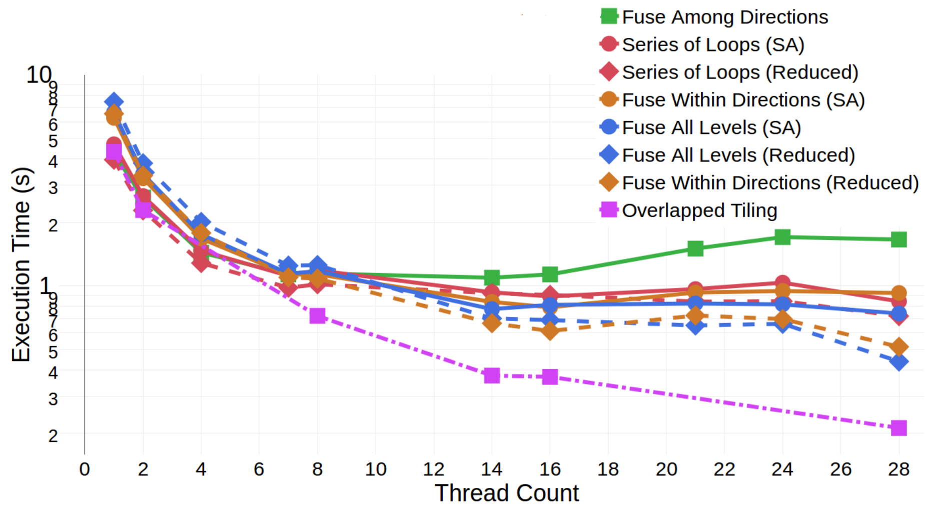
The address is provided by static liveness analysis applied to the graph as a whole. The liveness analysis proceeds by processing the graph in reverse execution order. A table is maintained with a list of spaces, the corresponding pointer ID, capacity, and a boolean indicating whether the location is active. During graph traversal a data node is assigned to an existing space that is of equal or greater capacity and marked as inactive. An existing, smaller space is expanded if no inactive space can accommodate the space required by the node . If no inactive spaces exist, a new space is added to the table, the node is assigned to it, and the space is marked as active. When the node that writes to the data node is visited, the space is marked as inactive.

3.3 Experimental Evaluation

This section details the experiments performed on the MiniFluxDiv benchmark and the larger AMR-Godunov application. PDFGs were used to guide a series of optimizations on the MiniFluxDiv. Our performance measurements demonstrate that scheduling optimizations are less effective without the corresponding reduction in temporary data, the overlapped tiling variant focusing on memory traffic reduction outperforms the vectorized version for this benchmark, and our performance is competitive with the performance achieved using Halide's and PolyMage's autotuning capabilities.



(a) Results with box size of 16 cells.



(b) Results with box size of 128 cells.

Figure 3.6: Performance of the `MiniFluxdiv` benchmark on 28-Core Intel Xeon E5-2680 CPU for both (a) small (16^3) and (b) large (128^3) boxes. The *y*-axis is in log scale.

The performance of a larger example application, AMR-Godunov, was explored using PDFGs. The application was manually optimized and a performance improvement of 17% was observed.

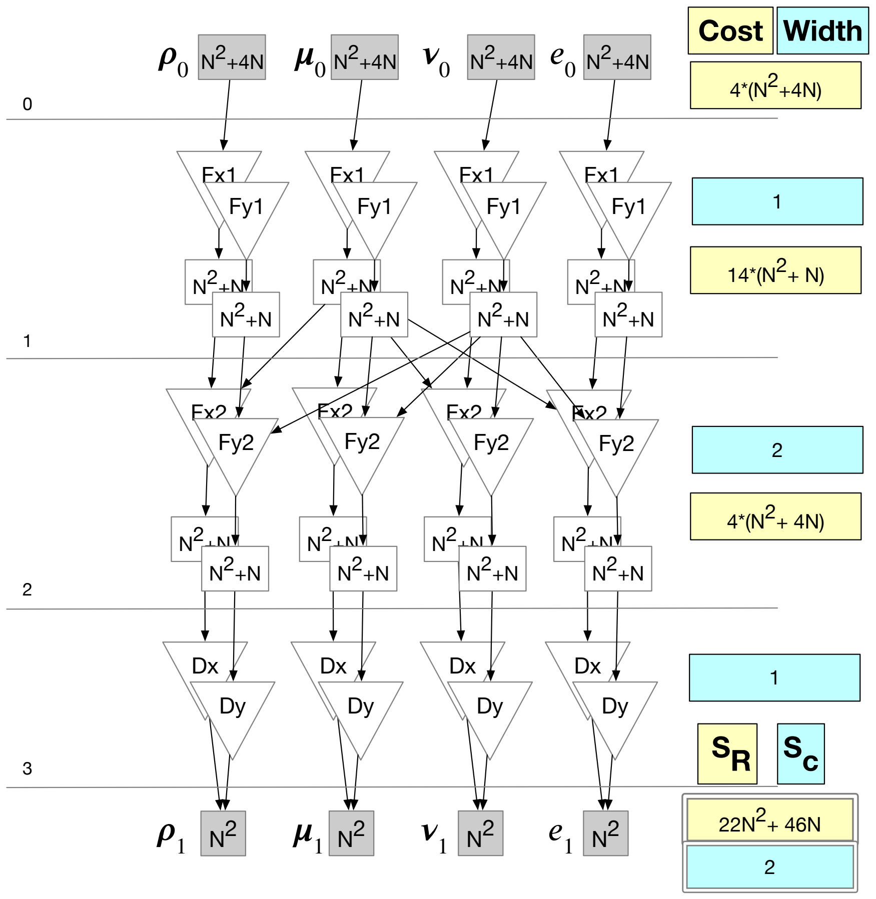


Figure 3.7: Graph for fuse among directions variant (green line in Figure 3.6).

3.3.1 Experimental Setup

The benchmark was optimized using several different schedules, each schedule was applied to a small box size of 16^3 and a large size of 128^3. The total number of cells per experimental run is 58, 720, and 256 cells, while the number of boxes is calculated accordingly (14,336 and 28, respectively). The scalability of each variant is explored by varying the thread count from 1 to 28, i.e., the number of cores on the target machine, with per thread parallelism over the boxes. Each experiment was run five times and the mean execution time is presented here.

All MiniFluxDiv experiments were conducted on the R2 cluster at Boise State

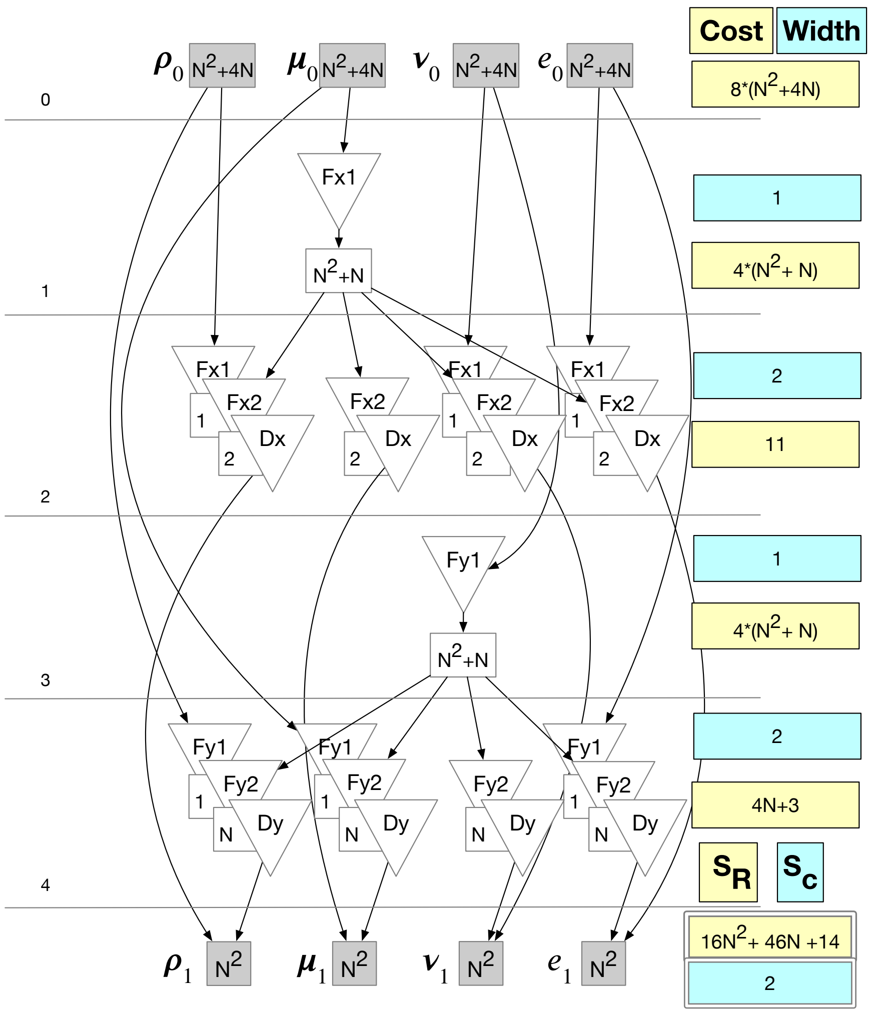


Figure 3.8: Graph for fuse within directions variant (orange lines in Figure 3.6).

University. Each node of R2 is a dual socket, Intel Xeon E5-2680 v4 CPU at 2.40 GHz clock frequency with 28 cores (14 per socket). The cores include a 32KB L1, 256KB L2, and 35840K L3 caches. The system contains 192GB of RAM split over 2 NUMA domains. GCC g++ version 6.1.0 was used to compile all the benchmarks, with optimization level `-O3` used by the compiler.

The experiments for AMR-Godunov were performed on Atlantis at Colorado State University. Atlantis is a 20-core machine composed of two 10-core Intel Ivy Bridge

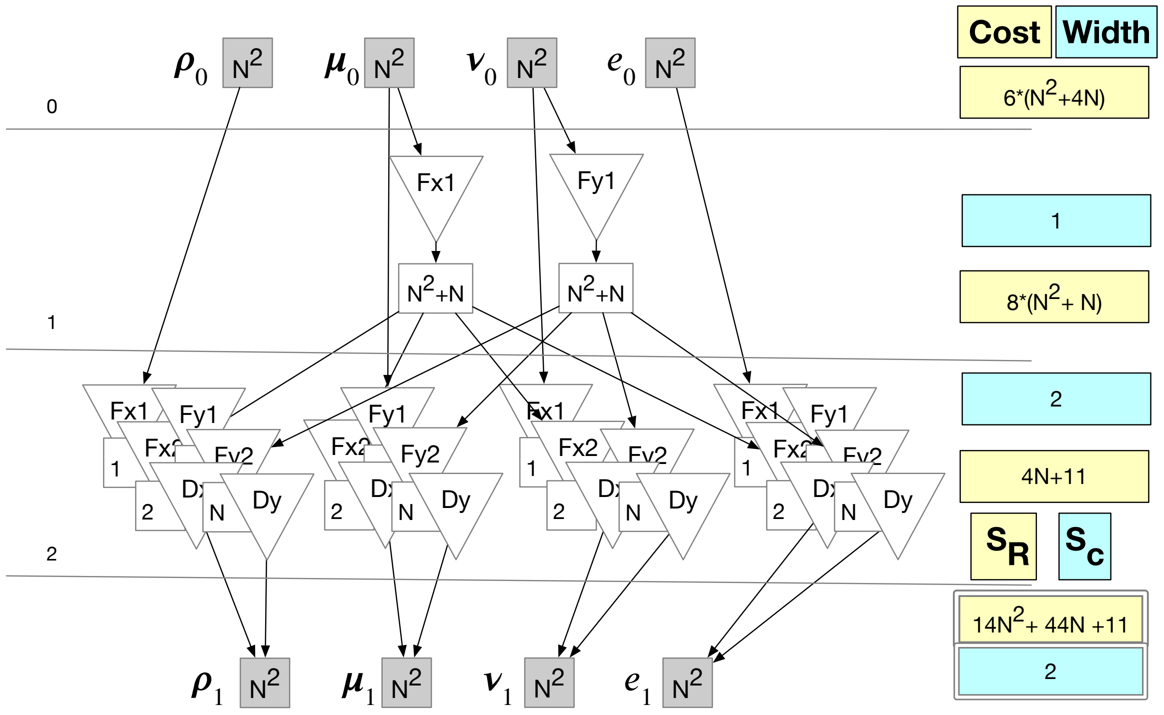


Figure 3.9: Graph for fuse all levels variant (blue lines in Figure 3.6).

E5-2670v2 chips running at a clock rate of 2.50 GHz. The system is configured with 128 GB of DDR3 RAM in a quad-channel configuration with a clock rate of 1600 MHz, giving 51.2 GB/s of bandwidth per socket or an aggregate system bandwidth of 102.4 GB/s. Each core has a 32 KB of level 1 instruction cache, 32 KB of level 1 data cache, and 256 KB level 2 cache. All cores on a socket share 25 MB of level 3 cache.

3.3.2 Benchmark Variants

Experiments were conducted using five variants of a 3D implementation of the benchmark. Four of the variants did not use tiling: 1) series of loops, 2) fuse among directions, 3) fuse all levels, and 4) fuse within directions. Series of loops is the baseline variant. This is the original implementation and is used as the performance baseline.

Variants two through four were created using PDFGs. An overlapped tiling variant was implemented using schedule 3 (fuse all levels) as the execution schedule within the tiles. Two versions of the first four variants were created. A single assignment (SA) version with no storage optimizations and, when possible, a version with storage optimizations (reduced).

The diagrams in this chapter present a 2D version of MiniFluxDiv. This was done to save space. In the diagrams there are four components and a series of 24 loop nests. All experimentation was performed with the full 3D version. The 3D version has five components and a series of 36 loop nests.

Series of loops. The baseline implementation is series of loops with storage optimizations. This is the original implementation of the benchmark mirroring the implementation in the Chombo framework. Figure 3.3 displays the original schedule. This schedule performs well for small box sizes. Figure 3.6 shows this variant in red. The solid line is without temporary storage optimizations and the dashed line is with them. The parallelization is straight-forward and can be done within boxes or over boxes, using OpenMP parallel for pragmas on each loop nest or on the outer loop nest over boxes. On our target machine the parallelization over boxes performed better and is used in all results unless labeled otherwise.

Fuse among directions. This variant is shown in Figure 3.7. Read reduction fusion is performed on the Flux operations (Fx1,Fy1) and the fusion of the Diff (Dx,Dy) operations results in better data locality for writing to the output buffers. Only the SA version was implemented, because there are no opportunities for storage reduction. Figure 3.6 displays this variant in green. This is the only schedule that improves on the baseline code for small boxes. However, the performance is poor for the large boxes.

Fuse all levels. This schedule is displayed in Figure 3.9. This schedule maximizes both producer-consumer and read reduction fusion. Both versions of this schedule perform well for large boxes, with the data reduced version being the most performant.

Fuse within Directions. The fuse within directions graph variant is given in Figure 3.8. This schedule maximizes the use of producer-consumer fusion. The Fx1 and Fy1 operations that are applied to velocity components cannot be included in the fusion. They are rescheduled before the fused row as to respect the data dependences. Fusing within directions is scalable, but does not outperform the series of loops for small boxes, or the fuse all rows schedule for large boxes.

Overlapped Tiling. Overlapped tiling was applied to the Fuse all levels schedule. The overlapped tiling variant performs the best for the large box case. This result is an improvement on our previous work. The improvement came from changing the intra-tile schedule. The use of PDFGs and code generation allowed for a larger set of intra-tile schedules to be attempted.

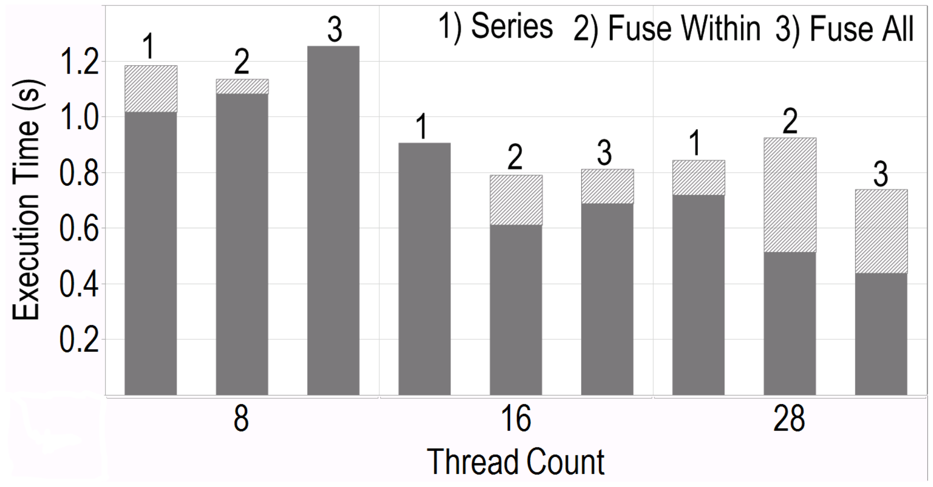


Figure 3.10: The execution time for schedules with storage mapping optimizations are significantly faster for most schedules. The original times are represented by the light gray bars, and the dark bars indicate the reduced times.

3.3.3 Temporary Storage Reductions

Figure 3.10 shows a subset of the schedules explored in this work. Each bar represents a variant with only scheduling changes, and the corresponding variant with both scheduling and data reduction optimizations. The benefits of the data reductions are most clearly seen for the large box sizes.

Figure 3.6 uses dashed and solid lines of the same color to display the impact of storage reduction. Each variant is shown twice in the same color. The solid line represents the variant without temporary storage reductions and the dashed line with those reductions. The impact of the storage reductions is most clearly seen at high thread counts and with the large box sizes.

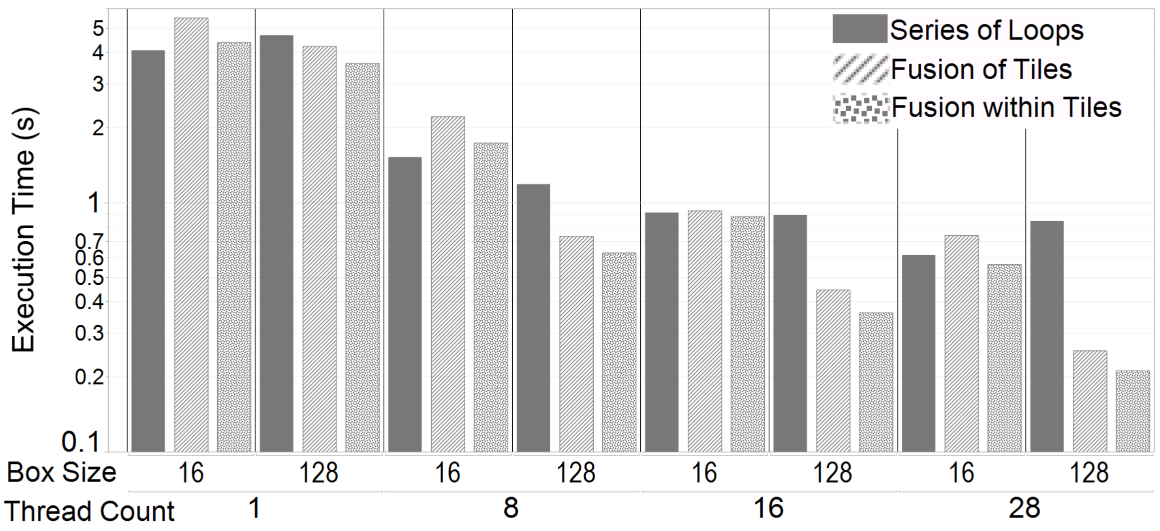


Figure 3.11: Overlapped tiling comparison of the two techniques applied to the `MiniFluxDiv` benchmark, including the original series of loops implementation as a reference. The x-axis is tiling method within box size and thread count. The y-axis is in log scale.

3.3.4 Overlapped Tiling Comparison

The data reduced variant of the fuse all levels schedule was selected to test the overlapped tiling implementation. This transformation produces the tiling as illustrated in Figure 3.5(f), or fusion within tiles. Tiling enables even further data reduction, as each thread only needs to allocate enough space for one tile. To compare with the overlapped tiling method given in Figure 3.5(c), the original series of loops schedule was tiled and then fused, referred to as fusion of tiles. The measurement results are compared with the baseline schedule and displayed in Figure 3.11. The fusion within tiles technique outperforms fusion over tiles for both small and large boxes on all four thread counts.

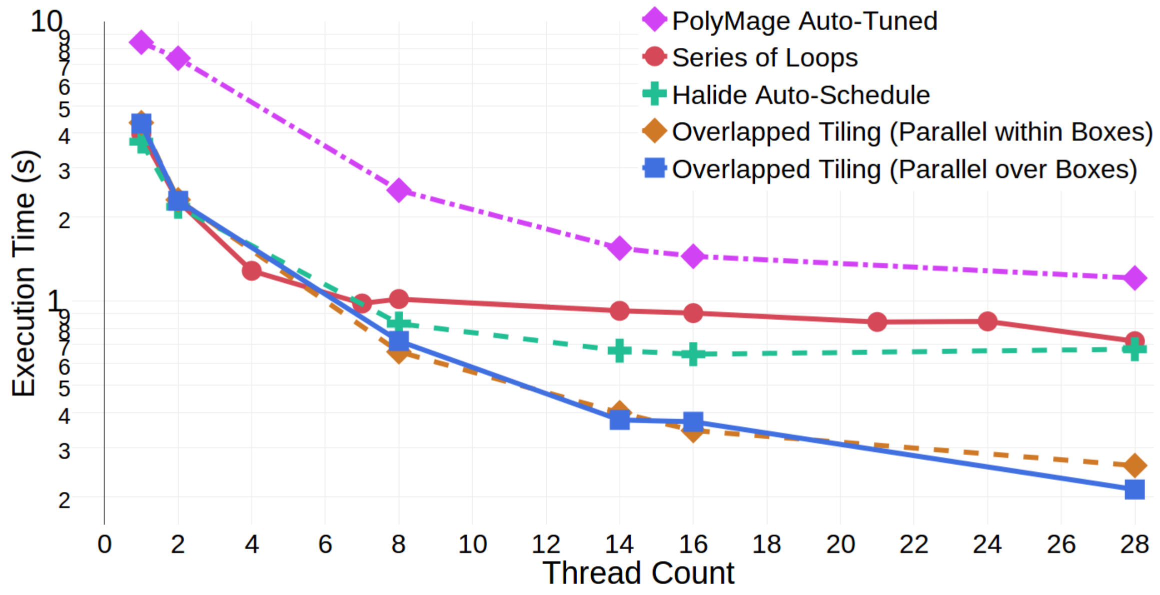


Figure 3.12: Series of loops, Overlapped tiling, Halide, and PolyMage with box size of 128 cells. The y-axis is in log scale.

3.3.5 Halide and PolyMage Comparisons

The `MiniFluxDiv` benchmark was implemented using the Halide [4] library and PolyMage [5]. The performance results show that the PDFG-guided schedules outperform the autotuned versions using Halide [56] and PolyMage (see Figure 3.12).

Results for the smaller box size (16^3) are omitted. The Halide and PolyMage implementations are limited to parallelization within the boxes. This limitation is not fundamental to the approach; it is implementation specific.

The overlapped tiling variant described here outperforms both the Halide and PolyMage variants. The results for the large boxes are limited to within boxes. In this case, we applied both parallelization over and within boxes for a fair comparison, each of those variants outperform Halide and PolyMage.

The primary difference between the two execution schedules is the iteration space fusion method. Our approach complicates vectorization, but reduces temporary storage requirements. Preserving straight-forward vectorization requires an increase in temporary storage use. Scaling out to 28 cores puts pressure on the memory subsystem and reducing that pressure takes precedence. This is not true for all applications, however, this is an important insight because MiniFluxDiv represents patterns commonly found in scientific applications.

3.3.6 AMR-Godunov Solver

AMR-Godunov [102] is an example AMR application published with the Chombo [100] software. It is an unsplit, second-order Godunov method. The application is written in a combination of C++ and Fortran. Each of the primary computational kernels is written in Fortran. PDFGs were used to organize a set of optimizations. The opti-

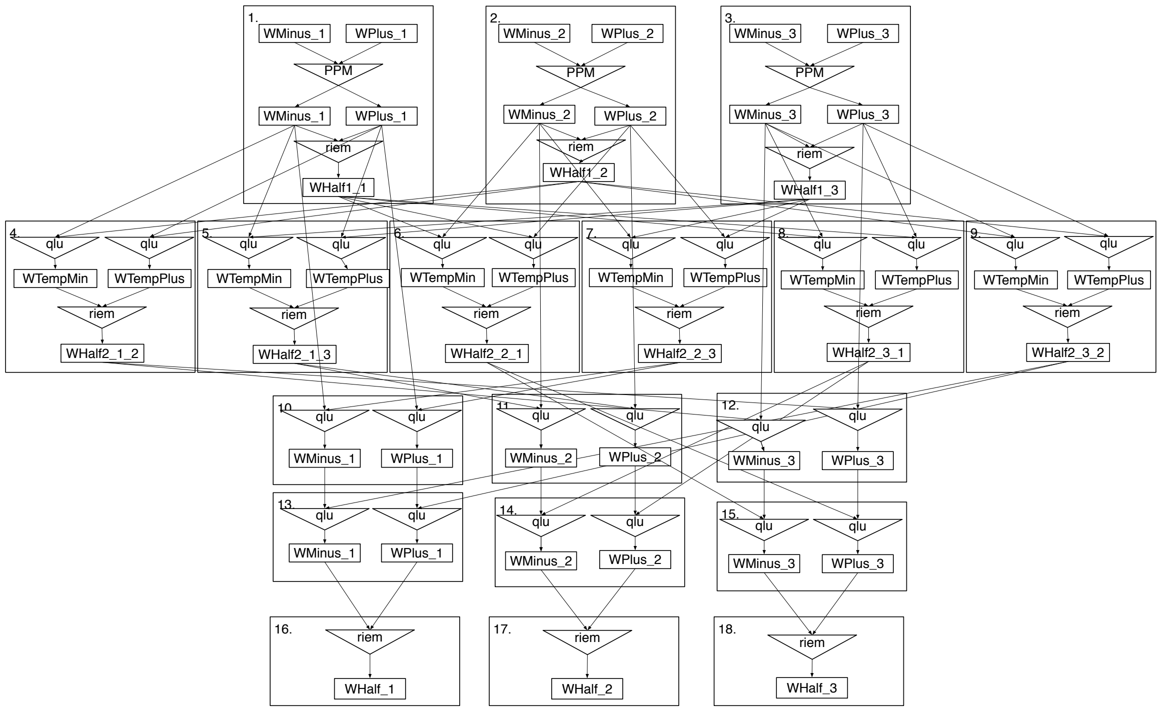


Figure 3.13: The PDFG for the original implementation of *ComputeWHalf*, a subroutine that is part of a single timestep of AMR-Godunov application.

mizations were applied by hand in the Fortran code. The final schedule reduced the size of the temporary space required by approximately 14KB. The overall execution time was reduced by 17%.

Figure 3.13 shows the PDFG for a subroutine that consumes approximately 80% of the execution time at each time step. Each time step involves communicating with ghost cells, and then processing each box independently. Optimizations were applied only within this subroutine. The problem domain is decomposed into independent subdomains called boxes. Each box contains a set of five component values for each 3D cell. In this example, the boxes are held at size 16^3.

The process for optimizing the code started at the bottom of the graph. Each of the *qlu* (quasi-linear-update) nodes were executed in pairs and were fused. This

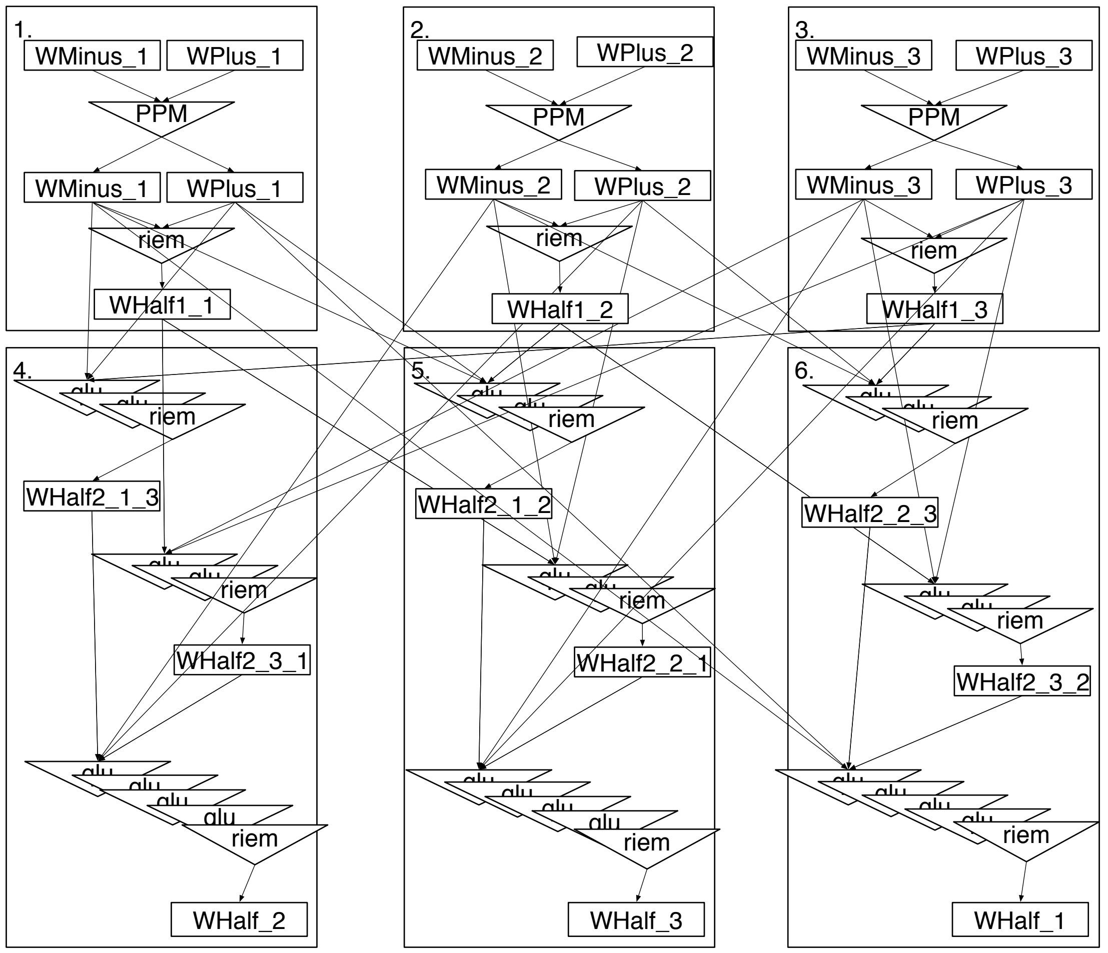


Figure 3.14: The PDFG for *ComputeWHalf* after optimization. The coding for the optimizations was performed by hand and was guided by manipulation of the PDFG.

created a simple producer-consumer pair between the fused *qlu* nodes and the following Riemann solve nodes which were subsequently fused. Fusion was accomplished by creating a new fusion-specific Fortran kernel. Each fusion was coded separately as the stencil dependences required shifting that was slightly different for each case. Figure 3.14 shows the final graph.

3.4 Summary

The polyhedral+dataflow graph intermediate representation presented in this chapter can be used to visually represent series of stencil computations (i.e., loop chains) that often occur in scientific applications. Transformations on PDFGs correlate to schedule changes including overlapped tiling. An algorithm for determining what temporary storage reductions can be done is provided. A cost model to compare schedules based on memory traffic is presented, and results show it enables comparing relative performance between variants. Experimental results on a CFD benchmark and AMR-Godunov solver show that performance obtained by some of the schedule variants can outperform the state of the art methods.

CHAPTER 4

POLYHEDRAL+DATAFLOW SPECIFICATION LANGUAGE

This chapter describes a specification language for the polyhedral+dataflow intermediate representation (PDFL) that can be written directly, or derived from existing source codes or representations. The language has been implemented as a an embedded domain specific language (eDSL) in C++. The polyhedral expressions are validated and simplified using the IEGenLib library from the Sparse Polyhedral Framework (SPF) [8]. Graph variants are produced by applying successive transformations to the graph. Arbitrary polyhedral transformations are supported by the introduction of transformation nodes. An overview of the process is given in Figure 4.1.

The intermediate representation detailed in Chapter 3 describes computations, execution schedules, and data mappings. The IR supports scheduling and dataflow-based code transformations, and includes a visual depiction that reflects the expected effect of transformations. The corresponding language specifies instructions to generate the IR and implements methods to perform the transformations.

This toolchain can be incorporated with other tools by implementing a frontend to create graph specifications. The user can interact with the tool via a command line compiler interface, a Python API, or an online version in the form of a Jupyter notebook. The contributions of this chapter include (1) a specification language to create the polyhedral+dataflow representation, (2) automatic allocation of temporary storage space and code generation for data mappings, and (3) example transformations on irregular applications.

4.1 Polyhedral+Dataflow Language

The polyhedral+dataflow language is designed to express regular (structured), or irregular (sparse) computations such as those commonly found in scientific or other numerical applications. Examples include stencil operations in partial differential equation (PDE) solvers, or sparse linear and multilinear algebra kernels. Each computation is represented by a space, bounded by a set of constraints to form a

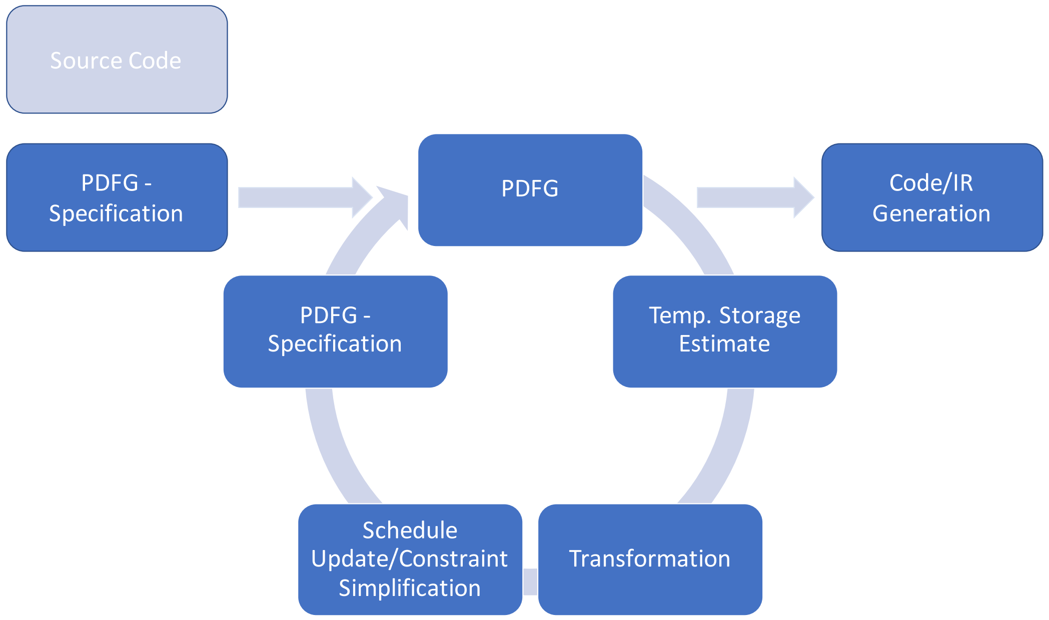


Figure 4.1: Flowchart of the overall process, beginning with the initial PDFG specification (PDFL), producing graph variants via the optimization process using IEGenLib, and the composition of dataflow graph code with the output from Omega+ into a final program.

polyhedron. Each space can describe either an iteration or data space. Iteration spaces are associated with statements that define the computations to be performed at each point. A simplified subset of the grammar is given in Figure 4.2. The terminals **ident**, **num**, **constraint**, and **stmt** are omitted for brevity, but are defined as identifiers, numbers, logical relations or assignments, and statements in the C programming language.

This eDSL consists of four primary constructs: spaces, functions, iterators, and computations.

1. Spaces represent points in space, either iteration spaces or data. Iteration spaces *are* collections of integers, describing the points of a polyhedron. Data spaces can have any primitive data type. *s2*(i,j,2)). Spaces can be scalar, consisting of a data or iteration single point.

2. *Iterators* are values that traverse a space. The space is bounded by constraints

$$\begin{aligned}
\text{expr\_list} &\rightarrow \text{expr “; ” expr} \\
\text{expr} &\rightarrow \text{set\_expr} \mid \\
&\quad\ \text{rel\_expr} \mid \\
&\quad\ \text{fxn\_expr} \mid \\
&\quad\ \textbf{ident} \mid \textbf{num} \mid \epsilon \\
\text{set\_expr} &\rightarrow \textbf{ident} \text{ “(” tuple\_list “)” “ = ”} \\
&\quad\ \text{“\{” constr\_list “\}” “ : ” “\{” stmt\_list “\}”} \\
\text{tuple\_list} &\rightarrow \text{“, ” tuple\_list} \mid \textbf{ident} \\
\text{constr\_list} &\rightarrow \text{“} \wedge \text{” constr\_list} \mid \textbf{constraint} \\
\text{stmt\_list} &\rightarrow \text{“}\wedge\text{, ” stmt\_list} \mid \textbf{stmt} \\
\text{rel\_expr} &\rightarrow \text{“\{” constr\_list “\}” “} * \text{” } \textbf{ident} \\
\text{fxn\_expr} &\rightarrow \textbf{ident} \text{ “ = ” } \textbf{ident} \text{ “(” expr\_list “)”}
\end{aligned}$$

Figure 4.2: PDFL Language Grammar

on the corresponding iterators. A space representing an $N \times M$ matrix, for example can be described by two iterators $(i,\ j)$ with constraints, $0 \leq i < N$ and $0 \leq j < M$, where $M,\ N \in \mathbb{Z}$.

3. *Functions* can represent uninterpreted functions, symbolic constants, or relations that map points in one space to those in another. An access function maps an iterator tuple referenced in a computation to the corresponding location of the data in memory. Computations are executed in lexicographic order.

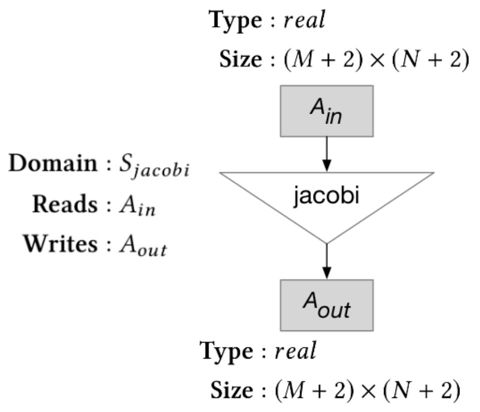
4. *Computations* are defined by an iteration space, combined with an ordered set of executable statements to be performed at each point. Each statement can be assigned a conditional expression (*guard*) that must be satisfied for the statement to be executed. Conditions that depend only on the iterators are affine. Non-affine conditions are handled with control or exit predicates [103].

Statements are assigned scheduling functions to indicate their positions within the overall execution schedule. The default ordering is lexicographic order. A computation performed over an iteration space with two iterators $(i,\ j)$, with three statements $(s0,\ s1,\ s2)$, will have the initial scheduling functions, $([i,\ j] \rightarrow s0(i{,}j{,}0),\ s1(i{,}j{,}1)$, and $s2(i{,}j{,}2)$.

The four basic language constructs correspond to nodes or operations that can be performed on the dataflow graphs. Set expressions define sets that can represent iteration or data spaces. These correspond to statement or data nodes. Relations describe transformation functions that can be applied to iteration or data spaces, and are represented by transformation nodes. Function expressions are operations that can be applied to the loop nests or data blocks, such as fusion, tiling, or rescheduling.

$$
\begin{aligned}
jacobi(t,i,j) \;=\; \{\, & 1 \le t \le T \;\wedge\; 1 \le i \le M \;\wedge \\
& 1 \le j \le N \;\} \;:\; \{ \\
A(t,i,j) = (& A(t-1,i,j-1) + \\
& A(t-1,i,j) + \\
& A(t-1,i,j+1) + \\
& A(t-1,i-1,j) + \\
& A(t-1,i+1,j)) * 0.2 \;\};
\end{aligned}
$$

(a)



(b)

Figure 4.3: (a) Specification, and (b) PDFG for the 2D Jacobi stencil calculation.

Sets can be defined as the domains of data or statement nodes. S_{jacobi} is the iteration space of the stencil statement in Figure 4.3, for example. The data domain for the A array is larger to account for the stencil access pattern, including ghost cells. Initial data space sizes are inferred from the read and write data access patterns. Read and write locations are extracted from the right and left hand sides of assignment statements, respectively.

A statement node also maintains the data mappings that represent the data locations read from and written to while the statements are executed. Statements can be defined as expressions of mapping functions, symbolic constants, or numeric literals. Each statement node represents a single loop nest, and the corresponding schedule is obtained from the original AST for that computation. Graph operations, such as *reschedule* or *fuse*, can modify the schedule of a statement node, or the data space occupied by a data node. An initial global schedule is produced by the order specified in the specification, i.e., the order that nodes are added to the graph. The PDFL script for the Jacobi stencil is provided in Figure 4.3(a), where matrix A is $M \times N$, and T is the number of time steps.

$$\begin{aligned}
fx(c,y,x) &= \{\, 0 \le c < C \;\wedge\; 0 \le y < N \;\wedge\; 0 \le x \le N \,\};\\
fy(c,y,x) &= \{\, 0 \le c < C \;\wedge\; 0 \le y \le N \;\wedge\; 0 \le x < N \,\};\\
df(c,y,x) &= \{\, 0 \le c < C \;\wedge\; 0 \le y < N \;\wedge\; 0 \le x < N \,\};\\
fx1(c,y,x) &= fx : \{\, C_{x1}(c,y,x) \;=\; \frac{1}{12} * (B_{in}(c,y,x-2) + 7 *\\
&\qquad (B_{in}(c,y,x-1) + B_{in}(c,y,x)) + B_{in}(c,y,x+1)) \,\};\\
fx2(c,y,x) &= fx : \{\, C_{x2}(c,y,x) \;=\; C_{x1}(c,y,x) \;*\; 2 \;* C_{x1}(2,y,x) \,\};\\
dx(c,y,x) &= df : \{\, B_{out}(c,y,x) \mathrel{+}= \; C_{x2}(c,y,x+1) \;-\; C_{x2}(c,y,x) \,\};\\
fy1(c,y,x) &= fy : \{\, C_{y1}(c,y,x) \;=\; \frac{1}{12} * (B_{in}(c,y-2,x) + 7 *\\
&\qquad (B_{in}(c,y-1,x) + B_{in}(c,y,x)) + B_{in}(c,y+1,x)) \,\};\\
fy2(c,y,x) &= fy : \{\, C_{y2}(c,y,x) \;=\; C_{y1}(c,y,x) \;*\; 2 \;* C_{y1}(3,y,x) \,\};\\
dx(c,y,x) &= df : \{\, B_{out}(c,y,x) \mathrel{+}= \; C_{y2}(c,y+1,x) \;-\; C_{y2}(c,y,x) \,\};
\end{aligned}$$

Figure 4.4: PDFL specification of the MiniFluxDiv benchmark in two dimensions.

The PDFL specification for the MiniFluxDiv benchmark described in Chapter 3 is given in Figure 4.4. This example demonstrates that spaces can be defined independently (e.g., *fx*) and then reused to define computations (e.g., *fx1*). The B_{in} and B_{out} data spaces represent the persistent data, input and output boxes, respectively. The C data spaces correspond to the temporary caches that can be modified to improve performance.

Each computation node has a corresponding iterator tree that describes the execution schedule. The root node connects all of the iterators. The internal nodes are iterators, and the leaf nodes statements. Edge labels indicate the order of the iterator in the resulting schedule. The scheduling function for each statement is derived by performing a depth first traversal of the iterator tree. The global schedule for the entire dataflow graph is determined by combining these individual trees as subtrees of a single root.

4.1.1 Relations and Transformations

Relations can be specified that transform one set into another, and can be applied using the multiplication operator (*), or in functional notation, e.g., $I = T(S)$. Code generation can be performed with the *codegen* function, producing a C language representation of the graph. The *graphgen* function is introduced to generate a graphical view of the statements, data, and transformations.

The *split* operation applied to a statement node partitions the domain on a given iterator by a split factor, *f*, and produces *f* new statement nodes, where the domain of each is one of the partitions of the original domain. Iterators not involved in the split are copied to the new nodes. The polyhedral dataflow graphs also support the fusion of statement nodes to produce fused loop nests. The nodes resulting from the split operation can be fused in a transformation similar to unroll and jam.

Loop unrolling is a technique for reducing loop overhead and exposing additional opportunities for parallelism. Much like the *split* operation, graph statement nodes can be unrolled by supplying a statement loop iterator to unroll, and an unroll factor, *f*. Fusing the statement sets that result from the *unroll* operation produces the unroll and jam transformation. Tiling is applied to improve both temporal and data locality of a loop nest. The general form of the *tile* function receives a set of iterators to be tiled and the corresponding tile sizes for each dimension.

4.1.2 Memory Allocation

The memory allocation algorithm traverses the graph in reverse-execution order, that is bottom to top, right to left. Output nodes are not considered as resizing is not permitted. Temporary data spaces are stored in a reference table. The table is

```
function ALLOCATEMEMORY($G : Graph$)
    $table \leftarrow$ newTable()
    for all $node \in$ reverse(dataNodes($G$)) do
        block($node$) = null
        for all $entry \in table$ do
            if not active($entry$) and
            size($entry$) $\geq$ size($node$) then
            block($node$) = $entry$
        if block($node$) = null then
            for all $entry \in table$ do
                if not active($entry$) then
                    size($entry$) = size($node$)
                    block($node$) = $entry$
        if block($node$) = null then
            $entry$ = newEntry($table$)
            active($node$) = true
            block($node$) = $entry$
        if not read($node$) and not written($node$) then
            active($node$) = false
    return $table$
```

Figure 4.5: Memory Allocation Algorithm

checked for an existing space of equal or greater size as each data node is visited. A space is marked as inactive if it is no longer being read from or written to at the current execution stage. If an existing, inactive space of adequate size is not found, a smaller inactive space will be resized. If no inactive spaces are available, a new active space will be allocated and assigned to the node. A data node is marked as inactive when the statement node that writes to it has been visited. The space requirements of a new data space are determined by the domain of the data node, the data access pattern of the statement nodes that access it, and the reuse distance resulting from the execution schedule. The algorithm is given in Figure 4.5.

4.2 Compilation Approach

The eDSL described here is the front end to the compiler IR. The polyhedral+dataflow graph intermediate representation is the next layer, consisting of statement, data, and transformation nodes, with dependences indicated by the edges between them. Transformations specified at the language level are applied at the graph level. Code for the target backend is generated from the graph after the specified transformations have been applied.

4.2.1 Derivation from Existing Code

The specification language can be produced by parsing existing code and traversing the resulting abstract syntax tree (AST). Matrix multiplication is a commonly occurring pattern in numerical applications, and is given as an example. The operation is expressed in the form, $C \leftarrow \alpha A \times B$, where $A_{M\times P}, B_{P\times N}, C_{M\times N}$ are matrices and α is a scalar. The code for this computation is given in Figure 4.6(a). The iterators and corresponding constraints that form the iteration space are extracted from the initialization statements and bounds in the *for* loops. The loop body basic block begins the statement node. The access functions, $\alpha, A(i,k), B(k,j), C(i,j)$, are derived from the statement body. The resulting PDFL specification is shown in Figure 4.6(b).

4.2.2 Graph Generation

The scope of each graph is at the function or method level. The language objects, i.e, `Space`'s and `Comp`'s, and PDFG structure, are constructed as the eDSL statements are executed. Each expression begins with a name, e.g., *spmv*,

```
for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    for (k = 0; k < P; k++) {
      C[i*N+j] += α * A[i*P+k]
         *
                         B[k*M+j];
} } }
```

(a)

$$mmul(i,j,k) \;=\; \{\, 0 \;\le\; i < M \;\wedge\; 0 \le j < N \;\wedge\; 0 \le k < P \,\} : \{\; C(i,j) \mathrel{+}= \; \alpha * A(i,k) * B(k,j) \;\}$$

(b)

Figure 4.6: Original source code for dense matrix-matrix multiplication kernel (a), and derived PDFL specification (b).

that defines the computation and becomes the corresponding statement node label. The iterator tuple, e.g., (i,n,j), defines the boundaries of the polyhedron, each with lower and upper bounds of integers. The following set is the collection of constraints in conjuctive normal form (CNF). A constraint is composed of integers, identifiers, and relational or arithmetic operators. The collection of constraints becomes the iteration space of the statement node.

Identifiers within constraint definitions that are not iterators are treated as function. The function arity is the number of arguments. A symbolic constant is a function with an arity of zero. Functions that appear with multiple arities will be assigned the maximum, and the remainder padded with zeros. Functions and constants become integer data nodes, with edges connecting them to the statement node. The last section is the set of executable expressions or statements, separated by conjunctions. Functions in this section become data nodes of *real* type. Expressions on the left hand side of an assignment become output nodes, and those on the right side are input nodes.

4.2.3 Code Generation

The code generation process first produces an initial polyhedral+dataflow graph instance from the PDFL specification. Graph operations are applied as specified and the schedule for each statement node is updated to generate the global schedule. The memory allocation algorithm in Figure 4.5 is then applied to minimize memory allocation. The iteration domains of the statement nodes are first passed to IEGenLib for simplification and normalization. The normalized expressions are translated into Omega+ calculator syntax and used to generate the loop nests.

The code generator traverses the graph in a top-bottom, left-right manner. Data allocation code, e.g., `calloc` for heap data, and array declarations for stack data, are emitted when data nodes are visited. Relations within transformation nodes are applied to set or data domains with IEGenLib. Macros are emitted when statement nodes are visited, followed by the loops generated by Omega+. Persistent data nodes become parameters to the resulting function, with input nodes marked as immutable (`const`), and output nodes as writable. Dynamic data nodes can be resized, e.g., with `realloc`.

The AST generation algorithm is given in Figure 4.7. The default generator produces C code, but other output formats can be supported by implementing additional graph visitors. Three subtrees are generated, one for initialization and allocation S_{init}, one for the computation loop nests, S_{comp}, and one for cleanup S_{free}. The scope of each graph is a function, so the three subtrees become children of the function body node. Data nodes with no incoming edges become inputs to the function, and those with no outgoing edges are treated as persistent outputs.

Data allocation statements are synthesized when data nodes are visited, and inserted into the S_{init} subtree. If the data are dynamically allocated, the corresponding deallocation statement is then added to the S_{free} tree. Allocation statements are not generated for persistent input nodes, as these are assumed to be already allocated. Instead, these are added as input parameters to the function AST subtree. The Presburger expression representing the node's domain is passed to the Omega+ polyhedral compiler for code generation. The resulting AST is then inserted into the S_{comp} subtree.

```
function genCodeAST($G = (S, D, T, E)$) {
  $S_I = (N_I, E_I)$;
  $S_C = (N_C, E_C)$;
  $S_F = (N_F, E_F)$;
  // Gen alloc/dealloc code
  for ($d \in D$) {
    $S_I \cup=$ genAllocCode($d$);
    $S_F \cup=$ genFreeCode($d$);
  }
  // Apply transformations ...
  for ($t \in T$) {
    $r =$ relation($t$);
    $s_{dst} =$ apply($r, s_{src}$);
  }
  // Gen computation code
  for ($s \in S$) {
    $S_C \cup=$ genCompCode($s$);
  }
  // Create function node
  $F =$ genFunction($G$);
  // Build AST
  return $A = (F, \{S_I, S_C, S_F\})$;
}
```

Figure 4.7: Source AST generation algorithm.

4.2.4 Data Types

The PDFG-IR allows data types to be assigned to data nodes. The IL will support this via a *datatype* function. The IL to IR algorithm, **genGraphIR** can infer data types as well. All data nodes default to the floating point type indicated in the configuration, either single or double precision. The cases where data nodes will be changed to the index data type (integer) include symbolic constants, functions used in data-dependent loop bounds, or those assigned to an iterator value.

4.2.5 Data Mapping

A data mapping assigns each instance of an access relation, e.g., $A(i,j)$, to a specific memory location. The default data mapping in PDFL is a row-major linearization of an N-dimensional array into a contiguous block of memory. The expression, $A(i,j)$, A will be allocated as $[ub(i)$ - $lb(i) + 1] \times [ub(j)$ - $lb(j) + 1]$, where ub and lb denote the upper and lower bounds, respectively. Given constraints $0 \leq i < N$ and $0 \leq j < M$, then A will be $N \times$ M. The access function for $A(i,j)$ is then $i*M+j$. For a column major data mapping, $A(j,i)$, the function would instead be $j*N+i$. The row-major access mapping in N dimensions generalizes to the following summation.

$$offset = \sum_{i=1}^{N} \prod_{j=i+1}^{N} \Big(ub(n_j) - lb(n_j) + 1\Big) n_i \tag{4.1}$$

4.2.6 Limitations

The PDFL is not a complete intermediate language, but is sufficiently expressive to encompass a wide range of regular and irregular computations. Non-affine expressions are supported with the aid of uninterpreted functions as described in the following examples. Arbitrary branching via **goto** statements is not supported. Standard **while** loops can be emulated by setting a large upper bound (e.g., `INT_MAX`) on an iterator, then using an uninterpreted function as an exit predicate to terminate the loop. Recursive algorithms are not directly supported.

4.3 Inspector/Executor Applications

The following examples demonstrate how to use the PDFL language to transform irregular codes within the context of inspector/executor applications.

4.3.1 Sparse Matrix-Vector Multiplication

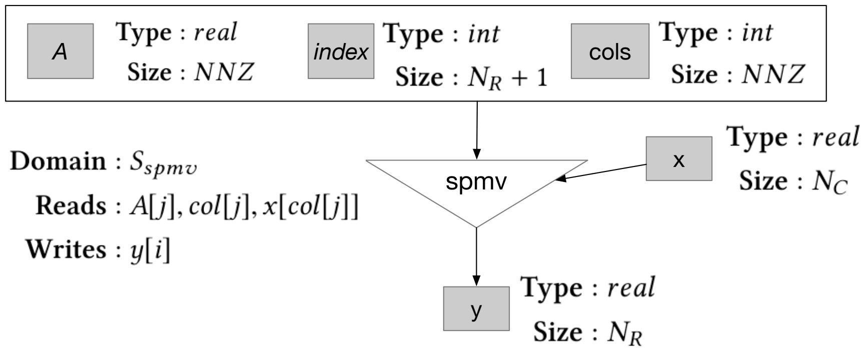
The sparse matrix-vector multiplication kernel (SpMV) for the compressed sparse row data format (CSR) requires uninterpreted functions or data-dependent loop bounds support. The CSR SpMV loop nest uses the index pointers to the compressed row indices. The symbolic constant N_R is the number of rows, N_C the number of columns, and NNZ is the number of nonzero values. The PDFL code is shown in Figure 4.8(a), the source code in (b), and the graph in (c).

$$\begin{aligned} spmv(i,j,k) \;=\; \{\, & 0 \le i < N_R \;\wedge\; index(i) \\ & \le j < index(i+1) \;\wedge\; k = col(j) \,\} : \\ & \{\, y(i) \mathrel{+}= \; A(j) \;*\; x(k) \,\} \end{aligned}$$

(a)

```
for (i = 0; i < N_R; i++) {}
  for (j = index[j]; j < index[i+1]; j++) {
    k = col[j];
    y[i] += A[j] + x[k];
} }
```

(b)



(c)

Figure 4.8: PDFL specification (a), C source code (b), and graph (b), for the sparse matrix-vector multiplication executor for a matrix in CSR format.

Inspector/executor applications require the specification of an input graph (e.g., CSR SpMV executor), as well as the transformations necessary to produce the target executor, e.g., BCSR SpMV. The executor graph will be generated by applying the specified transformations to the input executor. Unknown values in the target executor, either symbolic constants, e.g, *NB*, or uninterpreted functions, e.g., *b_index*, are identified and used to generate the inspector graph. These can be defined in terms of functions including *count*, *extract*, or *order*, or as user-defined statements. Code generated from data mapping functions copy data from the input data format to the output format if required. The inspector and executor graphs are composed so that the code generator will produce a single output with both functions. The outputs of the inspector become inputs to the executor.

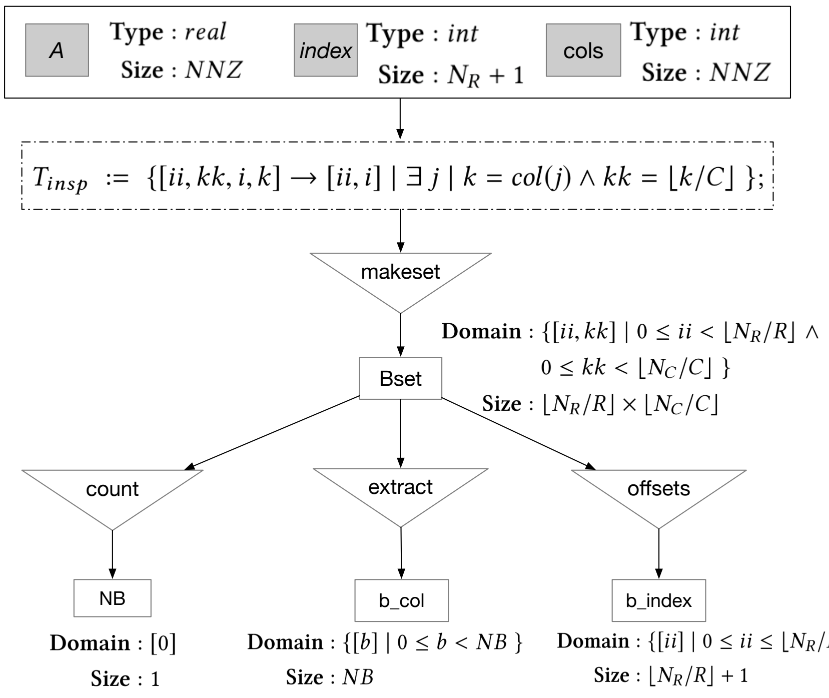
Inspector Transformations

This section describes a run-time transformation from CSR format to blocked sparse row (BCSR). The generation process begins with the specification of a graph for the original executor. In this case, the CSR SpMV graph specified in Figure 4.8 is the starting point. The transformations to move from CSR to BCSR require additional uninterpreted functions that must be clearly defined, e.g. *b_index*, *b_col*, and *NB*. The inspector is responsible for producing the explicit versions of these functions, as well as the transformed data space, denoted as *A'*. The inspector is generated as the combination of the PDFGs for each of these components. These transformations applied to the executor are listed in Figure 4.9(a), and the resulting dataflow graph is shown in (b).

The operations to produce the inspector are encoded in the polyhedral+dataflow language. The naive schedule for the inspector is shown in Figure 4.10(a). The

$$
\begin{aligned}
insp(ii, i, j, kk, k) &= \textbf{tile}(spmv, [i, k], [R, C]);\\
B_{set} &= \textbf{makeset}(insp, [ii, kk]);\\
NB &= \textbf{count}(B_{set});\\
b\_index &= \textbf{offsets}(insp, [ii]);\\
b\_col &= \textbf{extract}(insp, [kk]);\\
A\_pr(b, ri, ck) &= \textbf{copy}(insp, A(j), \{\, 0 \le b < NB\\
&\wedge\ ri = i - ii * R \wedge ck = k - kk * C \,\});
\end{aligned}
$$

(a)


A
Type : real
Size : NNZ
index
Type : int
Size : $N_R + 1$
cols
Type : int
Size : NNZ
$T_{insp} := \{[ii, kk, i, k] \to [ii, i] \mid \exists\, j \mid k = col(j) \wedge kk = \lfloor k/C \rfloor\ \};$
makeset
Domain : $\{[ii, kk] \mid 0 \le ii < \lfloor N_R/R \rfloor \wedge 0 \le kk < \lfloor N_C/C \rfloor\ \}$
Size : $\lfloor N_R/R \rfloor \times \lfloor N_C/C \rfloor$
Bset
count
extract
offsets
NB
Domain : [0]
Size : 1
b_col
Domain : $\{[b] \mid 0 \le b < NB\ \}$
Size : NB
b_index
Domain : $\{[ii] \mid 0 \le ii \le \lfloor N_R/R \rfloor\ \}$
Size : $\lfloor N_R/R \rfloor + 1$

(b)

Figure 4.9: Transformation functions to generate the CSR to BCSR inspector (a), and the resulting dataflow graph (b).

inspector can use a further compacted version of the tiled iteration space that relies only on the row iterators due to the compressed row format, with the column iterations projected as expressions. The specification to produce the CSR to BCSR inspector is given in Figure 4.10(b).

$$\begin{aligned}&tiled(ii,kk,i,j,k) \;=\; \textbf{tile}(spmv,[i,k],[R,C]);\\&\quad // \{\, 0 \le ri < R \wedge i = ii * R + ri \wedge\\&\quad // \, 0 \le ck < C \wedge k = kk * C + ck \,\}\\&b\_spmv(ii,b,kk,i,k) \;=\; \{\, tiled \;\wedge\; b\_index(ii)\\&\quad \le b < b\_index(ii+1) \;\wedge\; kk = b\_col(b) \,\} : \{\\&\quad y(i) \mathrel{+}= A'(b,ri,ck) * x(k) \,\};\end{aligned}$$

(a)

```
for (ii=0; ii < N_R/R; ii++) {
  for (b = b_index[ii];
      b < b_index[ii+1]; b++) {
    kk = b_col[b];
    for (i = R*ii;
        i<min(N_R,R*ii+R);i++) {
      for (k = C*kk;
          k<min(N_C,C*kk+C);
          k++) {
        ri = i - R * ii;
        ck = k - C * kk;
        y[i] += A_pr[b,ri,ck]*x[k
          ];
} } } }
```

(b)

Figure 4.10: Transformation functions to generate the iteration space for the BCSR executor from the initial CSR space (a), and generated code (b).

The *count* function counts the number of elements in a given set or iteration space, and in this case produces the number of nonzero blocks in the sparse matrix. The *extract* operation extracts iterator values from an iteration space as a list at run-time. In this case, the *kk* iterator values become the sparse block columns in the *b col* explicit function. The *offsets* operation produces a list of offsets from a given iterator in an iteration space. In this case, *b index* contains the offsets (i.e, number of nonzero blocks per block row.

Finally, the data transformation function to map data from the original CSR matrix, *A* to the blocked sparse row matrix, *A'* is defined. The mapping function is denoted as $R_{A \to A'}$ to indicate it is a run-time data reordering.

The initial graph is executed from top to bottom left to right, with each statement running to completion before the next begins. Read reduce fusion can be applied to the *count* and *offsets* nodes because no dependences exist between them. The same is true for the *copy* and *extract* nodes. Further fusion is limited by the dependence between the number of blocks (*NB*) and the *copy*/*extract* nodes. However, *NB* is a scalar, and is only required to allocate space for the *A_prime* and *b_col* data nodes.

The introduction of a dynamic data structure to represent those nodes satisfies the *NB* dependences and allows the *copy* / *extract* and *count* / *offsets* nodes to be fused as well. This implementation uses a dynamic array with an initial size 10% of the worst case, $NB_{max} = N_R/R \times N_C/C$. The growth factor begins at 2 and scales based on the current block density, NB/NB_{max}.

The *makeset* operation produces the set of tuples (*ii*, *kk*) within the *insp* space. The resulting B_{set} is the set of all nonzero blocks in the matrix, ensuring that each block is only counted once. A naive implementation is to allocate a two dimensional array that can accommodate all possible nonzero block coordinates. The dependence between the B_{set} node and the fused statement nodes can be considered a producer-consumer relationship. Fusing the *makeset* node with the other nodes reduces the reuse distance so that only the block columns need to be stored in B_{set}. This space reduction requires the array to be reinitialized to zero at the beginning of each block row. The *makeset* operation is split into three operations to perform this fusion, *clear*, *lookup*, and *insert*.

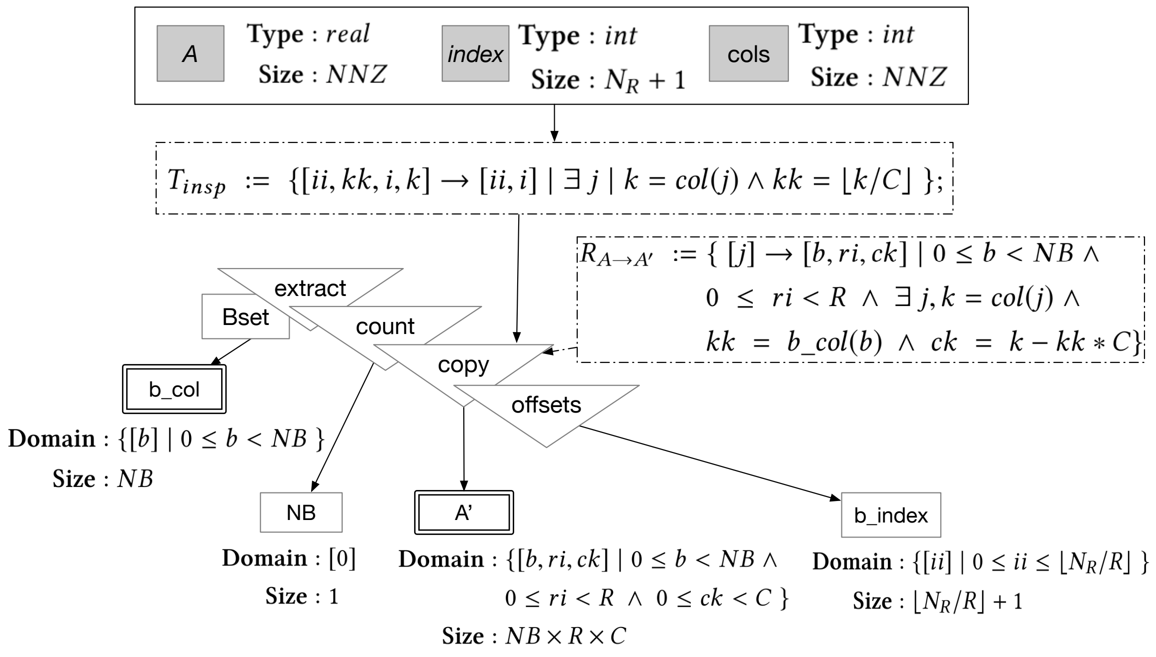
Once all the valid node groupings and fusions have been performed, and the memory allocation algorithm described in subsection 4.1.2 has been applied, the last step is to generate the final schedule. The initial schedule is obtained by performing a left to right, top to bottom traversal of the dataflow graph. Dependence analysis of the iteration spaces combined with the statement definitions for each node can be used to optimize the schedule. The *extract* operation is moved before *count* as it reads *NB*, while *count* may write it. The *clear* operation must be executed at the beginning of each iteration of the *ii* loop. Finally, the *offset* statement only relies on the *ii* iterator, but reads *NB* after updating, so is moved to the end of the *ii* loop. The resulting PDFG for the inspector and the optimized code are given in Figure 4.11. The double line boundaries on the *b_col* and *A'* nodes indicate dynamic reallocation.

Executor Generation

The initial executor is produced directly from the S_{exec} space in Figure 4.9(a). The inner two loops of the BCSR executor can be completely unrolled if the tile sizes are known at compile-time. These transformations are supported by the *unroll* graph operation. Another optimization that makes the executor competitive with previous work is the insertion of temporary storage buffers of size R and C to prefetch the values of the row vector y and the column vector x, respectively. This is supported in the dataflow graph with the *copy* operation, by mapping subsets of the vector data into the buffers.

4.3.2 Matricized Tensor Times Khatri-Rao Product

The sparse coordinate format (COO) is a common structure to represent sparse tensors, for example in the Matrix Market and FROSTT [91] formats. One of the primary computations in tensor-based data analysis is factorization. Canonical polyadic decomposition (CPD) is a common factorization technique, analogous to single value decomposition in matrices. The matricized tensor times Khatri-Rao product (MTTKRP) kernel can be a bottleneck in CPD calculations. The Khatri-Rao product is the Kronecker product between a third-order tensor B, and two matrices C and D, denoted as $A_{I\,x\,J} = B_{I\,x\,K\,x\,L} \otimes C_{K\,x\,J} \otimes D_{L\,x\,J}$, where $I,\ J,\ K,\ L$ are the dimensions.


A
Type : $real$
Size : NNZ
index
Type : int
Size : $N_R + 1$
cols
Type : int
Size : NNZ
$T_{insp} := \{[ii, kk, i, k] \rightarrow [ii, i] \mid \exists\, j \mid k = col(j) \wedge kk = \lfloor k/C \rfloor \};$
$R_{A \rightarrow A'} := \{ [j] \rightarrow [b, ri, ck] \mid 0 \leq b < NB \wedge 0 \leq ri < R \wedge \exists\, j, k = col(j) \wedge kk = b\_col(b) \wedge ck = k - kk * C \}$
extract
Bset
count
copy
offsets
b_col
Domain : $\{[b] \mid 0 \leq b < NB \}$
Size : NB
NB
Domain : $[0]$
Size : 1
A'
Domain : $\{[b, ri, ck] \mid 0 \leq b < NB \wedge 0 \leq ri < R \wedge 0 \leq ck < C \}$
Size : $NB \times R \times C$
b_index
Domain : $\{[ii] \mid 0 \leq ii \leq \lfloor N_R/R \rfloor \}$
Size : $\lfloor N_R/R \rfloor + 1$

(a)

```
for (ii = 0; ii < N_R/R; ii++) {
 memset(Bset,0,(N_C/C+1)*sizeof(int));
 for (i = R*ii; i < min(N_R,R*ii+R); i++) {
   for (j=index[i];j < index[i+1]; j++) {
    k = col[j];
    kk = k/C;
    b_col[NB]=kk;
    b=Bset[kk];
    if (!b) {
      NB++;
      Bset[kk] = NB;
    }
   A_pr[b-1,i-R*ii,k-C*kk] = A[j];
 }  }
 b_index[ii+1] = NB;
}
```

(b)

Figure 4.11: Optimized PDFG for the CSR to BCSR inspector (a), and the generated code (b).

$$\begin{aligned}
&coo(m,i,j,k,r) \;=\; \{\; 0 \le m < M \;\wedge \\
&\quad i = index(0,m) \;\wedge\; j = index(1,m) \;\wedge \\
&\quad k = index(2,m) \;\wedge\; 0 \le r < R \;\} \;:\{ \\
&\quad A(i,r) \mathrel{+}= B(m) * C(j,r) * D(k,r) \;\};
\end{aligned}$$

(a)

$$\begin{aligned}
&csf(p,i,q,j,m,k,r) \;=\; \{\; offset(0,0) \le \\
&\quad p < offset(0,1) \;\wedge\; i = indices(0,p) \;\wedge \\
&\quad offset(1,p) \le\; q < offset(1,p+1) \;\wedge \\
&\quad j = indices(1,q) \;\wedge offset(2,q) \le \\
&\quad m < offset(2,q+1) \;\wedge \\
&\quad k = indices(2,m) \;\} \;*\; coo;
\end{aligned}$$

(b)

Figure 4.12: Specifications for the COO-MTTKRP executor (a), and transformation statement for the CSF executor (b).

The COO format for tensors is a list of nonzero values and the corresponding indices for each mode. The *index* array is a two dimensional array of indices with size $M \times N$, where M is the number of nonzeros, and N is the order of the tensor. The *val* array contains the M nonzero values. The COO representation of the MTTKRP kernel for a third-order tensor (N=3) can be represented by the specification in Figure 4.12(a).

The compressed sparse fiber (CSF) format for sparse tensors [98] is a generalization of CSR or CSC for matrices. The modes, or dimensions, of the tensor are compressed into a tree-like structure such that only the unique index values for each are stored. The first dimension will have the most compression, followed by the second, and none for the last, resulting in M leaf indices, one for each nonzero value. The representation requires two index arrays, one to store the actual coordinates for each mode, and another to store the offsets into those arrays. The relation to transform a MTTKRP executor for a third order tensor in COO format to CSF is given in Figure 4.12(b).

Inspector Transformations

The *newnode* uninterpreted function determines whether to create a new node at the child level, i.e., the level that will be visited in the subsequent iteration of n. This function evaluates to true if the current index is different at the current dimension and the leaf level (last dimension) has not been reached. The *extract*, *offset*, and *copy* nodes can all be fused by again using dynamic arrays to represent the *indices* and *offset* data sets. The PDFL specification is given in Figure 4.13(a), the optimized graph is displayed in (b), and the generated code in (c).

Executor Generation

The sparse matrices tested in the BCSR example were comparatively small, with little benefit derived from automatic parallelization. Sparse tensors can be quite large, so the ability to insert OpenMP pragmas into the generated code becomes quite valuable. The heuristic for this process is straightforward. The iterators in each loop nest, i.e., set of fused statement nodes, are traversed and their constraints checked for uninterpreted functions. The outermost loop without data-dependent bounds has a parallel for pragma inserted above it, and a SIMD pragma at the innermost loop bound that is not data-dependent.

For the MTTKRP executor, this results in a parallel for on the p loop. This loop does depend on the *offset* function, but no outer loop relies on it, so it can be safely parallelized. The innermost j loop is dense, so vectorization is applied.

4.4 Experimental Evaluation

The experiments performed to evaluate this work include inspector/executor applications for the CSR to BCSR matrix and COO to CSF tensor transformations. Each test was performed nine times and the median value is reported. These benchmarks demonstrate the applicability of these dataflow graphs to both regular and irregular applications. The combination of scheduling and dataflow optimizations within the polyhedral model can improve performance results in either of these problem domains.

4.4.1 Target Architecture

The experiments were conducted on a single node of a research cluster. Each node contains an Intel Xeon E5-2680 v4 dual socket CPU with 28 cores, clocked at 2.40 GHz. The cores include 32KB L1, 256KB L2 caches, and each shares a 35840K L3 cache. The nodes have 192GB of DRAM with two NUMA domains. The benchmarks were compiled with the 7.2 version of the GCC compiler with the -O3 optimization flag. The nodes were running the CentOS 7.5 operating system.

4.4.2 CSR to BCSR for Sparse Matrices

The inspector/executor approach for conversion of a sparse matrix from the CSR to BCSR data format has been used to motivate the application to sparse data structures. The PDFG implementation was compared with previous work in CHiLL [39] and the OSKI sparse matrix kernel library [104] on several sparse matrices in the SuiteSparse Matrix Collection [90]. The best performing block size, 8 x 8, was selected for evaluation. The run times were separated by inspector and executor.

The CHiLL code is produced using a script that converts the CSR implementation into a single function containing the inspector followed immediately by the executor. The OSKI inspector creates a CSR matrix, provides the fixed block size as a tuning hint, and then tunes the vector on the SpMV operation. The PDFG version is produced by composing the inspector and executor graphs, and performing code generation, resulting in two separately timed functions. The inspector results are displayed in Figure 4.14(a) and the executor in Figure 4.14(b), respectively.

The dynamic array technique used to reduce the number of passes through the matrix in the PDFG inspector implementation outperforms the linked list version produced by CHiLL for most matrices. The CHiLL executor remains the faster than OSKI, but the PDFG version is competitive. The OSKI implementation performs nine autotuning steps, and this is reported as the inspector time.

4.4.3 COO to CSF for Sparse Tensors

The COO to CSF inspector/executor transformations is compared with the TACO compiler [105] and the SPLATT library [31]. The three implementations are tested on several sparse tensors from the FROSTT [91] repository, including the NIPS, Enron, and NELL-2 tensors. Each experimental run was executed five times, with the median value reported.

The TACO code was generated using the `taco` command line utility, by specifying the MTTKRP computation with sparse tensor B and dense matrices, A, C, and D. The resulting code was then inserted into an executor and timed. The SPLATT implementation simply invokes the built-in MTTKRP function included in the library. The results are given in Figure 4.15. The PDFG and TACO implementations yield very similar results. This is expected as both produce nearly identical source code.

The SPLATT method performs better on the `nell2` and `crime` tensors, but not as well on `nips` and `enron`.

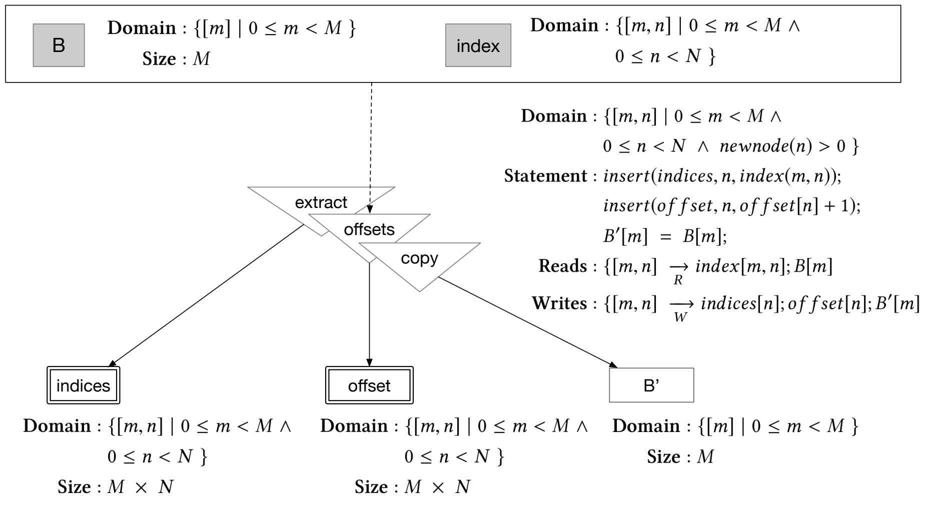
4.5 Summary

The polyhedral+dataflow language and graph implementation described in this chapter combine execution schedule transformations with dataflow optimizations. The language can be derived from another programming language or intermediate representation. The support for sparse data structures allow the optimizations to be applied to both regular and irregular applications. This versatility makes them applicable to PDE solvers, stencils, or sparse linear algebra kernels.

The high-level language can improve the productivity of engineers and scientists by allowing computations to be specified concisely as mathematical expressions, while the intermediate representation can act as a performance portability layer to enhance the optimization of existing applications or other representations. Visual feedback is also provided to the user when code transformations are applied.

$$
\begin{aligned}
&insp(m,n) \;=\; \{\, 0 \le m < M \;\wedge \\
&\quad 0 \le n < N \;\wedge\; newnode(m,n) > 0 \,\}; \\
&extract(m,n) \;=\; \{\, insp \,\} : \{ \\
&\quad \mathbf{insert}(indices(n), index(m,n)) \,\}; \\
&offsets(m,n) \;=\; \{\, insp \,\} : \{ \\
&\quad \mathbf{insert}(offset(n), offset(n)+1) \,\}; \\
&copy(m) \;=\; \{\, 0 \le m < M \,\} : \{ \\
&\quad B\_pr(m) \;=\; B(m) \,\};
\end{aligned}
$$

(a)

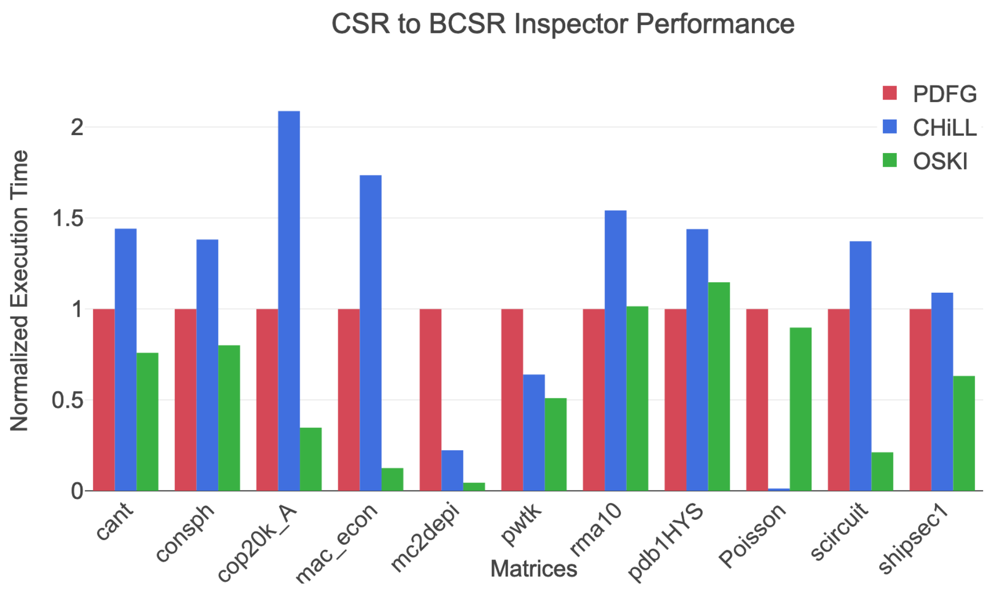

B
Domain : {[m] | 0 ≤ m < M }
Size : M
index
Domain : {[m, n] | 0 ≤ m < M ∧ 0 ≤ n < N }
Domain : {[m, n] | 0 ≤ m < M ∧ 0 ≤ n < N ∧ newnode(n) > 0 }
Statement : insert(indices, n, index(m, n)); insert(offset, n, offset[n] + 1); B′[m] = B[m];
Reads : {[m, n] →R index[m, n]; B[m]
Writes : {[m, n] →W indices[n]; offset[n]; B′[m]
extract
offsets
copy
indices
Domain : {[m, n] | 0 ≤ m < M ∧ 0 ≤ n < N }
Size : M × N
offset
Domain : {[m, n] | 0 ≤ m < M ∧ 0 ≤ n < N }
Size : M × N
B′
Domain : {[m] | 0 ≤ m < M }
Size : M

(b)

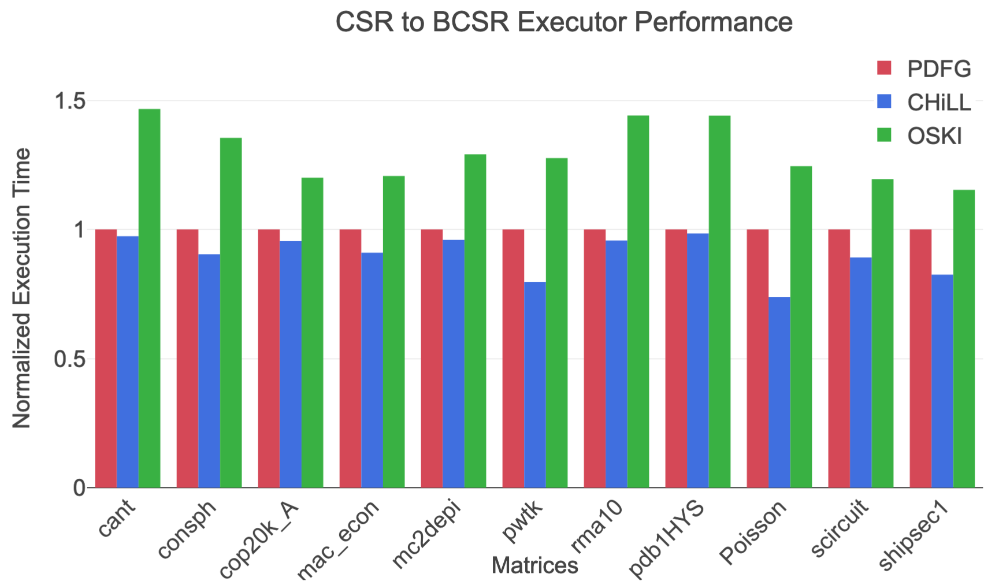
```
for (m = 0; m < M; m++) {
  for (n = 0; n < N; n++) {
    if (newnode(m,n) > 0) {
      insert(indices[n],index[m,n
          ]);
      insert(offset[n],offset[n
          ]+1);
    }
   B_pr[m] = B[m];
} }
```

(c)

Figure 4.13: PDFL specification (a), optimized PDFG (b), and the generated code (c) to produce the COO to CSF inspector.

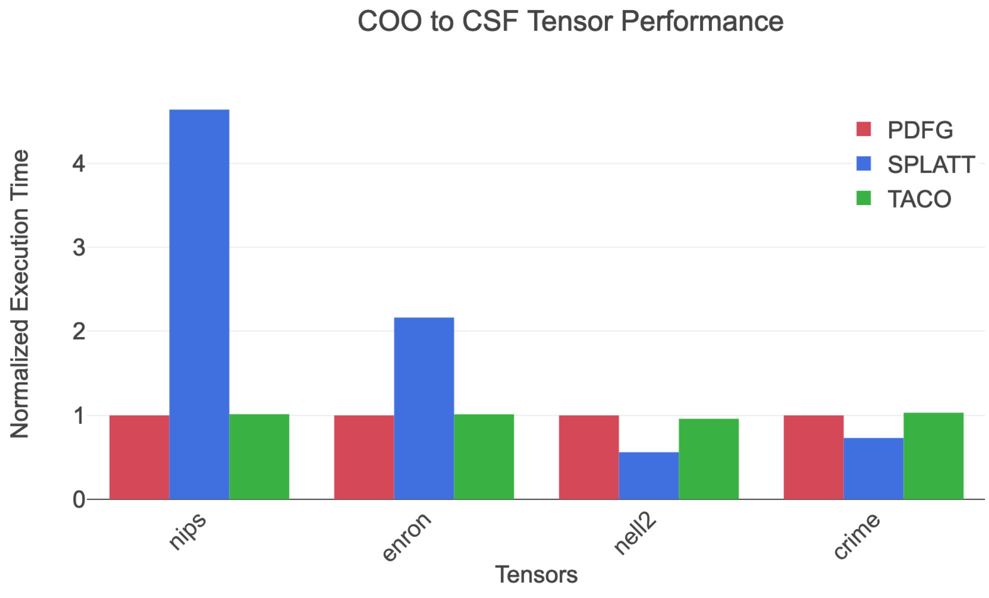


(a)



(b)

Figure 4.14: CSR to BCSR sparse matrix transformation performance for the (a) inspector and (b) executor between the CHiLL, OSKI, and PDFG methods.



(a)

Figure 4.15: COO to CSF tensor format results between PDFG, SPLATT, and TACO methods.

CHAPTER 5

STRUCTURED GRID SOLVER INTEGRATION

This chapter describes `Proto`, a DSL for structured grid applications, and its integration with the polyhedral+dataflow intermediate representation to achieve performance portability for shared memory, multi-core CPU and GPU backends.

`Proto` is a lightweight library designed for efficient solution of differential equations on domains that are composed of unions of structured, logically rectangular grids. The DSL improves the productivity of computational scientists through an intuitive programming interface that seamlessly integrates with an existing AMR framework. The goal of `Proto` is to decouple the precise description of a finite-difference discretization of a partial differential equation, and how that algorithm is executed on a specfied computer architecture. The `Proto` library includes support for CPU and GPU computations.

Embedded domain specific languages allow developers to add functionality to an existing language like C++ with mature compiler infrastructures such as GCC, Clang, and Intel (ICC). However, it can be difficult to optimize codes implemented in a high-level representation as the compiler cannot easily optimize code across several layers of abstraction. Challenges include limited data reuse, large quantities of temporary storage, and low arithmetic intensity in many small kernels. Parallelizing such kernels for multi-threaded architectures can suffer from excessive overhead from many fork/join calls in the case of OpenMP, or kernel launches in CUDA.

The intermediate polyhedral+dataflow representation [15] addresses this challenge by collecting computation information in a dataflow graph, then fusing nodes to increase AI, minimize fork/join overhead, perform storage reductions to eliminate unnecessary memory traffic, SIMD vectorization, and apply tiling to improve data locality. The dataflow representation is combined with a performance model that estimates FLOPs and memory throughput to guide optimizations and generate optimized code variants. The experimental results indicate that a fully fused and tiled code variant that increases arithmetic intensity, while reducing the working set size can achieve a performance speedup of up to 3X. GPU speedups up to 2.6X are also observed.

5.1 Background

This section provides background information on Chombo, as an example of an application framework that solves partial differential equations (PDEs), and the Euler equations that will be used as a motivating example in this paper.

5.1.1 Chombo and AMR

The Chombo [100, 106, 107] package supports conservative discretizations of complex PDEs. It provides programming abstractions for iterations spaces, data spaces, and more. The discretized problem domain comprises a set of boxes that each comprise a subset of the points in the domain. Chombo is used in a variety of scientific applications and is designed to perform well on many compute resources ranging from laptops to leadership class supercomputers [108–113].

Adaptive mesh refinement saves time and energy by refining sections of the problem domain based on the complexity of the phenomena modeled in that area. Areas where little change is taking place remain at a courser granularity and, therefore, require fewer compute resources to include in the simulation. The Chombo C++ Library is designed to support these kind of applications running across all modern supercomputing platforms. `Proto` is intended to support the same types of applications as Chombo with a high-level programming model that can be executed on heterogeneous architectures.

5.1.2 Euler Equations

Our running example is an implementation of the Euler equations. These provide a manageable example of a partial differential equation system that requires the properties of SAMR discretization methods that are highly localized in space or time features that develop due to nonlinearities. The Euler equations in fluid dynamics are quasilinear hyberbolic equations that are a special case of the Navier-Stokes with zero viscosity (inviscid), and zero thermal conductivity (adiabatic). They can be applied to both compressible and incompressible fluid flows [114]. The method described in this work is an implementation of the 4th-order Method Of Lines published in [82] and written in the `Proto` DSL.

A fourth order Runge-Kutta method is applied to solve for $\mathbf{u}$, the flow velocity vector. The solution is advanced by integrating over multiple time steps until some target time is reached, or a maximum number of iterations has been performed. The current state becomes the input to compute the next output state after each time increment. The step function is executed four times, forming the bottleneck of the solver.

Each point in the grid contains a component vector (ρ, p_x, p_y, p_z, e) where ρ is the density, p is momentum in each direction, and e is energy. These are the conserved values. The first operation in the step function converts the conserved values into their primitive counterparts, performed by the *consToPrim* function. A deconvolution stencil is applied to the input box, and the result is also converted to primitives. These two are added together and a Laplacian stencil is applied to compute the average. Interpolation is then performed for each dimension, with both a high and low wave speed calculated. This operation includes two stencils, an upwind state computation, a deconvolution, two flux calculations, a Laplacian, and a divergence with each added to the final result to produce the new state.

5.2 Proto Overview

`Proto` derives from earlier work on `AMRStencil`, a domain-specific language developed as part of the D-TEC project [115]. That effort relied upon a true augmentation of the underlying C++11 language specification with stencil-based language features. Learning a lesson from the transition of UPC [116] to UPC++ [117], it was determined that C++ is now powerful enough to describe language semantics from within a C++ template library itself, thus separating DSL development from compiler semantics.

`Proto` is a lightweight C++ library developed to efficiently solve differential equations over domains formed from the union of structured, rectangular grids. The goal is to decouple the complexities of *designing* an algorithm from the *scheduling.* The `PDFL` language and IR share this as a common goal. `Proto` contains a number of high-level constructs for achieving this goal. A `Point` represents a point $\in \mathbb{Z}^D$, a

D-dimensional integer space. A `Box` encloses a subset, B, of $\mathbb{Z}^D$, a rectangular domain over an array. Each is described by a pair of points, (l, h), for example bottom-left, and top-right in the 2D case. `Proto` boxes support many transformations that lend themselves to be supported by the polyhedral model, such as intersection, shifting, and coarsening or refinement.

Boxes describe a discretization of physical space, while data represent components in the state space. Data are encapsulated with boxes in a `BoxData` object. `Proto` uses C++ templates to an arbitrary type $\mathbb{T}$, that can either be the real numbers, $\mathbb{R}$, complex numbers, $\mathbb{R}$, or integers $\mathbb{Z}$. The data associated with each point can be a scalar value, a component vector of length, C, a component matrix $(C \times D)$, or a tensor $(C \times D \times E)$. Box ranges are computed at runtime, while component indices are known at compile time.

There are two primary ways to represent computations in `Proto`. The `forall` operation receives as inputs a function pointer, `F`, `Box`, and an arbitrary (variadic) number of parameters, including data boxes or scalars. If the box is omitted, the operation will be applied to each point in the intersection of the supplied `BoxData` objects. Finally, `Proto` supports the creation of arbitrary stencil objects at runtime, where each consists of a set of offsets (as points), and the corresponding coefficient weights. Stencils can be added, multiplied by scalars, or composed to create new stencils. Class methods in `BoxData` enable other pointwise operations via operator overloading, e.g., addition or scalar multiplication.

5.2.1 Euler in Proto

The `Proto` implementation to solve for velocity using the Euler equations is given in Figure 5.1. The input vector, **U** is the flow velocity vector. The `consToPrim` function converts the conserved quantities, i.e., momentum, into primitives, i.e., velocities in each direction. The input data is deconvolved into a local vector with the `deconvolve` stencil. The `laplacian` stencil computes the average velocity, and the deconvolved primitives are added to the result. Lower and upper interpolations are performed on the average velocity for each dimension. The fluxes for each dimension are then computed, and the divergence of the average is added to the output vector. Finally, each point is multiplied by the negative inverse of the step size (`dx`). Stencils are applied to each point in the data space including the component space. A `forall` statement executes the function on each point in the data space by operating on the component space. Arithmetic operations (e.g., +=) are applied to all points in the data and component spaces.

```
Vector W_bar = forall<double,C>(consToPrim,U_in,gamma);
Vector U = deconvolve(U_in);
Vector W = forall<double,C>(consToPrim,U,gamma);
Vector W_ave = laplacian(W_bar,1.0/24.0);
W_ave += W;

for (int d = 0; d < DIM; d++) {
  Vector W_aveL = interpL[d](W_ave);
  Vector W_aveH = interpH[d](W_ave);
  Vector W_ave_f = forall<double,C>(upwind,W_aveL,W_aveH,d,
      gamma);
  Vector F_bar_f = forall<double,C>(getFlux, W_ave_f,d,gamma);
  Vector W_f = deconvolve_f[d](W_ave_f);
  Vector F_ave_f = forall<double,C>(getFlux,W_f,d,gamma);
  F_bar_f *= (1 / 24);
  F_ave_f += laplacian_f[d](F_bar_f,1.0/24.0);
  U_out += divergence[d](F_ave_f);
}
U_out *= -1 / dx;
```

Figure 5.1: `Proto` implementation of the step function from the Euler solver.

5.3 Compiler Approach

Computations in Proto are executed directly in the C++ code of the algorithm specification at run time. This paper proposes a compiler-based approach that collects the details of the computation, builds a dataflow graph intermediate representation, applies loop transformations and storage reductions, then generates optimized code to perform the same computation in less time.

5.3.1 Intermediate Representation

The Proto code is translated into the polyhedral+dataflow intermediate representation via the PDFL embedded DSL that defines computations in the IR. Loop fusion is one of the primary transformations applied to the dataflow graph IR. Fusing two statement nodes results in the union of their iteration spaces and computations. An additional data structure is introduced to represent the internal execution schedule within a group of fused nodes to ensure that all producer-consumer relationships are maintained correctly.

5.3.2 Mapping Proto to Polyhedral+Dataflow IR

The entities defined in Proto are analagous to those in the PDFG-IR. A point in Proto is equivalent to an instance of an iterator tuple in PDFL. Collections of points are represented by Box objects. These are equivalent to iteration spaces in PDFL. BoxData objects in Proto correspond to data spaces.

The forall operation is represented by a computation in PDFL. The PDFG representation of a Proto kernel is transformed into C code before running. Function pointers are not directly supported since the compiler cannot determine the source of the original code. `Proto` kernel functions are expressed in the `PDFL` eDSL. The corresponding iteration space is built by adding the spatial dimensions in reverse order. In the 2D case, the x-dimension would become the inner loop, with y as the outermost loop.

Stencils in `Proto` are also mapped to computations in `PDFL`. A stencil, S, is represented by a point matrix, P of size $n \times d$ where n is the number of stencil points, and d the dimensions, and a coefficient vector, $\mathbf{c}$ of size n. The five-point Laplacian stencil, for example, would be represented by the offset points $\{(0,0),(1,0),(-1,0),(0,1),(0,-1)\}$ and coefficient weights $(-4,1,1,1,1)$.

A stencil is applied to all components so the iteration space must include the component space. Component loops are initially placed as the outermost loops. A 2D stencil over a box of N cells with component vector of length C would produce the following iteration space.

$$S \ = \ \{\ [c,y,x] \mid 0 \leq c < C \ \wedge \ 0 \leq y < N \ \wedge \ 0 \leq x < N \ \} \tag{5.1}$$

The offsets and weights in the stencil are unrolled to become a weighted sum expression. Applying the stencil to a data space, W, produces this expression:

$$\begin{aligned}\bar{W}(c,y,x) = -4\,W(c,y,x) + W(c,y,x+1) + W(c,y,x-1) + \\ W(c,y+1,x) + W(c,y-1,x)\end{aligned} \tag{5.2}$$

Data accesses are denoted by a data space followed by an iterator tuple (e.g., $W(y,x)$). Accesses on the right hand side of an assignment are assumed to be reads, and the associated statement node a consumer. Those on the left are considered writes and the statement node a producer. The default data mapping is a linearization of the data as a one dimensional array, e.g., $W(y, x) \rightarrow W[N * y + x]$.

`Proto` objects are transformed into PDFG-IR by an interface layer. As each `Proto` method is executed, the corresponding `PDFL` objects are generated while the PDFG-IR is constructed. Each time a function is called, an automatically incrementing identifer is assigned to prevent conflicts (e.g., *getFlux1*, *getFlux2*, etc.). After a single pass through the `Proto` kernel has been completed, the initial, serial dataflow graph is created. The PDFG for the two-dimensional Euler step function is given in Figure 5.2. The dimensional loop from the `Proto` code is effectively unrolled.

Representing control flow in a dataflow language is challenging. Some `Proto` functions, e.g., `upwind`, require control flow. This has been supported in `PDFL` by the inclusion of a conditional expression that implements the ternary operator. Intermediate computations in temporary variables are replaced with the actual expression. This is a form of redundant computation, and can help relieve register pressure or reduce memory traffic.

5.3.3 Performance Modeling

A performance model is generated for each graph variant. Floating point operations (FLOPs) are counted from eDSL operations. Read/write traffic is estimated from iteration space sizes and producer/consumer access mappings. The total number of bytes allocated are computed from the size of each data space. The number of active input/output streams at any point in the execution is determined from the incoming and outgoing dataflow graph edges.

Arithmetic intensity is computed from FLOPs and memory traffic estimates. Estimates are correlated to the results of the Intel VTune/SDE and LIKWID [118]

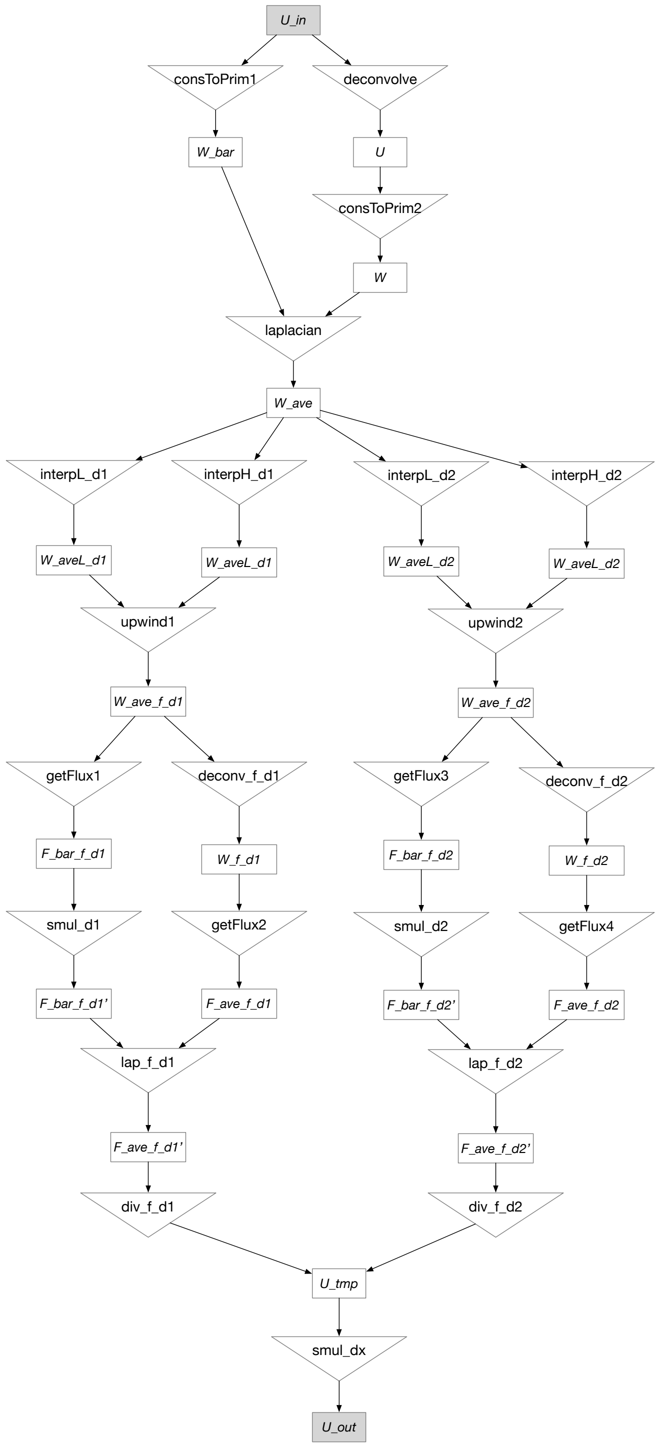


Figure 5.2: Polyhedral+dataflow graph for the Euler step function.

performance modeling tools. The model is applied to predict the profitability of IR transformations, and computes estimates that are hardware independent, i.e., an application signature. The performance model for three graph variants, including series of loops, partially fused, and fully fused are summarized in Table 5.1. The table demonstrates the correlation between reduced storage and increased arithmetic intensity with performance speedup over the original `Proto` implementation. The total number of FLOPs remains constant at 433 MFLOPs.

Table 5.1: Performance model for the three Euler graph variants, indicating the relationship between storage reduction, increased arithmetic intensity, and performance speedup.

Variant	Allocated (MB)	Processed (MB)	A.I.	Speedup
Series of Loops	55.3	1,860	0.233	1.4
Partially Fused	118	1,400	0.310	1.9
Fully Fused	80.7	764	0.567	3.1

5.3.4 Shift and Fuse Algorithm

The automatic fusion of loop nests requires that data dependences be satisfied to ensure correctness. The fusion algorithm consists of three steps. The iteration spaces of the computation nodes are first compared to determine whether loop interchange is required. Interchange is necessary when pointwise methods (i.e., `forall` operations) are fused with stencils, for example, because the pointwise methods do not have component loops. The component loops are moved to the inside as their bounds are known at compile time (the e.g., C, D, E template parameters), and are relatively small with respect to the spatial loops (i.e., $C \ll N$). These innermost loops become candidates for unrolling, allowing the innermost spatial loop (e.g., x) to be vectorized. Interchange is performed by exchanging the nodes within the iterator tree.

The next step calculates any iterator shifts that may be necessary. This requires finding the computation nodes within the current fusion group that produce data required by the node being fused. The data access mappings for each producer are processed to determine the maximum reuse distance. This distance is then added to the difference between the loop start bounds of the fused node and the current node. The result becomes the shift tuple, one per iterator, of the node being fused.

Finally, the iterator tree must be updated to position the new node within the fusion group. To accomplish this, a depth first search of the iterator tree is performed for each of the producer nodes in the previous step, returning the path through the tree. The fused node is then inserted into iterator tree at a position one greater than the maximum position of its producers, ensuring that it is not executed until the data it must read have been written.

As a motivating example, fusing the `laplacian` node with `consToPrim1` from Euler demonstrates each of the three steps. The component loop iterator, c in the `laplacian` node is interchanged before fusing. Each of the spatial iterators is shifted by 1 as the reuse distance of the stencil is 1-(-1)=2, and the loop bounds differ by -1 in each direction. In the last step, the `laplacian` node is inserted after `consToPrim1` because it produces the `W_bar` data that `laplacian` consumes. Figure 5.3(a) contains the original iterator trees for the two statement nodes, the interchanged and shifted `laplacian` tree is displayed in (b), and the resulting fused iterator tree in (c).

Once the serial version of a graph has been generated, additional variants are generated by applying transformations to the original. This can be done manually using eDSL methods such as `fuse`, `split`, or `tile`, or variants can be generated automatically.

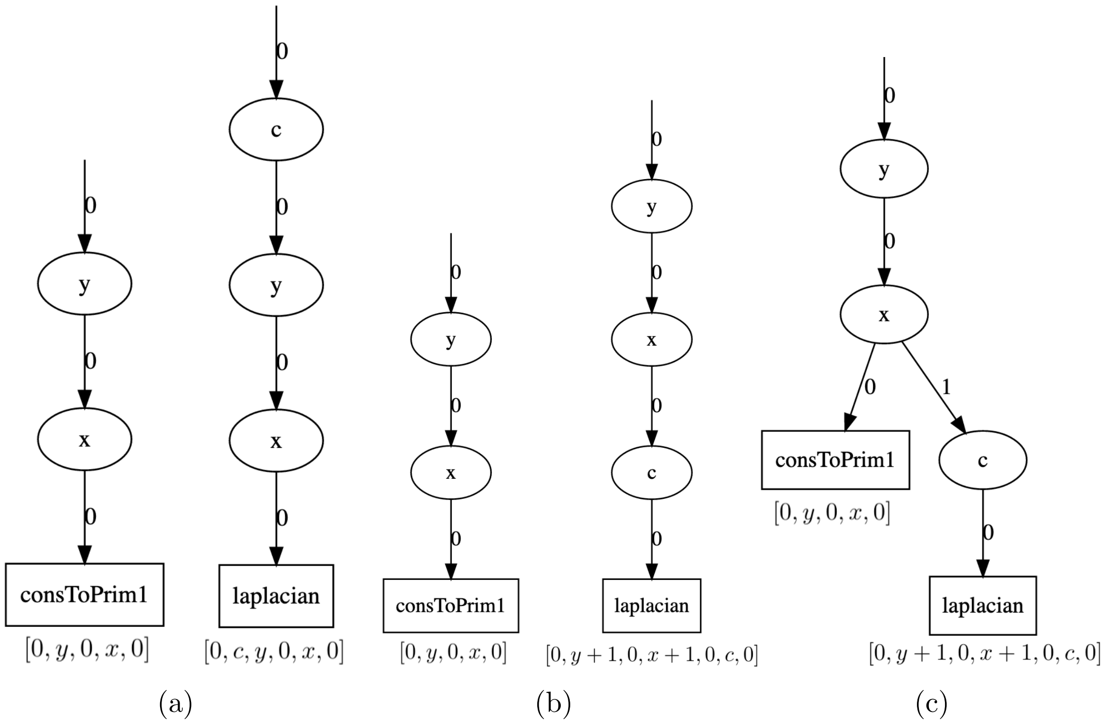


Figure 5.3: Iterator trees for the (a) original `consToPrim1` and `laplacian` computations, (b) `laplacian` after interchange and shift applied, and (c) final fused tree.

The PDFG infrastructure consists of a multi-pass system. Passes are implemented as visitors on the dataflow graph, that either annotate nodes with attribute values such as iterator shifts or tile sizes, or can produce a new transformed graph. Additional passes can be introduced by implementing new visitors. The original graph is not modified, so the passes can be applied in an arbitrary order, although visitors can be composed.

The passes are performed in a specific order by default. The schedule visitor traverses the computation nodes, then walks the iterator trees of each to produce the scattering functions needed by the polyhedral compiler. The performance model visitor traverses the graph to build the model, annotating the nodes with FLOPs, data reads and writes, and the number of input and output streams.

The data reduction visitor minimizes the temporary storage within fused node groups based on the reuse distance between data nodes. The distances are inferred from the data accesses in the statement nodes that read or write the data nodes. A data space is reducible if it is produced and consumed within the same fusion group. A reuse tuple is generated that contains the distance for each iterator. Those with a reuse of zero can be reduced to the size of a scalar, a component vector, or one spatial dimensions. The data space size and access mappings are updated accordingly.

The memory allocation visitor traverses the graph in reverse order and assigns each data space a memory location using liveness analysis, further reducing data allocation. This ensures that only sufficient memory that is required for the the currently live data spaces needs to be allocated. This leads to a balance between loop fusion and memory allocation.

The parallelization visitor decorates the iterator tree of each statement node with either thread level parallelism for outer loops, or SIMD parallelism for innermost loops. These tags are converted into pragmas during code generation. Loops that have been automatically shifted are not parallelized.

The transformation visitor attempts to produce an optimal version of the graph using the performance model that reduces control flow and temporary storage, and enables vectorization opportunities. Decisions made include whether to fuse two nodes or sets of nodes, perform loop interchange, unroll inner loops, or apply tiling.

5.4 Code Generation

Code generators are implemented as visitors on the dataflow graphs. Statement nodes are output as loop nests, data nodes as memory allocation statements, and data mappings as macros. The default generator also includes any necessary headers, and defines any other functions required in the code body. Data nodes that have no incoming edges (sources) become input parameters, and those without outgoing edges (sinks) become outputs, unless otherwise specified. Internal data nodes are assumed to be temporary storage and subject to reduction. Loop nests are generated by the *Omega+ polyhedral compiler*, and modified or annotated as needed by the code generator.

OpenMP pragmas are inserted into the loop nests as previously determined by the parallel visitor. The memory allocation statements and access functions are modified so that dedicated memory spaces are assigned to each thread. The maximum number of active threads is computed from the upper bound of the loop being parallelized. The remaining threads are applied over boxes.

Code variants can optionally be validated after generation against the data produced by the execution of the `Proto` code that produced the initial dataflow graph. Variants that do not match the desired output within a given error threshold are discarded.

5.5 Experimental Evaluation

Performance results were collected using an implementation of the Euler fluid equations [119] solver in `Proto`. The step function applied by the fourth order Runge-Kutta method at each time step is the most computationally intensive method.

An initial, serial version of the graph was generated from one pass through the Euler step function. Several code variants were produced by manipulating the dataflow graph IR and generating the resulting code. The performance results indicate that scheduling transformations are more effective when coupled with dataflow optimizations. The performance model predicts that the fastest variants are those that maximize arithmetic intensity, while reducing the data sufficiently to keep the working set size within the L3 cache.

5.5.1 Experimental Setup

The Euler step function was evaluated by computing boxes of size 64^3, with one box allocated to each parallel thread. Each experimental run was performed nine times with the mean time reported. Execution times are normalized with respect to the `Proto` implementation and output data were validated against the same to ensure correctness.

The CPU experiments were performed on single nodes of the Cori cluster at NERSC. Each Haswell node consists of a dual socket, Intel Xeon E5-2698 v3 CPU clocked at 2.30 GHz, each with 32 physical cores, 16 per socket, and 64 logical cores with hyperthreading. There are 64K of L1, 256K of L2, respectively, with 40960K of shared L3 cache. Each node contains 128GB of DRAM distributed over two NUMA domains, with a 2GB block size. The code variants were compiled using Intel compiler (ICC) v19.0.3 at optimization level -O3.

The GPU data were collected on an NVIDIA Quadro P1000 with 4GB of GDDR5 with an Intel Xeon Silver 4114 CPU at 2.20 GHZ. The OpenACC code was compiled with version 19.4 of the PGI compiler using the CUDA 9.2 toolkit, also with the -O3 flag, and managed memory enabled. Memory transfer times between host and device are included in the timings.

5.5.2 Code Variants

The first code variant is a series of loop nests, each representing one `Proto` kernel. This version has the lowest arithmetic intensity per loop, but also the least amount of allocated memory. The fully fused version fuses all loop nests into one. This has the effect of maximizing arithmetic intensity, but also the quantity of live data. This variant also contains increased control flow due to the guards inserted to ensure that data dependences are satisfied. The added control flow limits SIMD vectorization. The third variant is partial fusion. Statement nodes are grouped using a greedy approach that increases arithmetic intensity while reducing memory traffic within each group. When the working set size for a group exceeds a given threshold, a new group is created. The threshold is experimentally derived, approximately based on the L3 cache size of the target architectures. This variant strikes a balance between AI and memory traffic.

Tiled versions of each variant are also generated. Tile sizes are set to 8 in each dimension, as 4 is too small for the applied stencils, and no performance benefit is observed at size 16. Performance results are given in Figure 5.4. Speedup is computed relative to the baseline execution time of the Euler implementation in `Proto`. The fully fused and tiled code variant is fastest as it maximizes arithmetic intensity, while reducing the active memory footprint to a single tile size (8^3) for each data space.

A scalability study was performed by sweeping the number of threads from 1 to 64 by powers of two on each of the three code variants for three different box sizes, small (N=16), medium (N=32), and large (N=64). The number of boxes

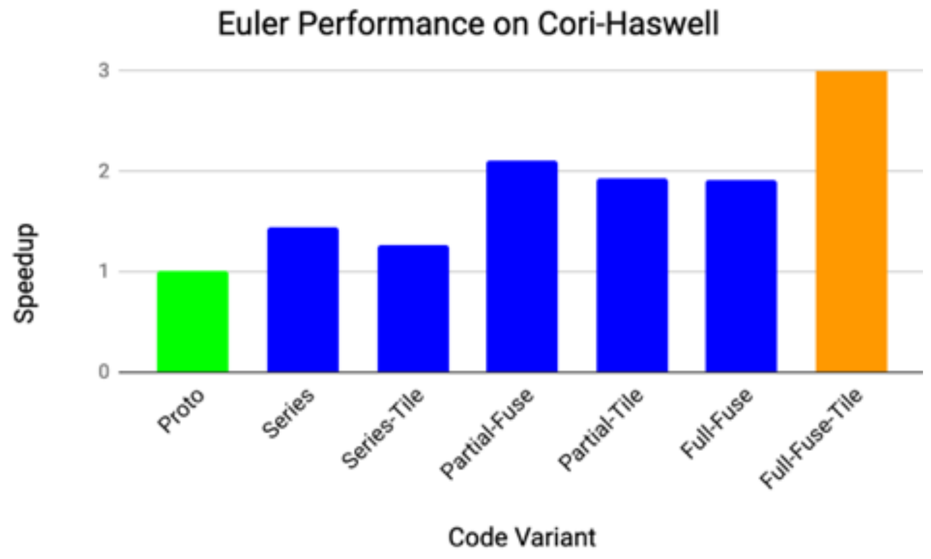


Figure 5.4: Performance results for the Euler step function on a Cori Haswell CPU node.

computed are set to ensure a constant number of cells (1536) for each run. The data are displayed in Figure 5.5. The `Proto` variant is excluded from these results as it does not implement OpenMP parallelism. The series of loops variant is used as the baseline. This variant is the fastest in all cases for small boxes. The fused and tiled variants do not outperform it until 16 threads for medium sized boxes, and 8 threads for large boxes.

The series of loops, partially fused, and fully fused variants were generated for the GPU with OpenACC pragmas. These variants are similar to those on the CPU, except component loops are interchanged and unrolled for vectorization. Figure 5.6 contains the performance results. Only the series of loops variant outperforms the original `Proto` implementation. The loop shifts required for fusion introduce additional control flow, probably causing thread divergence.

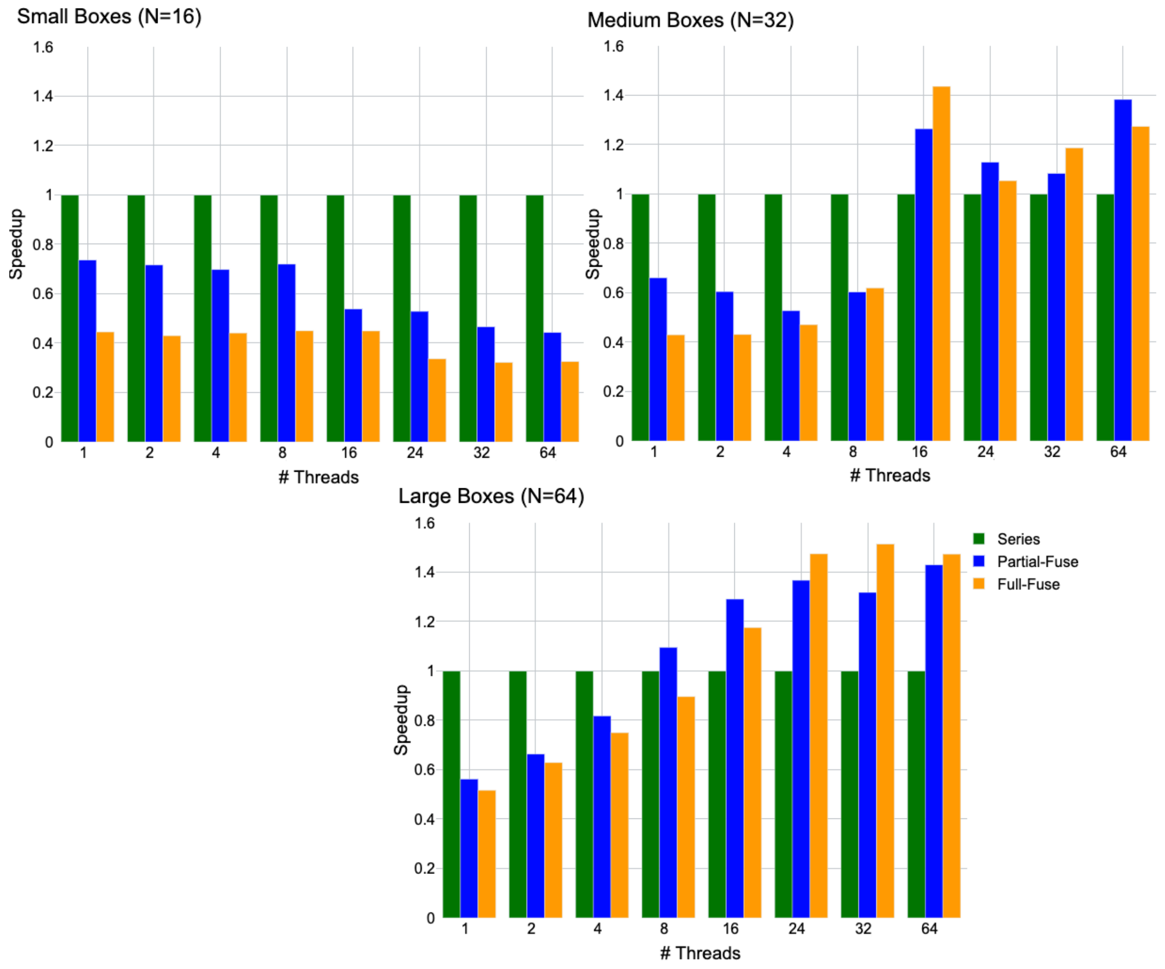


Figure 5.5: Comparison of code variants for each box size with thread sweep from 1 to 64.

5.6 Summary

This chapter presented `Proto`, an eDSL for structured grid PDE solvers, combined with the polyhedral+dataflow language (`PDFL`) to combine execution schedule transformations with dataflow optimizations. `Proto` statements and data are translated into the PDFG-IR and then optimized by applying a combination of loop fusion, tiling, parallelization, vectorization, and temporary storage reductions. A performance model including FLOP and memory throughput estimates is incor-

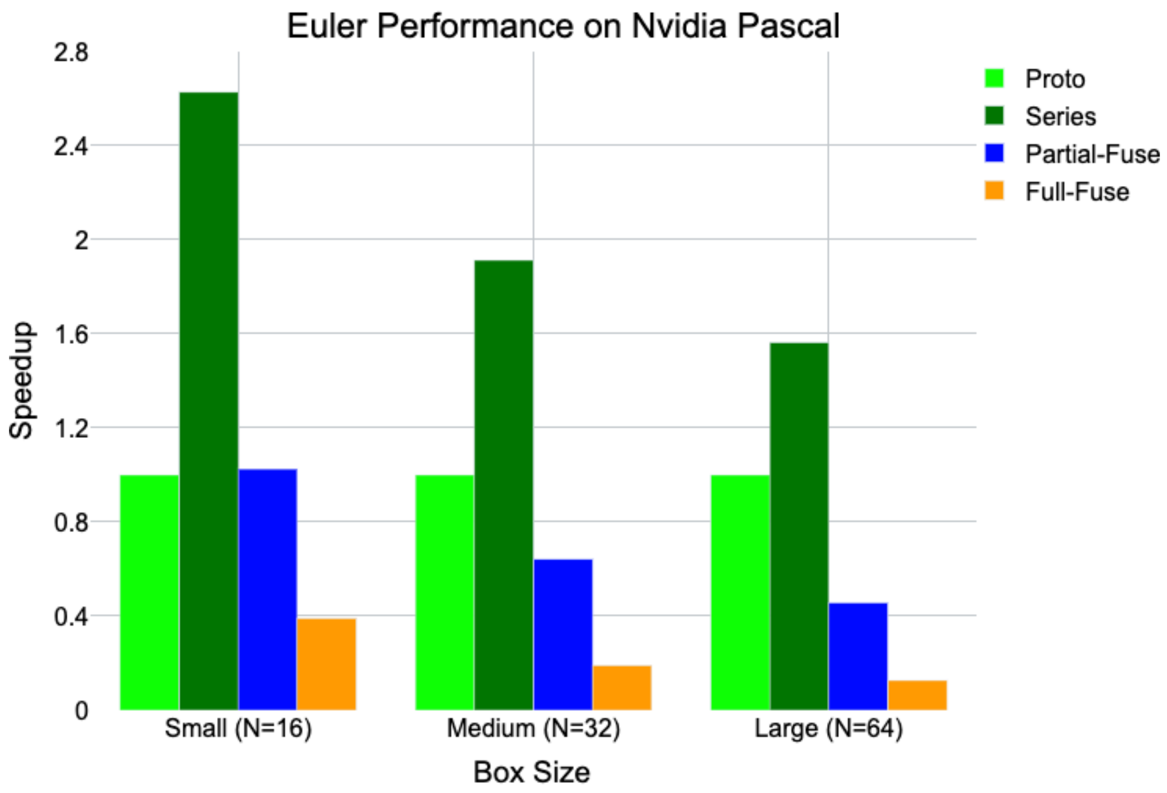


Figure 5.6: Performance results for the Euler step function on a NVIDIA Pascal GPU.

porated to automatically guide optimizations by maximizing arithmetic intensity while minimizing the working set size. Performance improvements of up to 3X were demonstrated with a CPU implementation of the Euler equations, and up to 2.6X for the GPU version.

CHAPTER 6

IRREGULAR ALGORITHM IMPLEMENTATIONS

This chapter demonstrates the effectiveness of the polyhedral+dataflow representation to express and transform numerical applications beyond small kernels or benchmarks. The two implementations include a conjugate-gradient solver for sparse matrices, and a canonical polyadic decomposition implementation for sparse tensors. Several different sparse formats are evaluated for each algorithm, and the corresponding inspectors and executors for each are defined for the primary kernels in both algorithms.

6.1 Conjugate Gradient

The sparse matrix-vector multiplication kernel is the computational bottleneck in the CG algorithm. The optimization space is explored by considering several sparse matrix storage formats. The study illustrates the effectiveness of the PDFG-IR for applications with limited opportunities for temporary storage reduction. The five storage formats evaluated include the default coordinate format (COO), compressed sparse row (CSR), doubly compressed sparse row (DSR), ELLPACK (ELL), and compressed sparse block (CSB). An overview of these sparse matrix formats is given in Section 2.6. A detailed description of the CG algorithm can be found in subsection 2.5.2, and in equations 2.8–7 specifically.

6.1.1 Executor Definitions

The sparse matrix-vector multiplication kernel is the target executor to be optimized in the CG algorithm. A general expression of the executor in PDFL for a dense matrix, A, is defined as follows:

$$\begin{aligned} spmv(i,j,n) = \{\, & 0 \le i < N \ \wedge\ 0 \le j < N \ \wedge\ n = i * N + j \,\} \ : \{ \\ & y(i) \mathrel{+}= A(n) * x(j) \,\}; \end{aligned} \tag{6.1}$$

The symbolic constant, N, is the number of rows and columns, M is the number of nonzeros, and (i,n,j) are the indices into the row vector y, matrix A, and column vector x, respectively. The SpMV kernels for the other sparse matrix formats are similarly defined in equations 6.2– 6.6, and applied as relations to transform the dense *spmv* definition.

$$coo(n,i,j) = \{\, 0 \le n < M \ \wedge\ i = row(n) \ \wedge\ j = col(n) \,\} \ * \ spmv; \tag{6.2}$$

$$\begin{aligned} csr(i,n,j) = \{\, & 0 \le i < N \ \wedge\ rp(i) \le n < rp(i+1) \ \wedge \\ & j = col(n) \,\} \ * \ spmv; \end{aligned} \tag{6.3}$$

$$\begin{aligned} dsr(m,i,n,j) = \{\, & 0 \le m < R \ \wedge\ i = crow(m) \ \wedge\ crp(m) \le n < \\ & crp(m+1) \ \wedge\ j = col(n) \,\} \ * \ spmv; \end{aligned} \tag{6.4}$$

$$\begin{aligned} ell(i,k,n,j) = \{\, & 0 \le i < N \ \wedge\ 0 \le k < K \ \wedge\ n = i * K + k \ \wedge \\ & j = lcol(i,k) \,\} \ * \ spmv; \end{aligned} \tag{6.5}$$

$$\begin{aligned} csb(b,n,i,j) = \{\, & 0 \le b < NB \ \wedge\ bp(b) \le n < bp(b+1) \ \wedge \\ & i = B * brow(b) + erow(n) \ \wedge \\ & j = B * bcol(b) + ecol(n) \,\} \ * \ spmv; \end{aligned} \tag{6.6}$$

6.1.2 Inspector Construction

The inspector transformations required to convert the executor for one matrix format into another can require the modification of iteration and/or data spaces. The matrix values In the ELL format, for example, are copied into a new 2D matrix, A_{ell}, of size $N \times K$, where K is the maximum number of nonzeros per row [97]. The compressed spare block matrix, A_{csb}, remains of size *M*, however the values are reordered to be visited in block order, e.g., Z-Morton ordering [120].

Inspectors generate the explicit functions or constants that satisfy the constraints containing the corresponding uninterpreted functions. The data spaces for the explicit functions are produced using constraints that are known in the source format, but unknown in the destination. The explicit functions are realized as data spaces by generating the code that satisfies the unknown constraints. The inspector code is generated from a combination of known constraints augmented with domain specific knowledge as needed.

An inspector that converts the COO format to CSR is given as an example. The two functions needed to satisfy the CSR executor constraints are the number of rows in the matrix, *N*, and the compressed row pointer, *rp*. The constraints needed to produce N are the known, $i{=}row(n)$, and the unsatisfied i <*N*. The code that produces N is generated by negating the unsatisfied constraint, i <*N*, to obtain $i \geq N$, the generating the statement that makes it true, $i = N + 1$. This process yields the first inspector component, $insp_N$, in equation 6.7,. The negated constraint is added to the iteration space as a guard condition, and bcomes an *if* statement in the generated code.

$$insp_N(n,i) = \{\, 0 \le n < M \ \wedge \ i = row(n) \ \wedge \ i \ge N\} \ : \ \{\, N = i + 1 \,\}; \tag{6.7}$$

$$insp_{rp_1}(n,i) = \{\, 0 \le n < M \ \wedge \ i = row(n) \,\} \ : \ \{\, rp(i+1) \mathrel{+}= \ 1 \,\}; \tag{6.8}$$

$$insp_{rp_2}(i) = \{\, 0 \le i < N \,\} \ : \ \{\, rp(i+1) \mathrel{+}= \ rp(i) \,\}; \tag{6.9}$$

The $insp_{rp1}$ component in equation 6.8 satisfies the requirement that each element of *rp* is a running count of the number of nonzeros in each row. The last component, $insp_{rp2}$ (6.9) ensures that the elements are monotonically non-decreasing,

The COO format does not enforce any particular ordering. Constraints solving to generate the given inspectors is simplified if the *row* function is sorted. The $insp_N$ inspector, for example, simplifies to the expression, $N = row(M-1)+1$. The sort introduces $O(MlogM)$ overhead, however this cost is amortized since the composed inspector will use it to generate other formats. The remaining inspector definitions assume the COO matrix has been sorted by row.

The doubly-compressed CSR format further compresses the rows data by removing any duplicate row values. The COO to DSR inspector is responsible for generating three unknown functions from the executor, the set of unique rows (*crow*), the compressed row pointer (*crp*), and the number of unique rows (R). The sorted COO *row* allows each function to be produced in one pass over the nonzero row indices .

$$insp_R(n) = \{\, 1 \le n < M \ \wedge \ row(n) \ \ne row(n-1) \,\} \ : \ \{\, R = R + 1 \,\}; \tag{6.10}$$

$$\begin{aligned} insp_{crow}(n) = \{\, 1 \le n < M \ \wedge \ row(n) \ \ne row(n-1) \,\} \ : \ \{ \\ crow(R) = row(n) \,\}; \end{aligned} \tag{6.11}$$

$$insp_{crp}(n) = \{\, 1 \le n < M \ \wedge \ n \ge crp(R+1) \,\} \ : \ \{\, crp(R) = n + 1 \,\}; \tag{6.12}$$

The COO to ELL inspector is derived from constraints that have already been satisfied in the CSR inspector. The K constant in the ELL format represents the maximum number of nonzeros per row in the matrix. The number of nonzeros for each row has already been captured in the rp function generated by the CSR inspector. Thus, the k value for any nonzero index, n, is the difference between itself and the nonzero count for that row, $k = n - rp(i)$. The ELL inspector is derived from the CSR inspector in equations 6.13–14.

$$insp_K(i) = \{\ 0 \le i < N\ \}\ :\ \{\ K = \mathbf{max}(K, rp(i+1) - rp(i))\ \}; \tag{6.13}$$

$$\begin{aligned} insp_{lcol,A_ell}(i,n,k) = \{\ & 0 \le i < N\ \wedge\ rp(i) \le n < rp(i+1)\ \wedge\ k = n - rp(i)\ \} \\ & :\ \{\ lcol(i,k) = col(n)\ \wedge\ A_{ell}(i,k) = A(n)\ \}; \end{aligned} \tag{6.14}$$

The compressed sparse block format for matrices can be generalized to the HiCOO format for tensors [33], so the CSB inspector derivation is covered by the HiCOO inspector described in Section 6.2.

The conjugate gradient algorithm from equations 2.8–7 is defined in the PDFL language below, where T is the maximum number of iterations, M the number of nonzeros in the sparse matrix A, and N is the number of rows and columns. The initial guess is assumed to be the zero vector, $x_0 = \vec{0}$. The *spmv* executor can be replaced with the corresponding kernel for the other matrix formats without modifying the remainder of the algorithm.

$$copy(i) = \{\ 0 \le i < N\ \} :\ \{\ r(i) = b(i),\ d(i) = r(i)\ \}; \tag{6.15}$$

$$spmv(t,n,i,j) = \{\ 1 \le t < T\ \wedge\ 0 \le n < M\ \wedge\ i = row(n) \wedge \tag{6.16}$$

$$j = col(n)\ \}\ :\{\ s(i) \mathrel{+}= A(n) * d(j)\ \}; \tag{6.17}$$

$$ddot(t,i) = \{\ 1 \le t < T\ \wedge\ 0 \le i < N\ \} :\ \{\ ds \mathrel{+}= d(i) * s(i)\ \}; \tag{6.18}$$

$$rdot0(t,i) = \{\ 1 \le t < T\ \wedge\ 0 \le i < N\ \} : \ \{\ rs0 \mathrel{+}= r(i) * r(i)\ \}; \tag{6.19}$$

$$alpha(t) = \{\ 1 \le t < T\ \} : \ \{\ \alpha = rs0\ /\ ds\ \}; \tag{6.20}$$

$$xadd(t,i) = \{\ 1 \le t < T\ \wedge\ 0 \le i < N\ \} : \ \{\ x(i) \mathrel{+}= \alpha * d(i)\ \}; \tag{6.21}$$

$$rsub(t,i) = \{\ 1 \le t < T\ \wedge\ 0 \le i < N\ \} : \ \{\ r(i) \mathrel{-}= \alpha * s(i)\ \}; \tag{6.22}$$

$$rdot(t,i) = \{\ 1 \le t < T\ \wedge\ 0 \le i < N\ \} : \ \{\ rs \mathrel{+}= r(i) * r(i)\ \}; \tag{6.23}$$

$$beta(t) = \{\ 1 \le t < T\ \wedge\ 0 \le i < N\ \} : \ \{\ \beta = rs\ /\ rs0\ \}; \tag{6.24}$$

$$dadd(t,i) = \{\ 1 \le t < T\ \wedge\ 0 \le i < N\ \} : \ \{\ d(i) = \beta * d(i) + r(i)\ \}; \tag{6.25}$$

6.1.3 Code Generation

The inspectors are first composed into one dataflow graph, then optimized by performing loop fusion and parallelization. Dynamic arrays allow loops that produce data spaces with sizes unknown at compile time to be fused with the loops that produce the final sizes. The *crow* space from the DSR inspector and the *brow* space from CSB, for example, have worst-case size *M*. These spaces are resized once the actual sizes are known, *R* and *NB*, respectively. The source code for the composed inspectors is given in Figure 6.1. The memory allocations statements and *bnum* function that generates block numbers are omitted for brevity.

A separate executor is generated for each matrix format since the SpMV kernel is diferent for each. The optimized code for the CG algorithm using the CSR version of the SpMV executor is displayed in Figure 6.2. The storage space for the **d**, **r**, and **s** vectors in the CG algorithm cannot be further reduced so the temporary storage reductions are due to the different matrix formats.

```
N = row(M-1) + 1;
rp(row(0)) += 1;

#pragma omp simd
for(n = 1; n < M; n++) {
  i = row(n);
  rp[i+1] += 1;

  if (i != row[n-1]) R+=1;
  crow[R] = i;
  if (n >= crp[R+1]) crp[R+1] = n+1;

  bi = row[n]/B;
  bj = col[n]/B;
  b = bnum(bi,bj);
  bmap[b][bcnt[b]++] = n;
  if (b >= NB) NB = b+1;
  brow[b] = bi;
  bcol[b] = bj;
}
R += 1;

for(i = 0; i < N; i++) {
  K = max(K, rp(i+1));
  rp[i+1] += rp[i];
}

#pragma omp parallel for
for(i = 0; i < N; i++) {
  #pragma omp simd
  for (n = rp(i); n < rp(i+1); i++) {
    k = n - rp(i);
    lcol[i*K+k] = col[n];
    lval[i*K+k] = val[n];
} }

#pragma omp parallel for
for (b = 0; b < nb; b++) {
  #pragma omp simd
  for (p = 0; p < bcnt[b]; p++) {
    n = bmap[b][p];
    erow[p] = row[n] - B * brow[b];
    ecol[p] = col]n] - B * bcol[b];
    if (p >= bp[b+1]) bp[b+1] = p+1;
    bval[p] = val[n];
} }
```

(a)

Figure 6.1: Optimized code for the composed inspectors to convert COO matrices into the CSR, DSR, ELL, and CSB formats.

```
#pragma omp parallel for
for(i = 0; i < N; i++)
 r[i] = d[i] = b[i];
for(t = 1; t <= T; t++) {
 ds = rs0 = rs = 0.0;
 #pragma omp parallel for
 for(i = 0; i < N; i++) {
  s[i] = 0.0;
  #pragma omp simd
  for(n = rp[i]; n < rp[i+1]; n++)
   s[i] += A[n] * d[col[n]];
  ds += d[i] * s[i];
  rs0 += r[i] * r[i];
 }
 alpha = rs0 / ds;
 #pragma omp parallel for
 for(i = 0; i < N; i++) {
  x[i] += alpha * d[i];
  r[i] -= alpha * s[i];
  rs += r[i] * r[i];
 }
 beta = rs / rs0;
 #pragma omp parallel for
 for(i = 0; i < N; i++) {
  d[i] = r[i] + beta * d[i];
} }
```

(a)

Figure 6.2: Optimized code for the conjugate gradient algorithm with SpMV executor of the CSR format.

6.2 Tensor Decomposition

The CPD algorithm follows a similar pattern as CG. The decomposition steps are applied for a maximum number of iterations, T, or until the error threshold is reached. The bottleneck kernel in CPD is the matricized tensor times Khatri Rhao product (MTTKRP). This section will consider three sparse tensor formats, coordinate (COO), compressed sparse fiber (CSF) [98], and hierarchical coordinate (HiCOO) [33]. The CPD algorithm is given in subsection 2.5.5.

The COO format is generalized for tensors by replacing the *row* and *col* functions with a 2D function, *index*(n,m) where n is the dimension ($0 \leq n < N$), and m is the nonzero position ($0 \leq m < M$). The MTTKRP kernel for an Nth order tensor, $\mathcal{X}$, in COO format is expressed in PDFL as follows:

$$\begin{aligned} krp(n,m,r,p,i) = \{\, & 0 \le n < N \ \wedge\ 0 \le m < M \ \wedge\ 0 \le r < R \wedge \\ & 0 \le p < N \ \wedge\ n \ne p \ \wedge\ i = index(p,m) \,\} \ : \{ \\ & A(n,i,r) \mathrel{+}= \ X(m) * A(p,i,r) \ \}; \end{aligned} \tag{6.26}$$

where M is the nonzero count, R is the rank, and A represents the factor matrices, $A_1, \ldots, A_N$.

The HiCOO format is a generalization of CSB, where *brow* and *bcol* are replaced with the 2D function, *bind*(n,b), where b is the block number. The *erow*, *ecol* functions are replaced with *eind*(n,m), where m is the index of each nonzero value. The *bp*(b) function contains pointers to the nonzero indices for each block, and B is a parameter indicating the block size. Block sizes can be varied per dimension by replacing B with a function, *bsize*(n), however in this case, the same B will be applied to all dimensions. The MTTKRP kernel for the HiCOO storage format is defined below.

$$\begin{aligned} hicoo(n,b,m,r,p,i) = \{\, & 0 \le n < N \ \wedge\ 0 \le b < NB \ \wedge\ bp(b) \le m < \\ & bp(b+1) \ \wedge\ 0 \le r < R \ \wedge\ 0 \le p < N \wedge \\ & n \ne p \ \wedge\ i = B * bind(b,p) + eind(m,p) \,\} \ * krp; \end{aligned} \tag{6.27}$$

The CSF format can be considered a generalization of DSR for matrices, with the compressed row pointer, (*crp*), replaced by function *coff*(n,q) representing compressed offsets, where q is the nonzero index pointer at each level, and *cind*(n,q) contains the compressed indices. Each compressed fiber is represented as a tree structure, as seen in Figure 2.8. Recursive definitions are not supported in the polyhedral model, so guard statements are added to the executor definitions, to support third or higher order tensors. The modes are compressed in ascending order, e.g. row-column-tube order (0,1,2) for a third order tensor.

$$\begin{aligned}
& csf(n,f,i,g,j,h,k,m,l,r,p) = \{ \\
& \quad coff(0,0) \leq f < coff(0,1) \ \wedge \ i = cind(0,f) \wedge \\
& \quad coff(1,f) \ \leq \ g \ < \ coff(1,f+1) \ \wedge \ j = cind(1,g) \wedge \\
& \quad coff(2,g) \ \leq \ h \ < \ coff(2,g+1) \ \wedge \ k = cind(2,h) \ \wedge \ N > 3 \wedge \\
& \quad coff(3,h) \ \leq \ m \ < \ coff(3,h+1) \ \wedge \ l = cind(3,m) \wedge \\
& \quad 0 \leq r < R \ \} * krp;
\end{aligned} \tag{6.28}$$

6.2.1 Inspector Construction

The COO to HiCOO inspector requires two passes. The first pass produces the *bind* function, the number of blocks, *NB*, and three intermediate spaces, *bnum* stores the block numbers, *bcnt*, contains the running count of nonzeros per block, and *bmap*, maps the block order of the nonzeros to the original COO order. The second pass generates the *bp* and *eind* functions.

$$\begin{aligned}
& crd(m,n) = \{ \ 0 \leq m < M \ \wedge \ 0 \leq n < N \ \} \ : \ \{ \\
& \quad bcrd(n) = \lfloor index(n,m)/B \rfloor \ \};
\end{aligned} \tag{6.29}$$

$$\begin{aligned}
& num(m,n,b) \ = \ \{ \ b = bnum(bcrd) \ \} \ : \ \{ \\
& \quad bmap(b, bcnt(b)) = n \ \wedge \ bcnt(b) \mathrel{+}= 1 \ \};
\end{aligned} \tag{6.30}$$

$$\tag{6.31}$$

$$cnt(n,b) \;=\; \{\, b \geq NB \,\} : \{\, NB = b + 1 \,\}; \tag{6.32}$$

$$ind(m,n,b) \;=\; \{\, 0 \leq n < N \,\} \;:\; \{\, bind(n,b) = bcrd(n) \,\}; \tag{6.33}$$

$$bptr(b,p,m) = \{\, 0 \leq b < NB \;\wedge\; 0 \leq p < bcnt(b) \;\wedge$$

$$m = bmap(b,p) \;\wedge\; bp(b+1) \leq p \,\} \;:\{\, bp(b+1) = p + 1 \,\}; \tag{6.34}$$

$$eind(b,p,m,n) = \{\, 0 \leq n < N \,\} \;:\{\, eind(n,p) =$$

$$index(n,m) - B * bind(n,b) \,\}; \tag{6.35}$$

$$cpy(b,p,m,n) = \{\, 0 \leq n < N \,\} \;:\{\, X_{cpy}(p) = X(m) \,\}; \tag{6.36}$$

The COO to CSF inspector is described in Chapter 4, and the specification is given in Figure 4.13.

6.2.2 CP-ALS Implementation

The PDFL representation of the CP-ALS algorithm is given in equations 6.37–44. The factor matrices, A_1, $\ldots$, A_N are initialized to random values with the *urand* function, that samples a uniform distribution of values between zero and one. The MTTKRP kernel updates the factor matrix for the current dimension, n, by multiplying it with the tensor value at each other dimension, p. Each factor matrix is multiplied by its transpose to compute $\mathbf{A_i}^\top\mathbf{A_i}$. The resulting data space, A_{mTm}, consists of N, $R \times R$ matrices. The component-wise Hadamard product is then applied to each matrix product to obtain the matrix, V.

The Moore-Penrose pseudoinverse, V_{inv}, of V, is computed with the *pinv* function. This function is implemented using the singular value decomposition (SVD) of the matrix, $A = U\Sigma V^T$, and discarding the singular values of Σ that are below a certain threshold [121]. The *pinv* implementation in PDFL uses the SVD algorithm from the GNU scientific library (GSL) [122], with a singular theshold of 1×10^{-15}. The Froebenius norm is calculated and stored in the λ vector, and the factor matrices are normalized. CP-ALS implementations typically return the results as a Kruskal tensor, that consists of the factor matrices stored in a tensor, $\mathcal{U}$, and the normalization vector, λ.

$$\begin{aligned}
&init(n,i,r) = \{\, 0 \le i < N \ \wedge\ 0 \le i < dim(n) \ \wedge\ 0 \le r < R \,\} :\ \{ \\
&\quad A(n,i,r) \ = urand()\ \}; && (6.37) \\
&krp(t,n,m,r,p,i) = \{\, 0 \le n < N \ \wedge\ 0 \le m < M \ \wedge \\
&\quad 0 \le r < R \ \wedge\ 0 \le p < N \ \wedge\ n \ne p \ \wedge\ i = index(p,m) \,\} \ : \{ \\
&\quad A(n,i,r) \mathrel{+}= \ X(m) * A(p,i,r)\ \}; && (6.38) \\
&m\mathrm{T}m(t,n,q,r,i) = \{\, 1 \le t \le T \ \wedge\ 0 \le q < R \ \wedge\ 0 \le r < R \ \wedge \\
&\quad 0 \le i < dim(n) \,\} :\ \{\ A_{mTm}(n,q,r) \mathrel{+}= A(n,q,i) * A(n,i,r)\ \}; && (6.39) \\
&had(t,n,p,q,r) = \{\, 0 \le p < N \ n \ne p \ \wedge\ 0 \le q < R \ \wedge \\
&\quad 0 \le r < R \,\} :\ \{\ V(q,r) \mathrel{*}= \ A_{mTm}(p,q,r)\ \}; && (6.40) \\
&pinv(t,n) = \{\, 1 \le t \le T \ \wedge\ 0 \le n < N \,\} :\ \{\ V_{inv} \ = pinv(V)\ \}; && (6.41) \\
&mmp(t,n,i,q,r) = \{\, 0 \le i < dim(n) \ \wedge\ 0 \le q < R \ \wedge\ 0 \le r < R \,\} \ :\ \{ \\
&\quad A(n,i,q) \mathrel{+}= \ A(n,i,r) * V_{inv}(r,q)\ \}; && (6.42) \\
&ssq(t,n,i,r) = \{\, 0 \le i < dim(n) \ \wedge\ 0 \le r < R \,\} \ :\ \{ \\
&\quad \sigma(r) \mathrel{+}= \ A(n,i,r) * A(n,i,r)\ \}; && (6.43) \\
&norm(t,n,r) = \{\, 0 \le r < R \,\} \ :\ \{\ \lambda(r) \ = \ sqrt(\sigma(r))\ \}; && (6.44) \\
&div(t,n,i,r) = \{\, 0 \le i < dim(n) \ \wedge\ 0 \le r < R \,\} \ :\ \{ \\
&\quad A(n,i,r) \mathrel{/}= \ \lambda(r)\ \}; && (6.45)
\end{aligned}$$

6.2.3 Code Generation

The CPD code was generated by fusing all possible loops while applying storage reductions. Two additional optimizations are manually applied. A coordinate buffer, *crd*, of size N is introduced to store the coordinates of each nonzero value. This allows the sparse index structures to be traversed once per value, for each iteration of the dimension loop. The MTTKRP kernel can then be the same for each data format, and only the index traversal code needs to be updated.

The next optimization is the introduction of a workspace to store the results when multiplying the factor matrices by the pseudoinverse matrix, $V^{\dagger}$. This reduces the temporary storage required for this step by only storing enough space for the factor matrix with the maximum dimension, D. The matrix times transpose multiplication results, $A_i^{\top} A_i$, used to compute the Hadamard product, V, could be reduced to one $R \times R$ matrix. However, this reduction is not performed since it would inhibit parallelism by introducing a race condition. The optimized code for the CPD algorithm using the HiCOO version of the MTTKRP executor is displayed in Figure 6.3.

6.3 Experimental Evaluation

The experimental evaluation is performed using a variety of sparse linear algebra formats. The CG algorithm is evaluated by exchanging the sparse matrix-vector multiplication kernel, using several matrix formats including COO, CSR, DSR, ELL, and CSB with block size B=128.

The CPU results were collected on a single node of an Intel Xeon E5-2680 v4 machine at 2.40 GHz clock frequency with 28 cores, 14 per socket. The cores include a 32KB L1, 256KB L2, and 35840K L3 caches. The system contains 192GB of RAM split over 2 NUMA domains. The GCC 7.2 compiler with -O3 flag was used. The GPU data were collected on an NVIDIA Quadro P1000 with 4GB of GDDR5 with an Intel Xeon Silver 4114 CPU at 2.20 GHZ. The OpenACC code was compiled with version 19.4 of the PGI compiler using the CUDA 9.2 toolkit, with the -O3 flag, and managed memory enabled. Memory transfer times between host and device are included in the timing results.

The performance of the CG algorithm executed for 500 iterations with 28 threads on each matrix format is displayed in Figure 6.4. The data were collected on twelve sparse matrices from the SuiteSparse repository, the names are along the x-axis. Speedup is reported on the y-axis, with higher values indicating better performance. Each experiment was conducted nine times with the mean value reported. The baseline implementation is from the Eigen high-performance library [123], which stores matrices in a version of CSR.

Table 6.1 contains a summary of the matrix formats evaluated, the parameterized sizes, mean sizes, and mean speedups. The size parameters are the number of rows and columns, N, the number of nonzeros, M, the number of nonzero rows, R (DSR), the maximum number of nonzeros per row, K (ELL), the number of nonzero blocks, NB (CSB), the size of an integer, I, and the size of a floating point value, F. The third column is the mean size in MB, and the last is speedup relative to Eigen.

The table data indicate an inverse relationship between the storage size and the speedup. Reduced storage size correlates with improved performance when all of the CPU cores are active, with the exception of CSB. This is possibly due to the additonal overhead introduced by the need to compute the dense indices, (i, j), at each iteration. CSB outperforms CSR for some matrices, e.g., *webbase-1M*,. This illustrates the importance of considering the input data structure. The poor performance of the ELL format on the CPU is expected, since it was developed for large SIMD architectures [124].

Reduced storage is not the most significant performance predictor on the GPU platform. Executors with reduced loop overhead and control flow, with predictable data access patterns exhibit superior performance. The GPU results are displayed in Figure 6.5. The Eigen library does not support GPUs, so the various formats are plotted with speedup relative to the COO execution time reported on the y-axis. The COO variant is the fastest overall, with a mean of 0.66 seconds, and CSR is next with a mean of 0.78 seconds. The DSR format is omitted because it is significantly slower than the other formats.

Table 6.1: Summary of matrix formats, expected sizes, mean sizes, and corresponding experimental mean speedups on Intel Xeon CPU.

Format	Expected Size	Mean Size (MB)	Mean Speedup
COO	$2 \times M \times I + M \times F$	45.7	2.68
CSR	$(M + N + 1) \times I + M \times F$	35.3	5.07
DSR	$(M + 2 \times R + 1) \times I + M \times F$	36.2	4.42
ELL	$(N \times K) \times (I + F)$	168	0.69
CSB	$(3 \times NB + 1) \times I + M \times (F + 1)$	28.9	3.45

The CPD algorithm is evaluated with the COO, CSF, and HiCOO sparse tensor formats applied to the MTTKRP kernel . Each tensor was decomposed into factor matrices of rank, R=10. The CPU results are given in Figure 6.6. Each experiment was performed five times with the mean value reported. The SPLATT [31] tensor library is the baseline for both execution times and data verification. The SPLATT implementation outperforms the PDFL version for all but one tensor, *flickr3d*. The data representation in the SPLATT CPD algorithm is CSF, which is the fastest format for the PDFL implementation as well. SPLATT is a high performance library, that calls the highly tuned linear algebra functions in the LAPACK library [125] for many of the matrix operations in the algorithm.

6.4 Summary

This chapter described the implementation of two numerical algorithms in the polyhedral+dataflow language, conjugate gradient for sparse matrices, and canonical polyadic decomposition for sparse tensors. The ability to construct inspectors for different sparse data formats and compose them was demonstrated. The transformed executors for computationally intensive kernels, e.g., SpMV or MTTKRP, can be exchanged in the dataflow graph representation without altering the remainder of the algorithm. Performance results indicate that the format with the greatest storage reduction yields the best speedup on the CPU. The GPU results indicate that reduced loop overhead and control flow are more important performance factors.

```
for(n = 0; n < N; n++) {
 D = max(D, dim[n]);
 #pragma omp parallel for
 for(i = 0; i < dim[n]; i++) {
  for(r = 0; r < R; r++) {
   A[n][i*R+r] = urand();
   #pragma omp simd
   for(q = 0; q < R; q++) {
    AtA[n][r*R+q] += A[n][r*dim[n]+i]*A[n][i*R+q];
} } } }
ws = (float*) calloc(D*R, sizeof(float));
for(t = 1; t <= T; t++) {
 for(n = 0; n < N; n++) {
  #pragma omp parallel for
  for (b = 0; b < NB; b++) {
   for(m = bp[b]; m < bp[b+1]; m++) {
    #pragma omp simd
    for (p = 0; p < N; p++)
     crd[p] = B*bind[b*N+p] + eind[m*N+p];
    for (r = 0; r < R; r++) {
     prod = 1.0;
     #pragma omp simd
     for (p = 0; p < n; p++)
      prod *= A[p][crd[p]*R+r];
     #pragma omp simd
     for (p = n+1; p < N; p++)
      prod *= A[p][crd[p]*R+r];
     A[n][crd[n]*R+r] += X[m] * prod;
  } } }
  #pragma omp parallel for
  for(q = 0; q < R; q++) {
   for(r = 0; r < R; r++) {
    V[q*R+r] = 1.0;
    #pragma omp simd
    for(p = 0; p < n; p++)
     V[q*R+r] *= AtA[p][r*R+q];
    #pragma omp simd
    for(p = n+1; p < N; p++)
     V[q*R+r] *= AtA[p][r*R+q];
  } }
  pinv(V,Vinv);
  #pragma omp parallel for
  for(i = 0; i < dim[n]; i++) {
   for(q = 0; q < R; q++) {
    #pragma omp simd
    for(r = 0; r < R; r++) {
     ws[i*R+q] += A[n][i*R+r] * Vinv[q*R+r];
  } } }
  #pragma omp parallel for
  for(i = 0; i < dim[n]; i++) {
   #pragma omp simd
   for(r = 0; r < R; r++) {
    A[n][i*R+r] = ws[i*R+r];
    sums[r] += ws[i*R+r] * ws[i*R+r];
    ws[i*R+r] = 0.0;
  } }
  #pragma omp parallel for
  for(t6 = 0; t6 <= R-1; t6++) {
   lmbda[r] = sqrt(sums[r]);
   sums[r] = 0.0;
   #pragma omp simd
   for(i = 0; i < dim[n]; i++) {
    A[n][i*R+r] /= lmbda[r];
  } }
  #pragma omp parallel for
  for(q = 0; q < R; q++) {
   for(r = 0; r < R; r++) {
    AtA[n][q*R+r] = 0.0;
    #pragma omp simd
    for(i = 0; i < dim[n]; i++) {
     AtA[n][q*R+r] += A[n][q*dim[n]+i] * A[n][i*R+r];
} } } } }
```

(a)

Figure 6.3: Optimized code for the CPD algorithm with HiCOO variant of the MTTKRP executor.

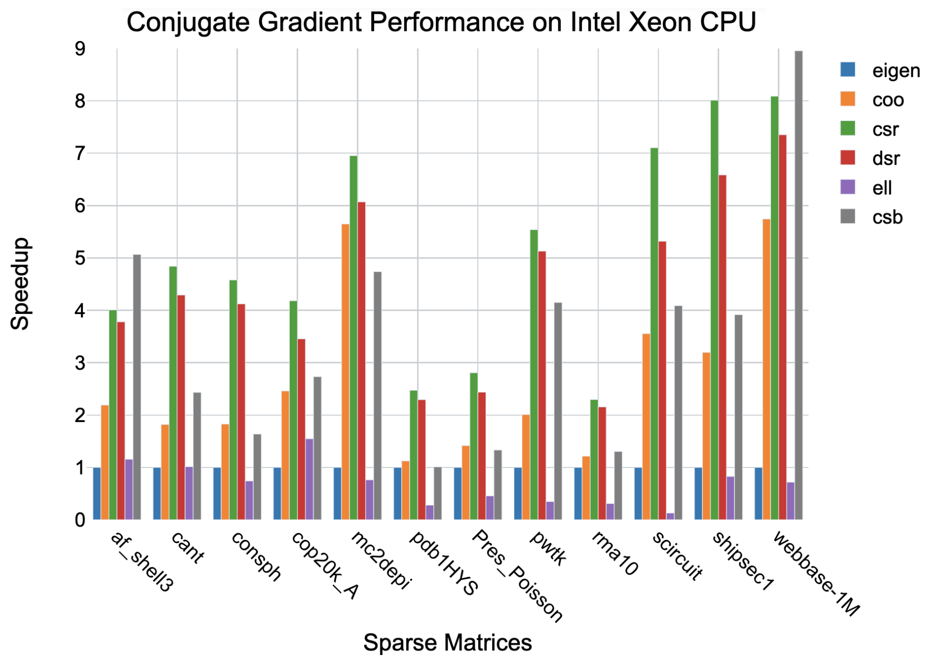


Figure 6.4: Performance results for the Conjugate Gradient algorithm on an Intel Xeon CPU.

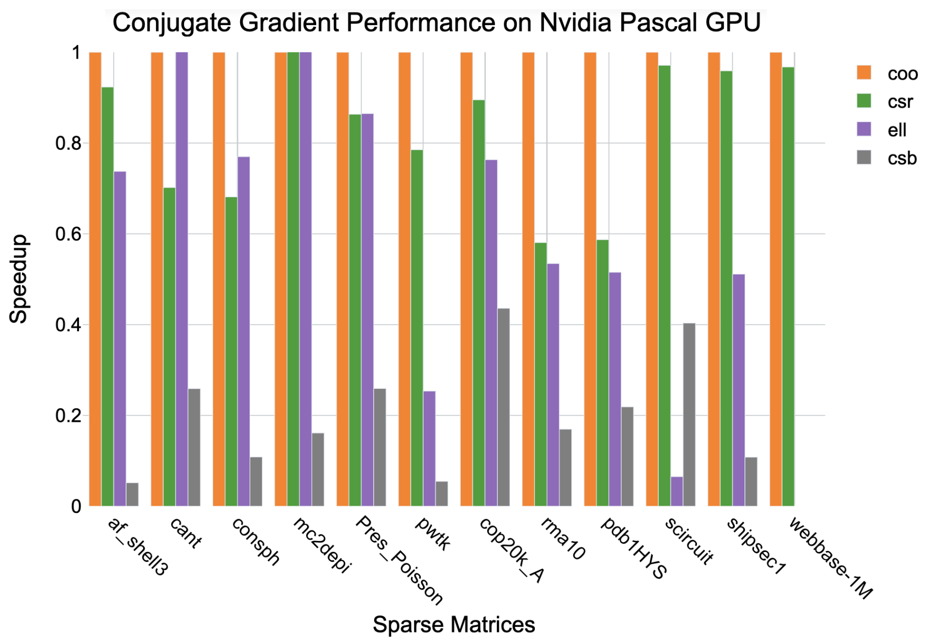


Figure 6.5: Performance results for the Conjugate Gradient algorithm on an Nvidia Pascal GPU.

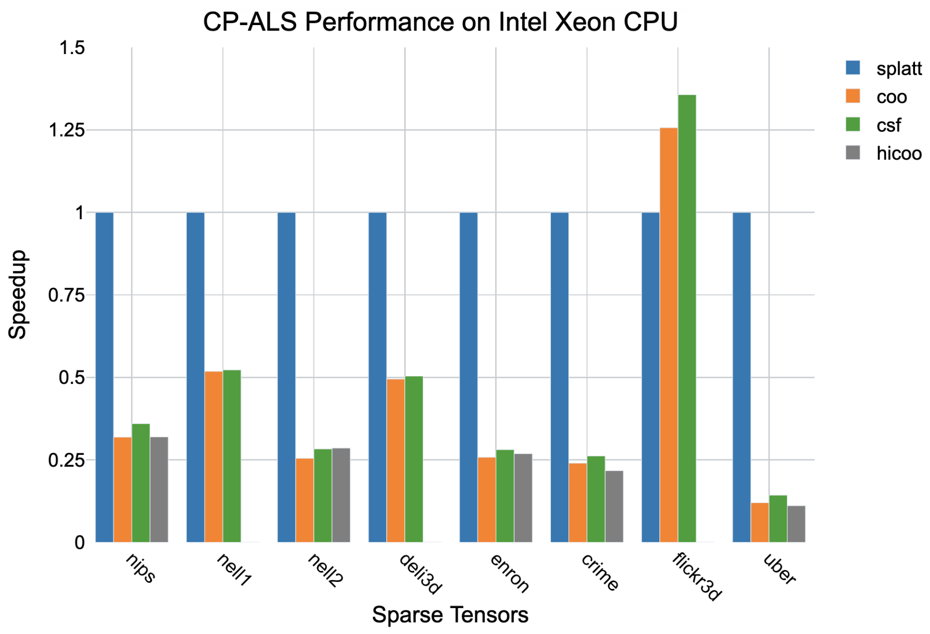


Figure 6.6: Performance results for the CP-ALS algorithm on an Intel Xeon CPU.

CHAPTER 7

RELATED WORK

This chapter dicusses prior work in this research area with a focus on programming and scripting languages, intermediate representations, polyhedral and tensor compilers, library-based approaches, performance modeling, and autotuning.

7.1 Programming Languages

Programming languages are the primary means of expressing computation in software. There are several paradigms, including imperative, functional, and object-oriented programming. Languages must be intuitive for the programmer, while communicating necessary information to the compiler. Significant investments have been made in existing applications to maitain compatibility.

Frontend approaches fall into several categories. One method is to annotate existing source code with pragmas (e.g, `#pragma` in C). Examples include automatic parallelization with OpenMP [126], empirical performance tuning using Orio [127], loop nest optimization with the loop chain abstraction [13], polyhedral transformations with PET [36], or building dataflow representations in DFGR [128]. These solutions are often implemented as source-to-source translators, manipulating the AST to convert one source level representation into another.

Domain specific languages (DSLs) target a particular problem space. DSLs provide a separation of concerns between the primary algorithmic expression and the underlying implementation, including execution schedules and storage mappings. Constructing an entirely new compiler is a considerable software engineering challenge, so DSLs are often embedded within existing language such as C++ or Python (eDSLs).

Halide [4, 56, 129] is an eDSL targeting image processing pipelines implemented as streams of stencil operations. It is a functional language embedded in C++ implemented as a library. Halide provides a systematic model of the tradeoffs between data locality, parallelism, and redundant computation. The Halide IR is a DAG representation, but is not directly accessible by the user. Halide uses interval analysis, which does not offer the precision or flexibility of the polyhedral model.

PolyMage [5, 130] is another eDSL developed to target image processing applications with a focus on decoupling algorithms from execution schedules. Algorithms are specified by a set of functional constructs and converted into an intermediate representation called a stage graph. The compiler traverses the graph from bottom-up and performs static bounds checking, function inlining, and live output analysis. A polyhedral representation of the graph is constructed from the derived loop bounds. Various polyhedral transformations can be applied to improve parallelism, and increase data locality, including parallelogram, split, and overlapped forms of tiling Automatic parallelization [56] was added, and a dynamic programming based performance modeling approach [57]. PolyMage does not directly support irregular applications, or code generation for GPU platforms.

PENCIL [131] is a polyhedral DSL that supports dynamic data-dependent control flow and array accesses. The DSL is C99 subset that is compiled into OpenCL [132] code that is optimized by iterative autotuning. PENCIL has restrictions to allow these transformations, omitting pointer arithmetic, recursion, and dynamically sized arrays, while requiring well-formed, structured `for` loops. Array accesses should not be linearized, as subscript information is used to build a polyhedral representation. The compiler is a modified version of `PPCG` [44], that is in turn built upon PET [36]. PENCIL does not include a performance model, a visual feedback mechanism, nor does it support non-affine polyhedral transformations.

PetaBricks [133] is a language that allows the programmer to specify multiple versions of an algorithm, along with rules to define the computation and define explicit producer-consumer relationships or data dependences. The rules may be defined at multiple granularities and corner cases. The compiler applies the rules to generate hybrid algorithms, and uses autotuning to explore the transformation space, including execution schedule, tiling scheme, and parallelism strategy. This work differs in that the user only supplies one version of an algorithm, and can optionally specify the desired transformations.

Firedrake [134] identifies four different expert roles in the development of a scientific application and provides unique interfaces for each. This reduces the breadth of expert knowledge required and results in application code that is more maintainable and portable between compute resources. The scheduling work presented here is applicable to the parallel programming expert interface in Firedrake.

7.2 Scripting Languages

Transformation scripts are an alternative to augmenting code with pragmas. The scripts describe transformations to be applied by the compiler. This approach is taken by the CHiLL [50], URUK [135], POET [136], and AlphaZ [137] frameworks. AlphaZ [137] is a polyhedral framework for exploring code transformations with support for memory remapping and simplifying reductions.

CHiLL is a loop transformation framework that uses the Omega+/CodeGen+ [11] polyhedral compiler for code generation. The algorithm derives alternative code variants from the input source code and accompanying transformation script that describes the target optimizations. A search engine locates opportunities to apply those transformations. The code variants are generated, compiled and executed to determine the best performing version.

High-order stencil optimizations were implemented in CHiLL using array common subexpression elimination with partial sums [25]. The compiler Common floating point operations across loop iterations are identified and reused. Redundant computations are reduced at the cost of increased storage. The compiler constructs an array of coefficients for the partial sum transformation. The partial sum optimization applies to out-of-place, constant-coefficient stencils.

CHiLL has been applied to compiler-directed autotuning in geometric multigrid applications [138]. Extensions include the *superscript*, a parameterized template for high level loop transformations. Superscripts are consumed by a script generator that produces transformations recipes for the existing CHiLL compiler. The generator creates OpenTuner [139] ensembles for the autotuner. The superscript represents the possible transformations, and each generated script a point in the search space.

7.3 Intermediate Representations

A compiler framework is an infrastructure to help developers create analysis tools or source-to-source translators, such as ROSE [140] and LLVM [55]. Cetus [141] is a compiler framework used to implement hierarchical overlapped tiling [101]. LLVM is language agnostic IR in SSA form designed as an intermediate representation for the Clang C/C++ compiler toolchain. Frontends have been developed for many languages, including Fortran [142], R [143], Python [144], and Julia [145]. PLuTo [10] is a fully automated, source-to-source polyhedral compiler that uses ClooG [46] for code generation. The Omega+ [11] is the polyhedral compiler used by CHiLL, and in this work. Polly [53] is a polyhedral compiler for the LLVM IR.

Compile time transformations often target the IR level. Dataflow graphs are constructed by compilers at the statement or instruction level. Macro dataflow graphs [60] allow similar analyses but coarsen granularity to the function, basic block, or loop nest level. Dataflow partitioning was implemented in the SISAL [146] and VAL [147] single assignment languages. Functional representations of an application are easily translated into macro dataflow graphs.

Dataflow representations can be exploited to find parallelism and inform the associated code generation [61]. Nodes within these graphs can be coalesced, combining lightweight nodes into fewer heavyweight nodes, thereby reducing communication. Prasanna et al. [62] take a hierarchical approach. Each macro node is scheduled for parallel execution on a node, unlike the previous work that assume serial execution. The entire graph is then partitioned and scheduled for distributed memory execution.

PolyAST [45, 148] presents an optimization flow that combines the polyhedral model with AST transformations to improve parallelism, but does not specifically target dataflow optimizations. DFGR [128] provides an implementation of macro-dataflow graphs in Habanero-C [149], a concurrent version of C that is built on the Intel Concurrent Collections (CnC) [150], and uses thread building blocks (TBB). The Data-Flow Graph Language (DFGL) [17] is an optimization framework based on DFGR that allows graph based dependences to be represented in the polyhedral model. TIDeFlow [16] is an execution model specifying the precedence of computations without concern for scheduling or synchronization. It differs from other dataflow models by introducing the use of transitions and places from Petri nets to determine node weights.

Tapir [151] extends the control flow graph representation of LLVM IR with *detach*, *reattach*, and *sync* primitives to enable the fork-join parallelism of OpenMP [126], Cilk [152], or the Habanero family of concurrent laguages [149]. The Heterogeneous Parallel Virtual Machine (HPVM) [153] implements a dataflow-based, shared memory IR, with a virtual instruction set (ISA) that can represent code for multiple target architectures, and a runtime scheduler. HPVM differs from this work in that the polyhedral model is not supported, the virtual ISA is lower level, and HPVM includes a runtime system.

LIFT [18] is a functional, data parallel IR that enables control and dataflow optimizations, and targets OpenCL code. Algorithms in LIFT are represented by compositions of a defined set of functional primitives derived from common parallel programming patterns such as *map*, *reduce*, *iterate*, and lambda functions. The data layout can be transformed with the *split*, *join*, *gather*, and *scatter* patterns. The *slide* pattern applies an operation to input data across a sliding window to represent stencil patterns [154]. Applications represented in LIFT are limited to the predefined patterns and cannot be used to define arbitrary computations. The goal is to maintain the algorithmic representation during compilation to effectively decouple parallelism from code generation.

Stateful Dataflow multiGraph (SDFG) [155] is a data-centric intermediate representation that separates program definition from optimizations by combining fine-grained data dependences with high-level control flow. Programs are specified in the data-centric environment (DaCe) that includes Python, Matlab, and TensorFlow [156] frontends. Code generation is supported for CPU, GPU, and FPGA architectures. SDFGs differ from PDFGs in that the polyhedral model is not supported, but similar scheduling and storage transformations can be applied.

7.4 Polyhedral Compilers

The polyhedral model provides a mathematical basis for representing loop nest iterations as lattice points within a polyhedron. Each iterator corresponds to one dimension in the corresponding iteration space. The polyhedra can be manipulated using set operations to perform compile-time loop transformations including interchange, reversal, skewing or shifting, fusion, fission, and tiling. The model is more flexible and expressive than the unimodular matrix approach [157] that preceded it. Iteration spaces represented as sets can be applied to reorder execution schedules to improve locality or enable effective vectorization across SIMD lanes. The approach is generally limited to affine iteration spaces that can be expressed with Presburger arithmetic.

Many compiler optimization frameworks are based on the polyhedral model. The Polyhedral Extraction Tool (PET) [36] is applied at the source code level using C pragmas to identify static control parts of programs (SCoPs). The Integer Set

Library (ISL) [52] is used to perform set operations, and the Clunky Loop Generator (CLooG) [46] for code generation. Explicit SIMD code generation for x86 CPU architectures was implemented by Kong et al. [158].

Polly [53] applies polyhedral transformations at the LLVM IR level. Polly-ACC [54] is an extension that targets both CPU and GPU devices, with support for Pthreads, CUDA, and OpenCL backends. The ΣC language [159] applies the polyhedral model to dataflow programs in the context of agent-oriented programming. These approaches differ from this work in that they do not provide a DSL, performance model, or temporary storage optimizations, nor do they support non-affine transformations.

Polyhedral expression propagation (PEP) [160] defines a C-like input language that for generating code including scalar or array data accesses, affine and arbitrary expressions, and loop statements. Data dependences of each statement set are also represented as sets. Data accesses within an expression are replaced with the expression that generated them, effectively introducing redundant computation. The redundant computation for conditionals described in this work is similar to PEP, however temporary storage reductions provided are not supported.

TIRAMISU [6] is a polyhedral compiler with a four-level IR, including an algorithmic expression layer, a computation scheduling layer, a data management layer, and a communication layer. The polyhedral model is supported using ISL and code is lowered to the Halide-IR. Code generation supports OpenMP, CUDA, and MPI backends. However, neither a performance model nor autotuning are provided to automate transformations, rather they are manually specified by the programmer using the provided eDSL.

7.4.1 Non-affine Transformations

Non-affine polyhedral transformations can be divided into two groups, those that support uninterpreted functions, and those that rely on predicates. The Omega library [48] included uninterpreted function support, and was later expanded in Omega+ [11]. Non-affine transformations were implemented in the CHiLL compiler framework by Venkat et al. [29]. These were extended to sparse inspector/executor applications with AST transformations such as *make-dense* and *compact-and-pad* [39], and wavefront parallelization for sparse matrix factorization [26].

The sparse polyhedral framework [8,161] introduced the IEGenLib library for specifying the composition of non-affine polyhedral expressions, including uninterpreted functions for inspector/executor applications with indirect memory accesses. Data transformations were specified with run-time reorderings, and inspectors were defined in terms of inspector-dataflow graphs (IDGs), similar to the polyhedral+dataflow graphs described here.

The polyhedral model can also support non-affine transformations via exit and control predicates [103]. Early loop exits are implemented with exit predicates and irregular control flow with control predicates. These enable the expression of *while* loops and non-affine *if* statements in the polyhedral model. AST generation for complex execution was improved with schedule trees [43], and implemented in ISL [52]. These techniques were applied to loops with dynamic data-dependent bounds [30].

7.4.2 Tensor Compilers

TensorFlow [156] is a dataflow based programming model for machine learning that includes the accelerated linear algebra compiler (XLA). The Tensor Algebra

Compiler [105] (TACO) allows computations to be defined with a set notation and compiled into optimized code. Tensor Comprehensions [162] (TC) is a DSL for tensor computations lowers Halide-IR [4] to a polyhedral representation, provides a JIT autotuning framework with code caching, and supports data layout optimizations. The TC code generator includes CPU and GPU backends.

The iterator trees described here are similar in purpose to schedule trees [43] in ISL, or iteration graphs in the tensor algebra compiler (TACO) [105]. TACO was extended in [163] to support additional matrix and tensor formats, similar to the inspector transformations defined in this work. The temporary workspaces used in the CPD benchmark in Chapter 6 are similar in purpose to TACO workspaces [164]. TACO does not incorporate the polyhedral model, nor does the specification language support loop-carried dependences.

7.4.3 Visualization Tools

Polyhedral transformations were applied to the LabVIEW graphical dataflow language by the PolyGLoT tool [165]. Other tools for visualizing computations within the polyhedral framework include Clint [166], later extended in [167], and the R-Stream auto-parallelizing compiler [168]. CFGExplorer [169] is a tool to help developers understand application level control flow. These visualization tools differ from the work described here in that data spaces are not treated as first class entities that are distinct from the iteration spaces.

7.4.4 Memory Optimizations

Memory optimizations in the polyhedral model were implemented in the Alpha functional language [170]. Array privatization [171] is a technique for enhancing parallelism by privatizing variables per thread by applying dataflow analysis both inter- and intraprocedurally. Array contraction [19] optimizes code by scalarizing array variables inside a loop, saving memory by removing temporary arrays and increasing data locality.

Communication avoiding optimizations reduce the movement of data within a memory hierarchy by rescheduling statements or overlapping computation with communication between nodes. Other communication avoiding optimizations include loop fusion, overlapped tiling, and wavefront parallelization to decrease memory traffic. Communication avoiding optimizations have been studied by Demmel et al. [172,173] This work incorporates similar techniques into a unifiied compiler intermediate representation with an integrated peformance model, but does not yet provide support for distributed applications.

7.5 Library Based Approaches

High performance, manually tuned libraries are developed for a specific target application, language, or architecture. These include BLAS, LAPACK, or Eigen, for dense linear algebra, and PETSc [174], Overture [175], or Chombo [100] for PDE solvers. Hand-tuned libraries are typically architecture specific and can lack portability. Autotuning libraries include ATLAS for linear algebra, OSKI [104] for sparse computations, and FFTW for fast Fourier transforms. SPLATT [31] is a sparse tensor library that introduced the compressed sparse fiber format, and performs high performance tensor decomposition.

Programming models such as Sequoia [176, 177] and *Legion* [178] allow explicit programming of the memory hierarchy. Legion describes data layouts, dependences, and locality with logical regions. The model exposes two primary abstractions, a *region* encapsulates a collection of data structures or objects, and a *task* defines a function that accesses those regions. Each region can be partitioned into disjoint or overlapping subregions. Interactions between regions and tasks are defined by privileges (e.g., read-only, read-write, reduce) and coherence (e.g., exclusive, atomic). The relationships are used to derive parallelism at compile time, or concurrency at runtime. Each logical region has one or more physical instances, as data can be replicated to increase parallelism. The scheduler applies a work-stealing technique similar to the Cilk runtime system [152]. The Regent programming language [179] is designed to exploit the logical regions of Legion. This work differs from the Legion approach in that it performs transformations at compile time rather than in a library at run time.

7.5.1 Adaptive Mesh Refinement

Block-structured adaptive mesh refinement (SAMR) is a computational technique for solving large-scale hyperbolic, parabolic, or elliptic PDE sets, computing different regions of the problem domain with different spatial resolutions, maintaining the blocks in some logically organized hierarchy [107].

BoxLib [180] is an AMR framework for implementing multigrid PDE solvers in many physics applications. The initial version employed a hybrid MPI/OpenMP parallelization approach, with OpenMP threads assigned to loops over individual grid cells. BoxLib covers a sizable code base, and manually tiling existing code is both error prone and labor intensive. *Perilla* [181] is a runtime system that reads the metadata to provide a task-driven, asynchronous parallelism to BoxLib and TiDA [182].

The Chombo library [106] consists of a set of C++ classes designed to support SAMR and is part of the BoxLib toolkit [180]. The `Proto` library is an extension of this work, providing performance portability by supporting CPU and GPU backends. AMREx [183] is another SAMR framework in C++, designed for PDEs with complex boundary conditions with support for heterogenous architectures.

TiDA is a multicore programming model based on tiling with NUMA-awareness [182], targeting geometric multigrid applications. Many runtime systems assume uniform distance between cores, an assumption that is neither portable nor scalable. TiDA replaces the data abstraction used by the original source code. The TiDA library probes the hardware at runtime with the `hwloc` tool [184]. TiDA is coupled with MPI for transferring data between nodes, but shared memory multithreading must be implemented by the programmer.

This work is complementary to these libraries in that it can be used as an intermediate representation for structed grid applications. However, it differs in that it applies static transformations at compile time, rather than with a run time library.

7.6 Performance Modeling

Performance data can be provided to the programmer or compiler to inform optimization decisions, and narrow the potentially vast search space. Performance modeling and benchmarking tools provide analytical techniques to improve performance by estimating the benefits of particular optimizations on specific target architectures.

Performance models must consider several hardware parameters for accuracy, such as register pressure, the number of caches and sizes, the cache coherency model, SIMD vector pipeline length, NUMA distances, and prefetchers. Software parameters must be considered as well, like compiler flags, optimization stages, and tiling strategies. This section is not an exhaustive treatment of performance modeling, but is intended to motivate the role which modeling plays in compiler optimizations.

Many decisions must be made while compiling source code to a target architecture, including execution schedule, data layout, parallelization strategy, and vectorization. Efforts to develop practical performance models to inform these decisions include Roofline [185], the execution-cache-memory (ECM) model [186], and polyhedral performance modeling [187]. The Roofline model was extended to model energy efficiency [188]. The Empirical Roofline model Toolkit (ERT) [189] is a practical tool for generating roofline models.

LIKWID [118] provides an API and a set of command line utilities for modeling performance on multicore x86 architectures, making use of hardware performance counters. LIKWID, along with Intel SDE and VTune, is one of the tools used n this work to verify the estimates produced by the performance model.

The performance model generated by the PDFG-IR is similar to Roofline in that it includes FLOPs memory bandwidth to compute an estimate of arithmetic intensity. Though the accuracy of the model was initally evaluated using experimental data, it is not produced empirically, nor does it access hardware performance counters.

7.7 Autotuning

Autotuning compilers attempt to find the best performing implementation by searching an optimization decision space. A programmer defines the search space of potential optimizations for each implementation, and the autotuner explores that space to determine the best performing code variant. This can make the optimization process more efficient and portable, but also incurs additional challenges. These include representing the configurable tuning parameters in the search space, constraining the space size, and handling tradeoffs between multiple objectives. Autotuners are limited by the need to execute numerous code variants on the target architecture each time the code is updated, and a potentially vast search space to traverse. Limiting the search space requires assumptions to be made, such as favoring vectorization over storage reductions. Other disadvantages are lengthy compile times and the need to benchmark the code on numerous architectures.

Projects that implement autotuning include PolyMage [130], and PetaBricks [133]. Halide initially included an autotuner based on a genetic algorithm [4], and was later integrated with OpenTuner [139]. CHiLL has also been integrated with OpenTuner and applied to geometric multigrid (GMG) applications [138]. SPIRAL [190] is an autotuning code generator for digital signal processing (DSP) applications. Kamil et al. present an autotuning framework tailored to stencil computations [191]. Park et al. combine predictive modeling with a polyhedral compiler [192]. ISAAC is an autotuner for compute-bound linear algebra kernels that applies predictive modeling with regression [193]. Autotuning has also been applied to reduce energy consumption using dynamic voltage scaling [194].

OpenTuner [139] is a general software framework for developing extensible, domain-specific autotuners. OpenTuner uses *ensembles* to combine multiple search techniques. These can be built-in or user-defined, and are executed in parallel. A greedy heuristic selects well performing techniques and allocates more tests to them. Less favorable variants will receive fewer tests, until disabled altogether. Collaboration between ensembles is enabled by storing results in a common database .

Autotuning is not directly supported by this work, however the dataflow graph variants that are generated from an algorithmic specification could be used as inputs to an autotuning framework. Optimization decisions are either user guided or informed by the internal performance model.

7.8 Summary

The programming language, compiler, and library-based techniques described in this chapter are summarized in Table 7.1. The table columns indicate the technology type, e.g., DSL, compiler, or library, the application domain, the targeted general purpose programming language, and whether the polyhedral model and/or irregular applications are supported.

Table 7.1: Summary of related programming language, compiler, and library-based technologies for algorithmic representation and optimization.

Technology	Type	Domain	Language	Polyhedral	Irregular
Halide	DSL	Image Proc.	C++		
PolyMage	DSL	Stencils	Python	X	
TIRAMISU	DSL	Regular	C++	X	X
SDFG / DaCe	DSL/IR	All	Python		X
TACO	DSL	Sparse	C++		X
PetaBricks	DSL	Regular	C++		
AlphaZ	Scripted	Regular	C++	X	
CHiLL	Scripted	All	C/C++	X	X
DFGR	IR	Regular	C++	X	
LIFT	IR	Regular	OpenCL		
HPVM	IR	All	LLVM		X
Polly	Compiler	Regular	LLVM	X	
PENCIL	Compiler	Regular	C	X	
Legion	Library	Regular	C++		
TiDA	Library	AMR	C++/Fortran		
SPLATT	Library	Tensors	C/C++		X

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This chapter summarizes and restates the contributions of this dissertation, then discusses potential directions for future research.

8.1 Compiler IR for Loop and Data Transformations

A compiler intermediate representation has been developed that combines macro-dataflow graphs with the polyhedral model. Transformations can be applied to both affine and non-affine iteration and data spaces. Execution schedules can be modified to enable further optimizations. Reuse distance and liveness analyses allow memory allocation to be reduced. Data access mappings are automatically generated for the reduced spaces The IR can be targeted to multiple hardware architectures for performance portability. The IR was evaluated with the MiniFluxDiv CFD benchmark, and outperformed implementations in DSLs such as Halide and PolyMage.

8.2 Domain Specific Language

A corresponding domain specific language has been developed to express computations at a high-level, decoupling the algorithms from the execution schedules, data spaces, and underlying target architectures. The language is designed to approximate the mathematical expressions understood by domain experts. The IR is constructed from the language specifications. Either component of the inspector/executor paradigm can be expressed. This approach was competitive with other compiler frameworks, including CHiLL and TACO, and with hand tuned libraries such as OSKI and SPLATT.

8.3 Integration with Existing Application

The polyhedral+dataflow graph IR was successfully integrated with `Proto`, an eDSL and library for structured grid algorithms such as those found in adaptive mesh refinement applications. The transformations provided by the IR, combined with the internal performance model it generates, enable performance improvements of up to 3X on CPU, and 2.5X on GPU target architectures. This work demonstrated the ability of the IR to improve the performance of existing applications, or those implemented with different programming abstractions.

8.4 Algorithms with Sparse Structures

The polyhedral+dataflow language was used to implement two numerical solver algorithms, conjugate gradient for sparse matrices, and canonical polyadic decomposition for sparse tensors. This work highlights the language's ability to express complete algorithms, including methods to specify inspectors and executors for multiple sparse data formats for each algorithm. The generated inspectors are composable, and the executors can be exchanged to select the most performant variant for each input matrix or tensor. The CG performance results were competitive with the Eigen library, but the CPD implementation was outperformed by the hand-tuned SPLATT tensor library.

8.5 Contributions

1. Development of the polyhedral+dataflow graph intermediate representation (PDFG-IR) that expresses execution schedules, dataflow, memory interactions, and program statements in a manner that expands the set of automated transformations available to optimizing compilers. The polyhedral model is combined with macro-dataflow graphs to explicitly represent data requirements, including data type, domain, and size. The graphs encapsulate code with execution schedules and data mappings for both persistent and temporary storage spaces.

2. Definition and implementation of compiler transformations to modify the execution schedules and storage mappings of the specified computation. These operations include statement rescheduling, producer-consumer and read-reduce loop fusion, and other loop transformations, such as unrolling, splitting, and tiling. Storage reductions are determined using reuse distance and reachability analyses. A memory allocation algorithm based on liveness analysis [7] is described that allocates sufficient space for only those data that are live at each point during a computation.

3. Extension of the IR to support irregular applications using the inspector/executor approach [8]. The inspector-executor method is applied when code or data transformations require run time support, including run time dependence analysis or data transformations. Both inspectors and executor components can be represented and optimized, by transforming both data and iteration spaces. Non-affine, data dependent loop bounds are represented by *uninterpreted* functions [9] and converted into *explicit* represetations at run time.

4. Development of an embedded domain specific language to construct the IR. Numerical algorithms are expressed in C++ using a combination of iterators, functions, constraints, spaces, and executable statements. An iteration space is composed of an iterator set and their corresponding boundary constraints. Data spaces are derived from access functions in the statements. A computation consists of an iteration space, execution schedule, and statement list. The PDFG-IR is generated from the eDSL specifications.

5. Generation of an internal performance model for each graph variant. Many different graph variants can be generated from an initial graph specification by applying the supported transformations. A performance model is generated for each variant that include estimates of floating point operations (FLOPs), memory throughput, and arithmetic intensity. The model can be used to reason about the performance of a given variant on a target architecture.

8.6 Future Directions

The specification language could be expanded to enable applications in other scientific domains, such as graph algorithms for analytics, string matching for bioinformatics, or machine learning primitives for deep learning. Recursive algorithms could be supported by introducing data spaces to simulate the run time stack. The language could also benefit from enhanced interoperability with other languages, such as Python bindings for Jupyter notebooks. Deriviation of PDFG-IR specifications from legacy applications could be improved, for example by extracting information from the Clang AST [195]. The PDFG-IR could be integrated with other developing multi-level compiler frameworks such as MLIR [196] or TVM [197].

Additional target backends could be implemented, for example LLVM IR [55], or SPIR-V for OpenCL [198]. Support could also be added for more diverse hardware architectures such as FPGAs. This work could also be expanded to generate code for distributed applications such as MPI [199], or GASNet [200].

The performance model could be augmented with additional parameters to describe the target architecture, such as multi-level cache hierarchies, or the number of SIMD lanes. This would help guide optimizations, emphasizing loop fusion and data reduction for CPUs, or loop overhead and control flow reduction for GPUs. The performance model could be improved by coupling it with an autotuning framework such as OpenTuner [139], by generating the code for each legal graph variant, and executing it on the target platform.

The visual elements of the dataflow graphs could be further developed to provide an interactive tool for domain scientists and performance engineers. The feedback from the graphs would allow the experts to reason about the performance of their applications.

This research demonstrates that a robust, expressive compiler intermediate representation that decouples algorithmic specification from code optimization with an emphasis on temporary storage reduction can generate fast and efficient code for multiple target architectures. The IR capabilities are enabled by execution schedule transformations using polyhedral compilation techniques, coupled with dataflow optimizations, and an internal performance model.

REFERENCES

[1] John P Holdren and Shaun Donovan. National strategic computing initiative strategic plan. Technical report, National Strategic Computing Initiative Executive Council Washington United States, 2016.

[2] Alex Hutcheson and Vincent Natoli. Memory bound vs. compute bound: A quantitative study of cache and memory bandwidth in high performance applications. In *Technical report, Stone Ridge Technology*. 2011.

[3] José Nelson Amaral. *Languages and Compilers for Parallel Computing*. Springer, 2008.

[4] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[5] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 429–443. ACM, 2015.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 193–205. IEEE Press, 2019.

[7] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[8] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. An approach for code generation in the sparse polyhedral framework. *Parallel Computing*, 53:32–57, 2016.

[9] Randal E Bryant, Shuvendu K Lahiri, and Sanjit A Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *International Conference on Computer Aided Verification*, pages 78–92. Springer, 2002.

[10] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[11] Chun Chen. Polyhedra scanning revisited. *ACM SIGPLAN Notices*, 47(6):499–508, 2012.

[12] Sven Verdoolaege. *barvinok: User Guide*. Compsys, 2015.

[13] Michelle Mills Strout, Fabio Luporini, Christopher D Krieger, Carlo Bertolli, Gheorghe-Teodor Bercea, Catherine Olschanowsky, J Ramanujam, and Paul HJ Kelly. Generalizing run-time tiling with the loop chain abstraction. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1136–1145. IEEE, 2014.

[14] Ian J Bertolacci, Michelle Mills Strout, Stephen Guzik, Jordan Riley, and Catherine Olschanowsky. Identifying and scheduling loop chains using directives. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 57–67. IEEE, 2016.

[15] Eddie C Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 265–277. ACM, 2018.

[16] Daniel Orozco. Tideflow: A parallel execution model for high performance computing programs. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 211–211, 3 Park Ave, New York, NY, USA, 2011. IEEE, IEEE Press.

[17] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Polyhedral optimizations for a data-flow graph language. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 57–72, Salmon Tower Building, New York, NY, USA, 2015. Springer, Springer Publishing.

[18] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*, pages 74–85. IEEE, 2017.

[19] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+ cl@ k: An implementation of lattice-based array contraction in the source-to-source translator rose. *ACM SIGPLAN Notices*, 42(7):73–82, 2007.

[20] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 29–38. ACM, 2013.

[21] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[22] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Effective automatic parallelization of stencil computations. In *ACM sigplan notices*, volume 42, pages 235–244. ACM, 2007.

[23] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[24] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[25] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 313–323. IEEE, 2015.

[26] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. IEEE Press, 2016.

[27] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[28] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky,

Anand Venkat, and Michelle Mills Strout. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 594–609. ACM, 2019.

[29] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 185. ACM, 2014.

[30] Jie Zhao, Michael Kruse, and Albert Cohen. A polyhedral compilation framework for loops with dynamic data-dependent bounds. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 14–24. ACM, 2018.

[31] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 61–70. IEEE, 2015.

[32] Payal Nandy, Mary Hall, Eddie C Davis, Catherine Olschanowsky, Mahdi Soltan Mohammadi, Wei He, and Michelle Strout. Abstractions for specifying sparse matrix data transformations. In *Proc. 8th Int. Workshop Polyhedral Compilation Techn.(IMPACT)*, pages 1–10, 2018.

[33] Richard Vuduc Jiajia Li, Jimeng Sun. Hicoo: Hierarchical storage of sparse tensors. *SC*, 2018.

[34] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[35] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[36] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, pages 1–16, 2012.

[37] Ian J Bertolacci, Michelle Mills Strout, Jordan Riley, Stephen MJ Guzik, Eddie C Davis, and Catherine Olschanowsky. Using the loop chain abstraction to schedule across loops in existing code. *International Journal of High Performance Computing and Networking*, 13(1):86–104, 2019.

[38] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[39] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Notices*, volume 50, pages 521–532. ACM, 2015.

[40] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988.

[41] Katsumi Nomizu, Nomizu Katsumi, and Takeshi Sasaki. *Affine differential geometry: geometry of affine immersions*. Cambridge university press, 1994.

[42] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *International Conference on Computer Aided Verification*, pages 400–411. Springer, 1997.

[43] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):12, 2015.

[44] Tobias Grosser, Albert Cohen, Justin Holewinski, Ponuswamy Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 66. ACM, 2014.

[45] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Oil and water can mix: an integration of polyhedral and ast-based transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 287–298. IEEE Press, 2014.

[46] Cédric Bastoul. Generating loops for scanning polyhedra: Cloog users guide. *Polyhedron*, 2:10, 2004.

[47] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*, pages 185–201. Springer, 2006.

[48] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The omega calculator and library, version 1.1. 0. *College Park, MD*, 20742:18, 1996.

[49] Hervé Le Verge. *A note on Chernikova's algorithm*. PhD thesis, INRIA, 1992.

[50] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[51] Christopher D. Krieger, Michelle Mills Strout, Catherine Olschanowsky, Andrew Stone, Stephen Guzik, Xinfeng Gao, Carlo Bertolli, Paul H.J. Kelly, Gihan Mudalige, Brian Van Straalen, and Sam Williams. Loop chaining: A programming abstraction for balancing locality and parallelism. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, May 2013.

[52] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[53] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly: performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[54] Tobias Grosser and Torsten Hoefler. Polly-acc transparent compilation to heterogeneous hardware. In *Proceedings of the 2016 International Conference on Supercomputing*, page 1. ACM, 2016.

[55] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[56] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[57] Abhinav Jangda and Uday Bondhugula. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 261–275. ACM, 2018.

[58] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

[59] T Sterling. Studies on optimal task granularity and random mapping. *Advanced Topics in Dataflow Computing and Multithreading, 1995*, pages 349–365, 1995.

[60] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 202–211, 2 Penn Plaza, Ste 701, New York, NY, USA, 1986. ACM, ACM Press.

[61] S. Ramaswamy and P. Banerjee. Processor allocation and scheduling of macro dataflow graphs on distributed memory multicomputers by the paradigm compiler. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 134–138, 3 Park Ave, New York, NY, USA, Aug 1993. IEEE Press.

[62] G. N. Srinivasa Prasanna, A. Agrawal, and B. R. Musicus. Hierarchical compilation of macro dataflow graphs for multiprocessors with local memory. *IEEE Trans. Parallel Distrib. Syst.*, 5(7):720–736, July 1994.

[63] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[64] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[65] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):869–916, 1998.

[66] I-Jui Sung, John A Stratton, and Wen-Mei W Hwu. 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*, pages 513–522. ACM, 2010.

[67] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *International Conference on Compiler Construction*, pages 225–245. Springer, 2011.

[68] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[69] Alessandro Cilardo and Luca Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):45, 2015.

[70] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, volume 49, pages 65–76. ACM, 2014.

[71] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. In *ACM SIGPLAN Notices*, volume 31, pages 244–255. ACM, 1996.

[72] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

[73] Phillip Colella. Defining software requirements for scientific computing. *presentation*, 2004.

[74] Steven C Chapra, Raymond P Canale, et al. *Numerical methods for engineers*. Boston: McGraw-Hill Higher Education,, 2010.

[75] Zhengqiu Zhang. An improvement to the brent's method. *International Journal of Experimental Algorithms*, 2(1):21–26, 2011.

[76] Giesela Engeln-Müllges and Frank Uhlig. *Numerical algorithms with C*. Springer Science & Business Media, 2013.

[77] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[78] Michael JD Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162, 1964.

[79] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[80] Leslie Hogben. *Handbook of linear algebra*. Chapman and Hall/CRC, 2013.

[81] Timothy A Davis. Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):196–199, 2004.

[82] Peter McCorquodale and Phillip Colella. A high-order finite-volume method for conservation laws on locally refined grids. *Communications in Applied Mathematics and Computational Science*, 6(1):1–25, 2011.

[83] Gilbert Strang. Linear algebra and its applications.: Thomson brooks. *Cole, Belmont, CA, USA*, 2005.

[84] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[85] Roger Penrose. On best approximate solutions of linear matrix equations. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 52, pages 17–19. Cambridge University Press, 1956.

[86] U Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324. ACM, 2012.

[87] Yao Zhang, Jonathan Cohen, and John D Owens. Fast tridiagonal solvers on the gpu. *ACM Sigplan Notices*, 45(5):127–136, 2010.

[88] Joseph WH Liu. Computational models and task scheduling for parallel sparse cholesky factorization. *Parallel computing*, 3(4):327–342, 1986.

[89] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[90] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[91] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[92] William F Tinney and John W Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *proc. IEEE*, 55(11):1801–1809, 1967.

[93] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[94] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual*

symposium on Parallelism in algorithms and architectures, pages 233–244. ACM, 2009.

[95] Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[96] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[97] Francisco Vázquez, José-Jesús Fernández, and Ester M Garzón. A new approach for sparse matrix vector product on nvidia gpus. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.

[98] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 5. ACM, 2015.

[99] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. A study on balancing parallelism, data locality, and recomputation in existing pde solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 793–804, 3 Park Ave, New York, NY, USA, 2014. IEEE Press, IEEE Press.

[100] M. Adams, P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. Technical Report LBNL-6616E, Lawrence Berkeley National Laboratory, 2015.

[101] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 207–218. ACM, 2012.

[102] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, and B. Van Straalen. AMR Godunov Unsplit Algorithm and Implementation. Technical report, Lawrence Berkeley National Laboratory, 2008.

[103] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you

think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[104] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[105] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[106] P Colella, DT Graves, TJ Ligocki, DF Martin, D Modiano, DB Serafini, and B Van Straalen. Chombo software package for amr applications design document. *Available at the Chombo website: http://seesar. lbl. gov/ANAG/-chombo/(September 2008)*, 2009.

[107] Anshu Dubey, Ann Almgren, John Bell, Martin Berzins, Steve Brandt, Greg Bryan, Phillip Colella, Daniel Graves, Michael Lijewski, Frank Löffler, et al. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing*, 74(12):3217–3227, 2014.

[108] Milo R Dorr, Phillip Colella, Mikhail A Dorf, Debojyoti Ghosh, Jeffrey AF Hittinger, and Peter O Schwartz. High-order discretization of a gyrokinetic vlasov model in edge plasma geometry. *Journal of Computational Physics*, 373:605–630, 2018.

[109] Genia Vogman and Phillip Colella. Continuum kinetic plasma modeling using a conservative 4th-order method with amr. In *APS Meeting Abstracts*, 2012.

[110] SL Cornford, DF Martin, V Lee, AJ Payne, and EG Ng. Adaptive mesh refinement versus subgrid friction interpolation in simulations of antarctic ice dynamics. *Annals of Glaciology*, 57(73):1–9, 2016.

[111] David Trebotich and Daniel Graves. An adaptive finite volume method for the incompressible navier–stokes equations in complex geometries. *Communications in Applied Mathematics and Computational Science*, 10(1):43–82, 2015.

[112] David Trebotich, Chaopeng Shen, Greg Miller, Sergi Molins, and Carl Steefel. An adaptive embedded boundary method for pore scale reactive transport.

[113] Brian Van Straalen, David Trebotich, Andrey Ovsyannikov, and Daniel T Graves. Scalable structured adaptive mesh refinement with complex geometry. *Exascale Scientific Applications: Scalability and Performance Portability*, page 307, 2017.

[114] Eleuterio F Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer Science & Business Media, 2013.

[115] Ponnuswamy Sadayappan. Domain specific language support for exascale. Technical report, The Ohio State Univ., Columbus, OH (United States), 2017.

[116] Tarek El-Ghazawi and Lauren Smith. Upc: unified parallel c. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27. ACM, 2006.

[117] John Bachan, Dan Bonachea, Paul H Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Scott B Baden. The upc++ pgas library for exascale computing. In *Proceedings of the Second Annual PGAS Applications Workshop*, page 7. ACM, 2017.

[118] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010.

[119] Phillip Colella and Paul R Woodward. The piecewise parabolic method (ppm) for gas-dynamical simulations. *Journal of computational physics*, 54(1):174–201, 1984.

[120] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

[121] Pierre Courrieu. Fast computation of moore-penrose inverse matrices. *arXiv preprint arXiv:0804.4809*, 2008.

[122] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, Fabrice Rossi, and Rhys Ulerich. *GNU scientific library*. Network Theory Limited, 2002.

[123] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[124] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.

[125] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, C Bischof, and Danny Sorensen. Lapack: A portable linear algebra library for high-performance

computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990.

[126] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[127] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using orio. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11. IEEE, 2009.

[128] Alina Sbirlea, Louis-Noel Pouchet, and Vivek Sarkar. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on*, pages 38–45, 3 Park Ave, New York, NY, USA, 2014. IEEE, IEEE Press.

[129] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. 2012.

[130] Vinay V Vasista. Automatic optimization of geometric multigrid methods using a dsl approach. 2017.

[131] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 138–149. IEEE, 2015.

[132] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[133] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[134] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)*, 43(3):24, 2016.

[135] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[136] Qing Yi. Poet: a scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience*, 42(6):675–706, 2012.

[137] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 17–31. Springer, 2012.

[138] Protonu Tharindu, Mary Hall, and Protunu Basu. Automating compiler-directed autotuning for phased performance behavior. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 1362–1372. IEEE, 2017.

[139] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 303–315. IEEE, 2014.

[140] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[141] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12), 2009.

[142] Paul Osmialowski. How the flang frontend works. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC 2017)*, 2017.

[143] Thomas Würthinger. Graal and truffle: modularity and separation of concerns as cornerstones for building a multipurpose runtime. In *Proceedings of the companion publication of the 13th international conference on Modularity*, pages 3–4. ACM, 2014.

[144] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM, 2015.

[145] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[146] James McGraw, Stephen Skedzielewski, Stephen Allan, Dale Grit, Rob Oldehoeft, John Glauert, Ivan Dobes, and Paul Hohensee. Sisal: streams and iteration in a single-assignment language. language reference manual, version 1. 1. Technical report, Lawrence Livermore National Lab., CA (USA), 1983.

[147] James R McGraw. Data-flow computing: the val language. *ACM Transactions on Programming Languages and systems*, 4(1):44–82, 1982.

[148] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. Polyhedral optimizations of explicitly parallel programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 213–226. IEEE, 2015.

[149] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, et al. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736. ACM, 2009.

[150] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[151] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *ACM SIGPLAN Notices*, volume 52, pages 249–265. ACM, 2017.

[152] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[153] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: heterogeneous parallel virtual machine. In *ACM SIGPLAN Notices*, volume 53, pages 68–80. ACM, 2018.

[154] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with lift. In *International Symposium on Code Generation and Optimization*, 2018.

[155] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 81. ACM, 2019.

[156] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[157] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.

[158] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. When polyhedral transformations meet simd code generation. In *ACM Sigplan Notices*, volume 48, pages 127–138. ACM, 2013.

[159] Romain Fontaine, Laure Gonnord, and Lionel Morel. Polyhedral dataflow programming: a case study. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018.

[160] Johannes Doerfert, Shrey Sharma, and Sebastian Hack. Polyhedral expression propagation. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 25–36. ACM, 2018.

[161] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 61–75. Springer, 2012.

[162] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[163] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):123, 2018.

[164] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. In *Proceedings of the 2019 IEEE/ACM*

International Symposium on Code Generation and Optimization, pages 180–192. IEEE Press, 2019.

[165] Somashekaracharya G Bhaskaracharya and Uday Bondhugula. Polyglot: a polyhedral loop transformation framework for a graphical dataflow language. In *International Conference on Compiler Construction*, pages 123–143. Springer, 2013.

[166] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Clint: A direct manipulation tool for parallelizing compute-intensive program parts. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 109–112. IEEE, 2014.

[167] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Visual program manipulation in the polyhedral model. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(1):16, 2018.

[168] Eric Papenhausen, Bing Wang, M Harper Langston, Muthu Baskaran, Tom Henretty, Taku Izubuchi, Ann Johnson, Chulwoo Jung, Meifeng Lin, Benoit Meister, et al. Polyhedral user mapping and assistant visualizer tool for the r-stream auto-parallelizing compiler. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 180–184. IEEE, 2015.

[169] Sabin Devkota and Katherine E Isaacs. Cfgexplorer: Designing a visual control flow analytics system around basic program analysis operations. In *Computer Graphics Forum*, volume 37, pages 453–464. Wiley Online Library, 2018.

[170] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):773–815, 2000.

[171] Peng Tu and David Padua. Automatic array privatization. In *Compiler optimizations for scalable parallel systems*, pages 247–281. Springer, 2001.

[172] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2008.

[173] Nicholas Sullender Knight. *Communication-optimal loop nests*. PhD thesis, UC Berkeley, 2015.

[174] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, LD Dalcin, Victor Eijkhout, W Gropp, Dinesh Kaushik, et al.

Petsc users manual revision 3.8. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[175] David L Brown, William D Henshaw, and Daniel J Quinlan. Overture: An object-oriented framework for solving partial differential equations. In *International Conference on Computing in Object-Oriented Parallel Environments*, pages 177–184. Springer, 1997.

[176] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[177] Michael Bauer, John Clark, Eric Schkufza, and Alex Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. *ACM SIGPLAN Notices*, 46(8):13–24, 2011.

[178] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[179] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 81. ACM, 2015.

[180] Weiqun Zhang, Ann Almgren, Marcus Day, Tan Nguyen, John Shalf, and Didem Unat. Boxlib with tiling: An adaptive mesh refinement software framework. *SIAM Journal on Scientific Computing*, 38(5):S156–S172, 2016.

[181] Tan Nguyen, Didem Unat, Weiqun Zhang, Ann Almgren, Nufail Farooqi, and John Shalf. Perilla: Metadata-based optimizations of an asynchronous runtime for adaptive mesh refinement. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 81. IEEE Press, 2016.

[182] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. Tida: High-level programming abstractions for data locality management. In *International Conference on High Performance Computing*, pages 116–135. Springer, 2016.

[183] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. Amrex: a framework for block-structured adaptive mesh refinement. 2019.

[184] Brice Goglin. Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 74–81. IEEE, 2014.

[185] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[186] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 207–216. ACM, 2015.

[187] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 3–13. ACM, 2018.

[188] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 661–672. IEEE, 2013.

[189] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 129–148. Springer, 2014.

[190] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[191] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[192] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P Sadayappan. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming*, 41(5):704–750, 2013.

[193] Philippe Tillet and David Cox. Input-aware auto-tuning of compute-bound hpc kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 43. ACM, 2017.

[194] Mohammed Sourouri, Espen Birger Raknes, Nico Reissmann, Johannes Langguth, Daniel Hackenberg, Robert Schöne, and Per Gunnar Kjeldsberg. Towards fine-grained dynamic tuning of hpc applications on modern multi-core architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. ACM, 2017.

[195] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, pages 1–2, 2008.

[196] Chris Lattner and Jacques Pienaar. Mlir primer: A compiler infrastructure for the end of moore's law. 2019.

[197] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[198] John Kessenich, Graham Sellers, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016.

[199] Brandon Barker. Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede*, volume 262, 2015.

[200] Dan Bonachea and P Hargrove. Gasnet specification, v1. 8.1. 2017.