Automated Runtime Analysis and Adaptation for Scalable Heterogeneous Computing

Ahmed E. Helal

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

December 3, 2019
Blacksburg, Virginia

Keywords: Parallel Architectures, Accelerators, Heterogeneous Computing, Performance Modeling, Runtime Adaptation, Scheduling, Performance Portability, MPI, GPU, LLVM

Automated Runtime Analysis and Adaptation for Scalable Heterogeneous Computing

Ahmed E. Helal

(ABSTRACT)

In the last decade, there have been tectonic shifts in computer hardware because of reaching the physical limits of the sequential CPU performance. As a consequence, current high-performance computing (HPC) systems integrate a wide variety of compute resources with different capabilities and execution models, ranging from multi-core CPUs to many-core accelerators. While such heterogeneous systems can enable dramatic acceleration of user applications, extracting optimal performance via *manual* analysis and optimization is a complicated and time-consuming process.

This dissertation presents *graph-structured program representations* to reason about the performance bottlenecks on modern HPC systems and to guide novel automation frameworks for performance analysis and modeling and runtime adaptation. The proposed program representations exploit domain knowledge and capture the inherent computation and communication patterns in user applications, at multiple levels of computational granularity, via compiler analysis and dynamic instrumentation. The empirical results demonstrate that the introduced modeling frameworks accurately estimate the realizable *parallel* performance and scalability of a given *sequential* code when ported to heterogeneous HPC systems. As a result, these frameworks enable efficient workload distribution schemes that utilize all the available compute resources in a performance-proportional way. In addition, the proposed runtime adaptation frameworks significantly improve the end-to-end performance of important real-world applications which suffer from *limited parallelism and fine-grained data dependencies*. Specifically, compared to the state-of-the-art methods, such an adaptive parallel execution achieves up to an order-of-magnitude speedup on the target HPC systems while preserving the inherent data dependencies of user applications.

Automated Runtime Analysis and Adaptation for Scalable Heterogeneous Computing

Ahmed E. Helal

(GENERAL AUDIENCE ABSTRACT)

Current supercomputers integrate a massive number of heterogeneous compute units with varying speed, computational throughput, memory bandwidth, and memory access latency. This trend represents a major challenge to end users, as their applications have been designed from the ground up to primarily exploit homogeneous CPUs. While heterogeneous systems can deliver several orders of magnitude speedup compared to traditional CPU-based systems, end users need extensive software and hardware expertise as well as significant time and effort to efficiently utilize all the available compute resources.

To streamline such a daunting process, this dissertation presents automated frameworks for analyzing and modeling the performance on parallel architectures and for transforming the execution of user applications at runtime. The proposed frameworks incorporate domain knowledge and adapt to the input data and the underlying hardware using novel static and dynamic analyses. The experimental results show the efficacy of the introduced frameworks across many important application domains, such as computational fluid dynamics (CFD), and computer-aided design (CAD). In particular, the adaptive execution approach on heterogeneous systems achieves up to an order-of-magnitude speedup over the optimized parallel implementations.

To my brother, Ali, who gave me my first thousand books.

Acknowledgments

I was fortunate to work with many wonderful people who made this Ph.D. endeavor a memorable and enjoyable time. I would like to thank Dr. Wu-chun Feng, Dr. Changhee Jung, and Dr. Yasser Hanafy for providing me with a multitude of opportunities to learn and to grow. I am grateful for their guidance, assistance, and support, which have been instrumental in my development as an independent researcher. I would especially like to thank Dr. Feng for empowering me to pursue my research ideas, to effectively communicate my work, and to set the highest expectations for myself.

I wish to thank Dr. Changwoo Min and Dr. Leyla Nazhandali for serving on my committee. I appreciate their insightful comments and feedback on my dissertation.

My thanks to Dr. Ashwin Aji, Dr. Michael Chu, and Dr. Bradford Beckmann (AMD Research) for their assistance and support during our research collaboration. I am grateful to Dr. Amr Bayoumi (AAST) for introducing me to the field of high-performance computing and for his help during the early days of this Ph.D. journey.

I would also like to thank the members of the Synergy Lab at Virginia Tech for their valuable feedback and assistance during my graduate studies. I would especially like to thank Paul Sathre, Vignesh Adhinarayanan, and Dr. Mark Gardner for many interesting and helpful discussions regarding this work.

My thanks to Dr. Yasser Hanafy, Dr. Sedki Riad, and the faculty and staff of the ECE department at Virginia Tech for making my VT-MENA and Prasad fellowships possible.

I am deeply grateful to all my family members and friends. I believe that without your encouragement and support, none of this work would have been accomplished.

Funding Acknowledgment. This work was supported in part by the Air Force Office of Scientific Research (AFOSR) Computational Mathematics Program via Grant No. FA9550-12-1-0442, NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC), NSF CAREER Award (1750503), the DOE PathForward program, and the Synergistic Environments for Experimental Computing (SEEC) Center via a seed grant from the Institute for Critical Technology and Applied Science (ICTAS), an institute dedicated to transformative, interdisciplinary research for a sustainable future. My thanks to Advanced Research Computing (ARC) at Virginia Tech for providing the computational resources used in this work.

Declaration of Collaboration. In addition to my committee members, this dissertation benefited from the following collaborators:

- Dr. Ashwin Aji, Dr. Michael Chu, and Dr. Bradford Beckmann (AMD Research) contributed to the work in Chapter 5.
- Dr. Amr Bayoumi (AAST) contributed to the work in Chapter 6.
- Paul Sathre contributed to the work in Chapter 7. In addition, several members of the Synergy Lab at Virginia Tech, including Kaixi Hou and Sriram Chivukula, contributed to the development of MetaMorph's prototype.

Contents

1 Introduction **1**

1.1 Research Problems 1

1.2 Thesis Statement 3

1.3 Methodology 4

1.4 Contributions 5

1.4.1 Automated Program Analysis for Performance Modeling 5

1.4.2 Runtime Adaptation for Scalable Performance 7

1.4.3 Interoperability Middleware for Simultaneous Execution 8

1.5 Dissertation Outline 9

2 Background and Related Work **10**

2.1 HPC Programming Models 10

2.2 Massively Parallel HPC Architectures 12

2.3 Performance Modeling on HPC Systems 14

2.3.1 Analytical Modeling 14

2.3.2 Simulation 15

2.3.3 Automated Performance Prediction 15

2.3.4 Communication and Scalability Analysis 17

2.4 Data Dependency Management on Massively Parallel HPC Architectures . . 19

3 Automated Estimation of Execution Cost on Heterogeneous Architectures 22

3.1 Introduction 22

3.1.1 AutoMatch: The First-Order Framework 24

3.1.2 Contributions 26

3.2 AutoMatch Approach 27

3.2.1 Hardware Architecture Model 27

3.2.2 Inferring the Architectural Specifications 29

3.2.3 Compiler-based Application Analysis 31

3.2.4 Analytical Modeling 36

3.3 Evaluation 38

3.3.1 Performance Forecasting and Analysis Case study 39

3.3.2 Workload Distribution Case Study 45

3.3.3 Discussion 46

3.4 Conclusion 47

4 Automated Estimation of Communication Cost and Scalability 49

4.1 Introduction 49

4.1.1 CommAnalyzer Framework 51

4.1.2 Contributions 52
4.2 Background 53
4.2.1 Distributed-Memory Execution Model 53
4.2.2 Simple Estimation of Communication Cost 54
4.3 Challenges 55
4.3.1 Indirect Memory Access 55
4.3.2 Workspace Data Structures 55
4.3.3 Singleton Data Items 56
4.3.4 Redundant Communication 56
4.4 CommAnalyzer Approach 57
4.4.1 Communication Characterization 59
4.4.2 Communication Cost Estimation 61
4.4.3 Implementation and Complexity 66
4.5 Efficiency Roofline Model 67
4.6 Evaluation 69
4.6.1 Experimental Setup 70
4.6.2 Benchmarks 73
4.6.3 MiniGhost and Heat2D 74
4.6.4 LULESH 76
4.6.5 K-means 78
4.7 Discussion 79

4.8 Conclusion 80

5 Efficient Dependency Management on Massively Data-Parallel Architectures 82

5.1 Introduction 82

5.2 Background and Motivation 85

5.3 Adaptive Task Aggregation (ATA) 87

5.3.1 Hierarchical DAG Transformation 88

5.3.2 Hierarchical DAG Execution on GPUs 90

5.3.3 Task Aggregation Policies 94

5.4 Evaluation 98

5.4.1 Experimental Setup 99

5.4.2 Experimental Results 102

5.4.3 End-to-End Application Performance 106

5.5 Discussion 107

5.6 Conclusion 108

6 Scalable Execution of Irregular Data-Dependent Computations 109

6.1 Introduction 109

6.2 Background 111

6.3 SPICE Profiling and Modeling 113

6.3.1 Performance Modeling Methods 113

6.3.2 Hardware Architecture Model 114

6.3.3 SPICE Performance Models 115

6.4 SPICE-H Approach 117

6.4.1 Adaptive Parallel Execution on Heterogeneous Distributed-memory Systems 118

6.4.2 Adaptive Circuit Partitioning and Distribution 120

6.4.3 Kernel-Architecture Matching 123

6.5 Evaluation 126

6.6 Conclusion 129

7 Interoperation on Heterogeneous HPC Systems 130

7.1 Introduction 130

7.2 Design Philosophy 133

7.2.1 Interoperability Layer 134

7.2.2 Abstraction Layer 136

7.2.3 Adaptivity Layer 138

7.3 Prototype Implementation 139

7.3.1 Programming Models 139

7.3.2 Top-Level User API 139

7.3.3 Accelerator Back-Ends 145

7.4 Evaluation 146

7.4.1 Applications 146

7.4.2 Programmability and Productivity 148

7.4.3 Experimental Results 150

7.5 Conclusion 155

8 Conclusion and Future Work 156

8.1 Dissertation Summary 157

8.2 Future Work 158

Bibliography 160

List of Figures

1.1 The gap between the computation and communication costs on the Cascades cluster at Virginia Tech. 2

1.2 Overview of the proposed methodology. 4

1.3 Dissertation overview. 5

2.1 VEGA GPU architecture. 12

2.2 GPU kernel. 13

2.3 The execution of a BSP superstep. 13

3.1 Overview of AutoMatch framework. 24

3.2 The abstract hardware architecture. 28

3.3 The design and implementation of AutoMatch compiler. 31

3.4 The ASAP schedule of the application operations on a theoretical architecture with infinite resources. 32

3.5 LRU stack distance analysis example 34

3.6 Detection of global synchronization. 36

3.7 Parallelism and LRU stack distance profiles 40

3.8 Achieved performance vs. AutoMatch's upper bounds. 41

3.9 AutoMatch's prediction of execution bottlenecks. 42

3.10 The prediction sensitivity of AutoMatch. 45

3.11 The performance of compute kernels with the different workload distribution strategies: Default, AutoMatch, and Oracle. 47

4.1 Overview of CommAnalyzer framework. 51

4.2 The SPMD execution model. 54

4.3 Simple estimation of communication cost for matrix multiplication. 54

4.4 Challenges for estimating the communication cost. 55

4.5 An example of value-flow chain detection. 61

4.6 Automated detection of singleton values (with extreme connectivity) using the threshold-based scoring algorithm. 64

4.7 Value decomposition using graph clustering and partitioning algorithms. . . 65

4.8 The effective bisection bandwidth per node. 72

4.9 The communication cost and performance of MatMul and SPMV benchmarks. 74

4.10 The finite difference stencils used in MiniGhost and Heat2D. 75

4.11 The communication cost and performance of MiniGhost and Heat2D. 76

4.12 LULESH uses an unstructured, hexa-hedral mesh with two centering. 77

4.13 The communication cost and performance of LULESH. 78

4.14 The communication cost and performance of K-means. 79

5.1 A triangular matrix and the corresponding DAG for SpTS. 86

5.2 The BSP (level-set) execution of the DAG in Figure 5.1. 87

5.3 ATA transformation of the application DAG in Figure 5.1 for hierarchical execution and dependency management. The adaptive tasks A_1, A_2, and A_4 require fine-grained dependency tracking and resolution, while A_3 can be executed as a data-parallel kernel. 89

5.4 SET scheduling of the hierarchical DAG in Figure 5.3. Each adaptive (A) task executes as a GPU kernel (K) with fine-grained dependency management using active waiting when deemed necessary. SET scheduling ensures forward progress by mapping the user (U) tasks to the worker wavefronts (W). . . . 92

5.5 Concurrency- and locality-aware aggregations of the application DAG in Figure 5.1. The adaptive task granularity is four. 98

5.6 An example of the different DAG classes. The x-axis shows the number of user tasks, while the y-axis represents the DAG levels (critical path). 101

5.7 The performance and overhead of SpTS (push traversal) kernels using the different execution approaches on VEGA-FE. 103

5.8 The performance and overhead of SpTS (pull traversal) kernels using the different execution approaches on VEGA-FE. 104

5.9 The performance and overhead of SpILU0 (pull traversal) using the different execution approaches on VEGA-FE. 105

5.10 The cost of hierarchical DAG transformation using the different task aggregation policies. 106

5.11 The performance of PCG solver on VEGA-FE. 107

6.1 SPICE transient analysis. 112

6.2 An overview of SPICE-H framework. 118

6.3 The trend of the optimal number of partitions, m^*, for $n = 100,000$ circuit equations/unknowns. 122

6.4 SPICE-H generates a hypergraph program representation for adaptive parallel execution of the target application. 122

7.1 MetaMorph uses a modular, layered design to hide the complexity of executing on and interoperating across a range of compute platforms.[1] 133

7.2 MetaMorph provides interoperability both with external libraries in a node (dashed lines) and across nodes with varying hardware (solid lines). 135

7.3 MetaMorph accelerates the development of new operations and kernels. . . . 137

7.4 Overview of the main user API exposed by MetaMorph. 140

7.5 Neighboring processes must exchange boundary data every iteration. 142

7.6 Face description data structure and hierarchical index computation. 143

7.7 MetaMorph's compute APIs that are used to accelerate the target applications.150

7.8 The performance of target applications with the different MetaMorph (MM) back-ends on the experimental cluster. 152

7.9 The runtime distribution of MiniGhost using a 256X256X512 local grid with the different MetaMorph (MM) back-ends on the experimental cluster. . . . 153

7.10 The performance of the 3D dot-product benchmark on the different MetaMorph (MM) backends with platform-specific BLAS libraries on the experimental cluster. 154

7.11 The scalability analysis of MiniGhost on the HokieSpeed cluster with MetaMorph vs. the reference MPI+OpenMP implementation. 155

List of Tables

2.1 Comparison of recent performance prediction tools for heterogeneous HPC architectures (CPUs and GPUs) 16

3.1 The specifications of target hardware architectures. 30

3.2 Rodinia and Parboil workloads. 39

4.1 Target HPC benchmarks and applications. 70

4.2 The size of value communication graph (VCG). 72

5.1 Characteristics of the sparse problems. 100

5.2 Target GPU architectures. 101

6.1 Performance parameters of cg1.4xlarge instances. 114

6.2 Test Circuits 127

6.3 The characteristics of target simulators. 127

6.4 Performance of SPICE-H. 128

7.1 The number of effective code lines changed/added to accelerate the baseline MPI implementation. 150

7.2 The experimental cluster's configurations. 151

7.3 The CPU-GPU cluster's configurations. 151

List of Abbreviations and Acronyms

API	Application Programming Interface.
ATA	Adaptive Task Aggregation.
BLAS	Basic Linear Algebra Subprograms.
BSP	Bulk Synchronous Parallel.
CAD	Computer-Aided Design.
CFD	Computational Fluid Dynamics.
CSC	Compressed Sparse Column.
CSR	Compressed Sparse Row.
CU	Compute Unit.
CUDA	Compute Unified Device Architecture.
DAG	Directed Acyclic Graph.
DSL	Domain Specific Language.
FLOPS	Floating-Point Operations per Second.
GPU	Graphics Processing Unit.

HPC High-Performance Computing.

HSA Heterogeneous System Architecture.

IR Intermediate Representation.

LRU Least Recently Used.

MIC Many Integrated Cores.

MPI Message Passing Interface.

NR Newton-Raphson.

OpenCL Open Computing Language.

OpenMP Open Multi-Processing.

PCG Preconditioned Conjugate Gradient.

PT Persistent Threads.

SET Sorted Eager Task.

SIMD Single Instruction Multiple Data.

SPICE Simulation Program with Integrated Circuit Emphasis.

SpILU Sparse Incomplete Lower-Upper factorization or decomposition.

SPMD Single Program Multiple Data.

SpTS Sparse Triangular Solve.

SSA Static Single Assignment.

VCG Value Communication Graph.

VFC Value-Flow Chain.

Chapter 1

Introduction

The availability of scalable computing resources has been crucial for advancing scientific discovery by empowering researchers to simulate complex physical phenomena and to analyze the multitudes of available data. For decades, high-performance computing (HPC) systems were homogeneous based on clusters of sequential CPUs [157]. With the end of frequency scaling [185], computer architectures have shifted to parallel processing on a single chip using a diverse set of compute resources, memory subsystems, and execution models. As a result, HPC systems evolved to incorporate an array of heterogeneous parallel devices, ranging from general-purpose processors to specialized accelerators. For example, the Summit supercomputer [86] integrates 48 latency-optimized cores, 30,720 throughput-oriented cores, and 3,840 high-density (tensor) cores in each node, delivering billions to quadrillions of floating-point operations per second (FLOPS) with different precision and accuracy characteristics.

1.1 Research Problems

To meet the ever-increasing demand for computing power, end users need to efficiently utilize current (and future) HPC systems by tackling three main challenges: heterogeneity, scalability, and interoperability.

Heterogeneity. The diversity of parallel architectures and execution models has complicated the design and development of high-performance applications, as end users have to decide which architecture, programming language, algorithm, and implementation technique are the most suitable for their applications, which in turn requires significant hardware and software expertise. In addition, extracting optimal performance demands *manual* design-, compile-, and run-time analysis and optimization for each underlying hardware architecture, which consumes significant time and effort. Therefore, while the hardware performance continued to improve according to Moore's Law, as a result of the transistor scaling, user applications typically attain a fraction of the theoretical performance. Furthermore, the transistor scaling is expected to reach the limits of lithography over the next decade [71], effectively ending Moore's Law and leading to the era of extreme heterogeneity [191, 192], where improving the application performance necessitates specialization across the whole software and hardware stack.

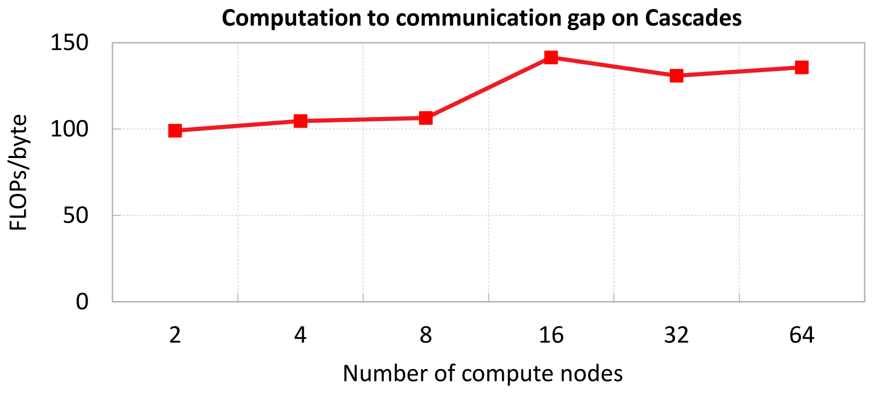


Figure 1.1: The gap between the computation and communication costs on the Cascades cluster at Virginia Tech.

Scalability. HPC systems suffer from an increasing gap between the computation and communication costs [24]. Our experiments demonstrate that the cost of data transfers, both within a node and across nodes, can be orders of magnitude higher than the cost of compute operations [82, 80, 78]. For example, Figure 1.1 shows that the computation to communication gap on Cascades, which is an HPC cluster at Virginia Tech, can be more than two orders of magnitude. Worse, such a gap increases with the number of nodes/processes because of the contention on the shared network and memory bandwidth. As a consequence, the scalability of user applications on HPC systems is limited by the communication cost. Unfortunately, estimating and realizing the minimum communication of a given application is extremely challenging for end users, as it requires comprehensive application and architecture knowledge and demands significant *manual* analysis and optimization. Moreover, in parallel applications, the communication arise from the inherent data dependencies between the computational tasks. Therefore, most efforts to parallelize irregular computations with *fine-grained data dependencies* has had limited scalability [79, 83], although such computations constitute the core kernels in many important application domains, such as computational fluid dynamics (CFD), computer-aided design (CAD), and data analytics.

Interoperability. Parallel architectures are changing faster than parallel programming models and software. Thus, heterogeneous HPC systems require hybrid programming models and various optimization approaches to exploit their potential performance and energy efficiency. Dealing with interoperation between different devices and programming models is a tedious and error-prone task for end users. While many approaches have been proposed to abstract the hardware details, such as directive-based programming models [146, 145] and portable runtime systems [180, 129], they come with a performance penalty or lack performance portability across different platforms [79].

1.2 Thesis Statement

To streamline the transition of scientific applications to the exascale era of heterogeneous billion-way parallelism, this dissertation presents novel program analyses and adaptive execution approaches. The central thesis is: ***graph-structured program representations, which communicate the inherent application characteristics and the complex interdependencies to the execution environment, enable automated runtime analysis and adaptation methods that significantly improve the performance, scalability, and interoperability on heterogeneous HPC systems.***

1.3 Methodology

Figure 1.2 depicts the proposed methodology to enable (1) automated generation of parallel performance and scalability models for sequential codes, (2) scalable execution of compute operations with fine-grained data dependencies, and (3) interoperation and workload distribution across heterogeneous parallel architectures. To bridge the semantic gap between user applications and HPC platforms, we construct *graph-structured program representations* that capture the inherent application characteristics and the complex (many-to-many) interdependencies at multiple levels of computational granularity. These program representations allow the generation of cross-layer abstractions (models) to reason about the performance bottlenecks on heterogeneous parallel architectures, which makes it possible to appropriately map the application kernels to the underlying hardware architectures for maximum performance via our novel frameworks [80, 82, 83, 78, 79] for performance analysis, runtime adaptation, and interoperation.

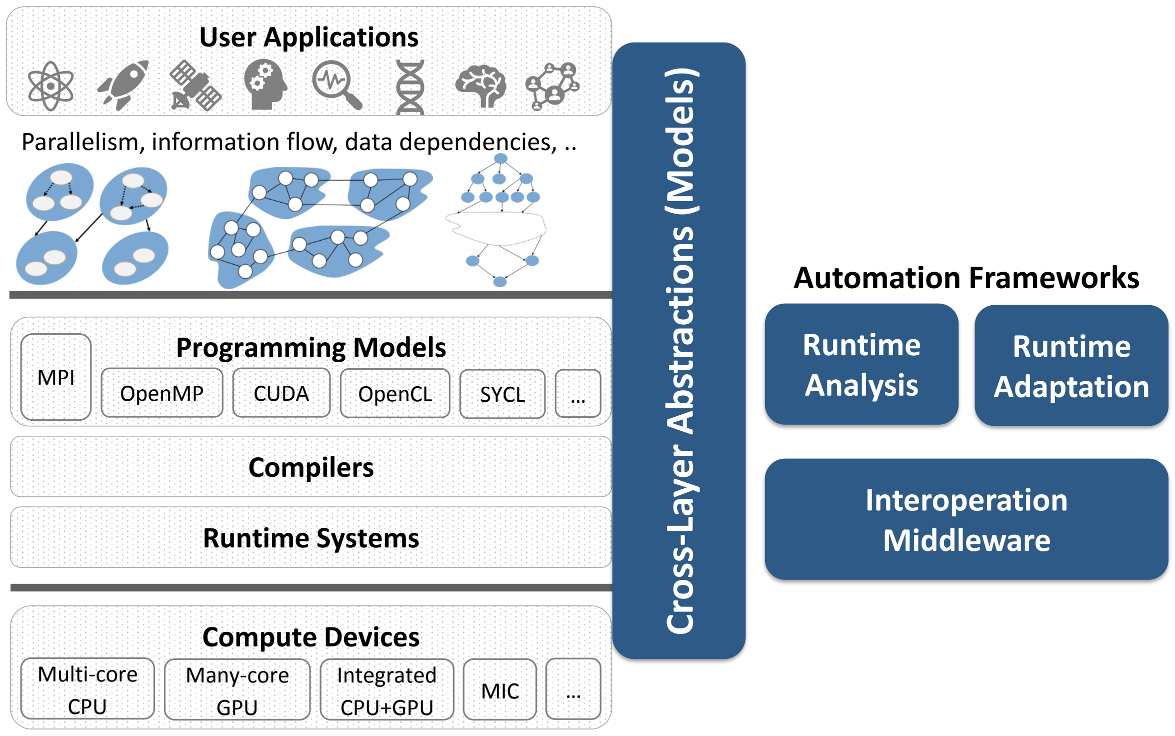


Figure 1.2: Overview of the proposed methodology.

The experimental results demonstrate that the introduced methods can deliver multiplicative performance gains for real-world applications, even when such applications suffer from limited parallelism and fine-grained data dependencies. Due to the ubiquity of heterogeneous computing, the proposed runtime analysis and adaptation approaches are applicable not only to the target HPC platforms, but also to embedded and mobile computing systems.

1.4 Contributions

This dissertation tackles the heterogeneity, scalability, and interoperability challenges of modern HPC systems via innovative approaches for automated program analysis and runtime adaptation. To this end, we demonstrate that graph-structured program representations are powerful abstractions to reason about the performance bottlenecks not only on a single HPC node, but also on multi-node HPC systems. Figure 1.3 presents a summary of the research artifacts and contributions, which are detailed below.

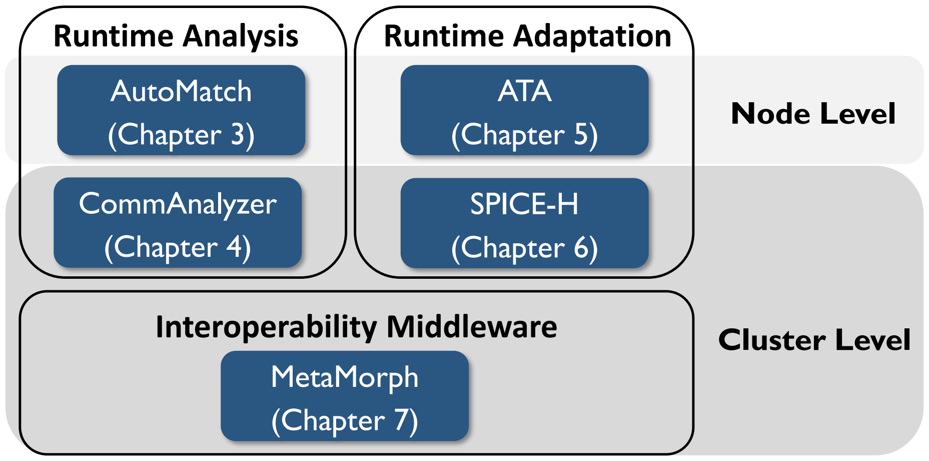


Figure 1.3: Dissertation overview.

1.4.1 Automated Program Analysis for Performance Modeling

Automated Estimation of Execution Cost on Heterogeneous Architectures

Projecting the relative performance of a given code across different types of HPC architectures plays a critical role in the design and development of high-performance applications.

Unlike the previous “black-box” approaches that rely on the performance of a training set of parallel applications, the proposed AutoMatch framework [80] automatically projects the relative *parallel* performance of *sequential* codes across multi-core CPUs and many-core GPUs. Specifically, AutoMatch constructs *a graph-structured program representation at the instruction level* and uses a hybrid (static and dynamic) analysis to estimate the best dependency-preserving parallel schedule of target programs. As a result, AutoMatch enables a runtime workload distribution scheme that simultaneously utilizes the different hardware architectures within an HPC node in a performance-proportional way, i.e., the architecture with higher performance is assigned more workload. For a set of open-source HPC applications with different characteristics, AutoMatch turns out to be very effective, identifying the performance upper-bound of sequential applications across five different HPC architectures. In addition, AutoMatch’s workload distribution scheme achieves approximately 90% of the performance of the profiling-driven oracle [80]. Furthermore, AutoMatch indicates that the predicted speedup by the previous “black-box” performance modeling approaches can be misleading, due to the lack of a diverse set of reference training applications along with their *equally-optimized* parallel implementations for each architecture type.

Automated Estimation of Communication Cost and Scalability

The proposed CommAnalyzer framework [82] is a novel and automated approach for estimating the communication cost of sequential codes when ported to HPC clusters, which makes it possible to project the scalability upper-bound of the effective distributed-memory implementation before even developing one. CommAnalyzer instruments the sequential code to precisely capture the inherent flow of program values (information) through multiple levels of access indirection to construct a *value communication graph (VCG).* Next, it uses graph analytic algorithms to project the communication cost of VCG on distributed-memory models. Thus, CommAnalyzer is applicable for both regular and irregular problems and works also for programs that cannot be auto-parallelized. The experiments with real-world HPC applications demonstrate the utility of CommAnalyzer in estimating the minimum communication cost with more than 95% accuracy on average [82]. As a result, CommAnalyzer enables the development of optimized distributed-memory implementations that realize the estimated communication and scalability bounds [82].

1.4.2 Runtime Adaptation for Scalable Performance

While regular applications (e.g., dense linear algebra and structured grids) have demonstrated scalable speedup in the heterogeneous computing era, irregular applications still suffer from limited scalability due to the inherent synchronization and communication cost. Thus, the HPC community is moving to use irregular benchmarks for the procurement and ranking of HPC systems [59, 134]. This dissertation proposes runtime adaptation approaches for efficient parallel execution of irregular computations with fine-grained data dependencies. Such workloads are extremely challenging for acceleration on heterogeneous HPC systems as the dispatch, scheduling, communication, and synchronization cost becomes the dominant performance bottleneck compared to the execution cost of compute operations.

Efficient Dependency Management on Massively Data-Parallel Architectures

Many-core architectures, such as GPUs, have a massive number of throughput-oriented compute units that are primarily designed to support data-parallel execution. Consequently, irregular computations with fine-grained data dependencies suffer from limited performance on these architectures, due to the substantial overhead required to schedule the computations and to manage their data dependencies. We propose the adaptive task aggregation (ATA) framework [83] to efficiently execute such irregular computations on massively data-parallel architectures. ATA represents the data-dependent computations as *a hierarchical directed acyclic graph (DAG)*, where nodes are multi-grained application tasks and edges are their aggregated data dependencies. Unlike previous approaches, ATA is aware of the dependency structure of input problems and the processing overhead on target architectures. Therefore, it adapts the parallel execution at runtime by selecting the appropriate task granularity and by managing data dependencies at multiple execution levels. On a gamut of representative problems with different data-dependency structures, ATA significantly outperforms the existing GPU task-execution approaches, achieving a geometric mean speedup of 2.2× to 3.7× across important sparse solver kernels and delivering up to an order-of-magnitude improvement in the end-to-end application performance [83].

Scalable Execution of Irregular Data-Dependent Computations

We propose SPICE-H [78], a heterogeneous distributed-memory framework for a representative irregular application with limited and data-dependent parallelism, namely transistor-level circuit simulation [135]. The target application has been traditionally restricted to a single homogenous architecture because of the sequential dependencies between its fine-grained compute operations. SPICE-H constructs a *hypergraph program representation at the application level* to drive compensation-based algorithms for program transformation that greatly reduce the communication and synchronization overhead as well as increase the available parallelism. Unlike prior work, our framework uses model-driven runtime analysis and adaptation to (1) find the optimal workload partitioning and load balancing, and (2) map the partitioned kernels to the most suitable architecture that matches their characteristics. The experimental results on Amazon EC2 heterogeneous cloud demonstrate an order-of-magnitude speedup compared to the optimized multi-threaded implementations [78].

1.4.3 Interoperability Middleware for Simultaneous Execution

Parallel architectures are changing more rapidly than parallel programming models, and until now, there has been no universal programming model that provides both functional portability and performance portability. We present MetaMorph [79], a lightweight middleware that resides between user applications and hardware platforms to allow interoperability and workload distribution across heterogeneous compute resources.

MetaMorph addresses the challenges of programming heterogeneous architectures through its core design principles: abstraction, interoperability, and adaptability. First, it provides high performance and abstraction using a highly-optimized back-end layer and a light-weight interface layer that does not require significant application re-factoring. Second, MetaMorph supports interoperability across different hardware accelerators and with existing code. Most importantly, MetaMorph is designed to exploit runtime information to improve the performance even more by providing access to all hardware devices present in a system to enable simultaneous execution and workload distribution. The experimental evaluation shows that MetaMorph reduces the development time, while delivering performance and interoperation across an array of heterogeneous devices, including multi-core CPUs, Intel Many Integrated Cores (MICs), AMD GPUs, and NVIDIA GPUs. In particular, MetaMorph enables a simultaneous execution approach that effectively uses all the heterogeneous resources in the target HPC systems [79].

1.5 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 presents the background and related work. Chapters 3 and 4 detail our research on projecting the parallel performance and scalability of sequential codes. Chapter 5 describes our runtime adaptation framework for dependency management on massively parallel GPU architectures. Chapter 6 attacks the problem of scaling irregular data-dependent computations on heterogeneous HPC clouds. Chapter 7 presents our middleware framework which provides interoperation and simultaneous access to the available compute resources in an HPC system. Finally, Chapter 8 summarizes the dissertation and outlines the future research opportunities.

Chapter 2

Background and Related Work

2.1 HPC Programming Models

With the end of frequency scaling and the shift to parallel and heterogeneous computing, developing software has become much more complex [185]. While the Message Passing Interface (MPI) remains the de facto standard for programming distributed-memory systems [56, 20], many approaches have been proposed to address the challenges of programming shared-memory parallel architectures by abstracting their hardware details. This section discusses the related parallel programming models and runtime systems.

Directive-based programming models, such as OpenMP (Open Multi-Processing) [146] and OpenACC (Open Accelerator) [145], move the burden of explicit thread management, workload partitioning and scheduling, data movement across the memory hierarchy, and inter-thread communication and synchronization to the compiler. While OpenMP and OpenACC abstract away complex details and provide a convenient interface to describe the parallelism to the compiler, achieving acceptable performance requires deep understanding of the underlying hardware architecture, runtime system, compiler limitations, and a number of complex directive specifications. Moreover, the programmer must use thread-safe functions, eliminate inter-thread data dependencies, avoid pointer aliasing, and manage access to shared variables and data structures. In addition, the high-level abstraction of directive-based programming can come with a performance penalty [199, 196, 95, 50] compared to low-level programming models, such as OpenCL (Open Computing Language) [133] and CUDA (Compute Unified Device Architecture) [141].

Portable runtime systems, such as OpenCL [180] and OCCA [129], provide a kernel specification language and a runtime compilation and execution environment on multiple heterogeneous platforms. Although these approaches support functional portability, performance portability is not guaranteed, as the application developers need to modify and optimize the kernel implementation for each target hardware platform to achieve the required performance. Moreover, these approaches have relatively lower programmability in comparison with directive-based programming, as the programmer must explicitly manage all the hardware control operations, such as memory allocation, data transfer, thread creation/destruction, and synchronization.

Domain-specific libraries provide both abstraction and high performance for a set of kernels and algorithms in a target domain. However, this often comes with the cost of complicated installations and extensive application refactoring. For example, MAGMA [58] provides highly optimized and intelligently scheduled BLAS (Basic Linear Algebra Subprograms) kernels, but due to the dependency on external libraries is difficult to install, configure, and tune, and it does not yet provide unified or consistent capability across its many-core implementations. Although MAGMA supports multi-GPU BLAS kernels, there is no built-in support for interoperability across different hardware accelerators. PARALUTION [124] and ViennaCL [164] provide iterative solvers and preconditioners for many important application domains, such as computational fluid dynamics (CFD), and support various multi-core and many-core architectures. However, these solver frameworks require recasting the application to use complex cases and object types, which presents a barrier to the incremental porting of user applications.

In summary, directive-based programming models trade performance with high-level abstraction, while low-level languages and portable runtime systems can achieve high performance with the cost of low programmability and explicit hardware control. Domain-specific libraries can achieve high-level abstraction and high performance, but only for a set of algorithms in a specific domain and with a significant application re-factoring, complicated installation, and steep learning curve.

2.2 Massively Parallel HPC Architectures

Here, we use graphics processing units (GPUs), which are the most popular accelerators in modern HPC systems [130], and the OpenCL terminology to describe massively parallel architectures and their execution models.

Figure 2.1 depicts the recent VEGA GPU architecture from AMD [10], which consists of multiple compute units (CUs) organized into shader engines (SEs). Each CU contains single-instruction, multiple-data (SIMD) processing elements. SEs share global memory and level-2 (L2) cache, while CUs have their own dedicated local memory and level-1 (L1) cache. At runtime, the control/command processor (CP) dispatches the workload (kernels) to the available SEs and their CUs.

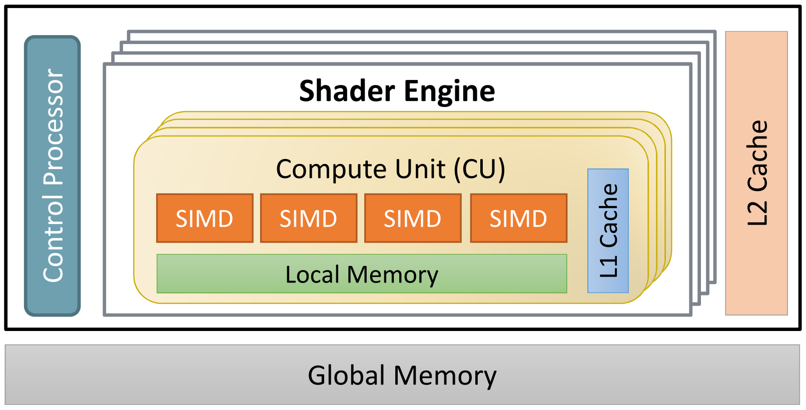


Figure 2.1: VEGA GPU architecture.

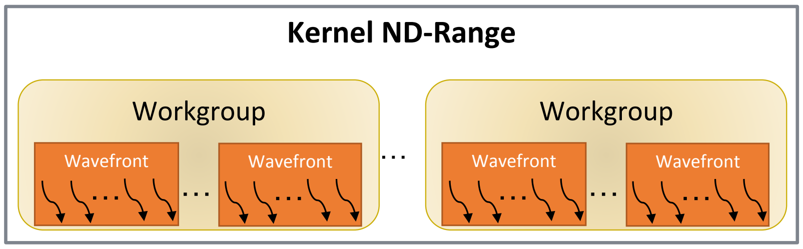


Figure 2.2: GPU kernel.

Like GPU hardware, GPU kernels have a hierarchical thread organization, as shown in Figure 2.2, consisting of workgroups of multiple wavefronts. The SIMD elements execute each wavefront in lockstep; thus, wavefronts are the basic scheduling units.

Massively parallel GPUs provide fundamental support for the bulk synchronous parallel (BSP) execution model [189], where the computations proceed in data-parallel supersteps. Figure 2.3 depicts a BSP superstep which consists of three phases: local computations on each CU, global communication (data exchange) via main memory, and barrier synchronization.

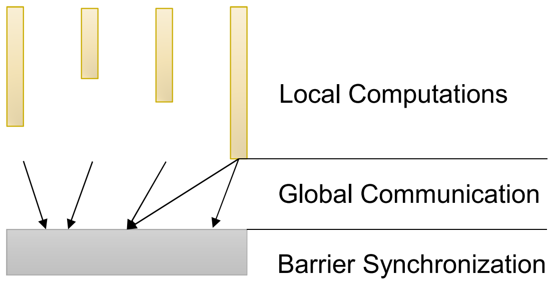


Figure 2.3: The execution of a BSP superstep.

In BSP execution, the computations in each superstep must be independent and can be executed in any order. To improve workload balance, each CU should perform a similar amount of operations. Moreover, GPUs require massive computations in each superstep to utilize the available compute resources and to hide the long memory-access latency. Due to these limitations, the efficient BSP execution of irregular applications with variable and data-dependent parallelism is challenging.

2.3 Performance Modeling on HPC Systems

There are many approaches and substantial prior work for performance modeling on HPC systems that can be classified into four categories: analytical modeling, simulation, automated performance prediction, and communication and scalability analysis.

2.3.1 Analytical Modeling

Analytical modeling maps the application and execution environment into a set of parameters and mathematical expressions that can be evaluated to predict important performance metrics, such as the execution time. While analytical modeling provides useful insights into the application performance by capturing the complex interactions between the HPC platform, application, and input data [200, 92], it requires tedious manual analysis of both the target applications and hardware architectures. Typically, there is a tradeoff between the accuracy of an analytical model and the number of its parameters. While complex performance models have higher accuracy levels, they require extensive analysis and are harder to generalize to different types of hardware architectures. The Roofline model [200] and LogP model family [47, 7] are popular modeling approaches that provide a *high-level* view of the application performance using a few parameters; however, they abstract away critical factors, such as the parallelism, data locality, and synchronization overhead.

Empirical models [92, 30] use profiling and statistical analysis to calculate the critical coefficients that are hard to derive from manual analysis. Therefore, while such models can predict the actual performance of an application on a particular architecture with high accuracy, they require the availability of both the target hardware and optimized parallel code.

To explore the algorithmic and architectural design space, Spafford and Vetter present ASPEN [176], a language for describing formal application and machine models. PALM [186] simplifies the performance modeling process by supporting the generation of standard analytical models from annotated source code. Snavely et al. [175] provide a framework to generate analytical performance models from the execution and communication traces of HPC applications and the abstract machine profiles.

2.3.2 Simulation

Hardware simulators, such as Sniper [31] and gem5 [25, 153], can provide accurate performance prediction and detailed information about the application behavior without the availability of target hardware architectures. However, they need the parallel application code and typically consume significant time and compute/memory resources. In addition, the predicted performance depends on the ability of end users to prallelize and optimize the application to the simulated architecture. Similarly, distributed-memory simulators (e.g., LogGOPSim [91] and DIMEMAS [160]) incorporate detailed network and architecture models to estimate the communication time and performance from the compute/communication traces of parallel code. MUSA [73] adopts a multi-level simulation approach with different levels of hardware details, simulation cost, and simulation accuracy. In addition, it identifies and simulates the representative application phases to reduce the overall simulation time.

2.3.3 Automated Performance Prediction

Recently, several tools have been proposed to automate the performance modeling and prediction using program analysis and machine-learning. Table 2.1 summarizes the comparison of the state-of-art performance prediction tools for heterogeneous HPC architectures.

COMPASS [114] generates a structured performance model from the parallel application code using static analysis. However, the user must annotate the code to indicate the available parallelism and required data movement to generate an accurate model. Otherwise, COMPASS may generate a conservative parallelism profile, due to the difficulty of alias analyses [111, 156]. Therefore, COMPASS does not work well for irregular applications whose computation and memory access patterns are data-dependent.

Table 2.1: Comparison of recent performance prediction tools for heterogeneous HPC architectures (CPUs and GPUs)

	COMPASS [114]	XAPP [13]	**AutoMatch**
Input code	Annotated	Sequential	Sequential
Features extraction	Static analysis	Dynamic analysis	Hybrid analysis
Machine model generation	By users	Training data	Micro-benchmarking
Performance modeling	ASPEN model	Machine-learning	Execution Cost model
Cache-awareness	No	Yes	Yes
Usability	Low	High	High
Application generality	Low	High	High
HW generality	High	Low	High
The tool speed	Fast	Slow	Moderate

XAPP [13] uses machine-learning (ML) to find the correlation between the CPU execution profile of the application and the GPU speedup. Therefore, XAPP is heavily influenced by the training data, and its prediction accuracy depends on the availability of a diverse set of applications along with their optimized GPU implementations. So, extending XAPP to new architecture types requires huge effort to rewrite and re-optimize each training application to the target architectures. Moreover, to predict the performance on a specific GPU device, the user needs to run all the training applications on this device, which takes days. Such long-running model generation of ML-based tools end up being orders of magnitude slower than our AutoMatch framework, which generates the device parameters using micro-benchmarks that takes few minutes. In addition, XAPP's predicted speedup is not the speedup upper-bound, and it depends on which optimization techniques are applied in the training applications.

Kismet [97] predicts the potential speedup of sequential applications on multi-core processors. It instruments the code to build the self-parallelism profile, and estimates the memory access latency by profiling the application on a CPU cache simulator. Kismet optimistically assumes that the memory bandwidth is scalable with the number of threads, which is unrealistic assumption especially for massively parallel architectures such as GPUs. Therefore, its predicted speedup is unattainable at higher core counts and for memory-bound workloads. As an alternative, Parallel Prophet [107] predicts the speedup of the annotated code on multi-core CPUs. Unlike Kismet, it does not require parallelism discovery, but relies on user annotations to identify the available parallelism. To build the performance model, Parallel Prophet collects architectural parameters such as instruction counts and cache misses through hardware performance counters, which requires the availability of the target CPUs and the parallel (or annotated) code.

2.3.4 Communication and Scalability Analysis

Communication Cost Estimation

Several approaches have been proposed for estimating the communication cost of sequential codes to enable code generation for distributed-memory platforms [87, 201, 77]. The FORTRAN-D compiler [87] uses static data-dependence analysis to detect the communication between different sections/parts of a data array, which is partitioned according to a user-specified decomposition technique. However, this approach suffers from communication overestimation, and it is limited to regular applications. Similarly, Gupta et al. [77] adopt a compile-time, data-dependence analysis for estimating the communication of sequential codes, which is only applicable for regular problems with array-based data structures. The SUIF compiler [201] solves the communication overestimation problem using the exact static data-flow analysis, but it is limited to affine loop nests with regular memory access pattern, where dependencies between loop iterations and memory locations can be detected at compile time. In contrast, our CommAnalyzer framework predicts the communication between program values regardless of the underlying data structures and without any user-specified decomposition techniques using a combination of novel dynamic analyses and graph analytics. Thus, CommAnalyzer is applicable to a wide range of regular and irregular applications.

While the above approaches estimate the communication cost using the single program, multiple data (SPMD) execution model, which is by far the dominant approach on HPC systems [56, 20], Bondhugula [28] presents a polyhedral framework to estimate the communication when there is no fixed data ownership, i.e., the data can move between compute nodes according to the distribution of the computations and data dependencies. However, this approach is limited to regular applications with affine loop nests as well.

Communication Pattern Detection

Identifying the communication patterns of HPC applications is helpful in the design-space exploration of both the system architectures and parallel algorithms. Several tools [163, 105] have been proposed for the detection of MPI communication patterns by finding a match between the actual inter-process communication and a set of known communication templates/patterns. These tools instrument the MPI implementation to capture the inter-process communication and to generate the communication matrix across MPI processes, and then they recognize existing communication patterns in the communication matrix. In particular, AChax [163] can identify multiple communication patterns (such as nearest neighbor, sweep/wavefront, broadcast, and reduction) in the communication matrix and generate a parametrized communication model for the actual communication based on a combination of the detected communication patterns.

Scalability Analysis

The scalability analysis tools [18, 209, 29, 33] project the performance of a given MPI implementation at a massive scale based on small-scale experiments. Typically, these tools extract the communication, computation, and/or memory traces of MPI applications using dynamic instrumentation and profiling. Therefore, they require the distributed-memory parallel implementations and at least a single node of the target cluster to predict the performance on multiple nodes. TAU [173] and HPC toolkit [3] are integrated frameworks for portable performance analysis, profiling, and visualization. Similar to scalability analysis tools, their main goal is to simplify the performance analysis and diagnosis of the existing MPI code, rather than estimating the potential performance at a large scale before investing effort and time in developing the distributed-memory implementation of the original applications.

2.4 Data Dependency Management on Massively Parallel HPC Architectures

There is a large body of prior work on optimizing irregular applications for massively parallel architectures. This section focuses on irregular applications that suffer from *fine-grained data dependencies*, which limit their parallel performance and scalability on these architectures. Such applications are naturally represented as directed acyclic graphs (DAGs), where nodes are compute tasks and edges are data dependencies across tasks, to expose parallel tasks.

Researchers have designed many software and hardware frameworks to improve the performance of *data-dependent* computations on massively *data-parallel* architectures, such as many-core GPUs. In the following, we discuss the state-of-the-art dependency management frameworks.

Juggler [23] is a data dependency management scheme for GPUs that maintains task queues on the different compute units and employs persistent workers (workgroups) to execute ready tasks and to resolve the data dependencies of waiting tasks. Other frameworks [178, 179, 210] adopt a similar execution model with persistent threads (PT) on GPUs. PT execution significantly reduces GPU resource utilization and limits the latency hiding ability of GPU hardware schedulers. While GPUs require massive multithreading to hide the execution and memory access latencies [66, 161, 10], PT execution only runs one worker per compute unit. Therefore, such frameworks typically achieve a limited performance improvement compared to the traditional data-parallel execution (for example, 1.05 to 1.30-fold speedup [23]) with portability issues across different GPU devices. Conversely, our ATA framework executes multiple workers per compute unit to maximize the utilization of GPU resources and to expose the inherent parallelism of user applications. Moreover, achieving workload balance using distributed task queues is difficult and requires significant processing overhead. As a result, PT execution approaches typically execute user tasks at the granularity of workgroups. In contrast, ATA leverages the existing hardware schedulers for GPUs, which perform dynamic resource management across active wavefronts, to reduce the idle/waiting time by concurrently executing the available tasks in user applications and mapping them to active wavefronts. Furthermore, ATA can support a wide range of granularity, ranging from wavefronts to workgroups.

Alternatively, several runtime systems for task management [15, 21, 67, 154] support irregular applications by launching each user task as a parallel kernel and by using device streams or queues to manage data dependencies across kernels. In such systems, the task launch overhead is on the order of microseconds, and the dependency management using streams (queues) only supports a finite number of pending dependencies. Thus, these approaches are limited to irregular applications with coarse-grained tasks, where the dependency management overhead is a fraction of the overall execution time. Specifically, StarPU [15] and Legion [21] schedule the data-dependent computations on heterogeneous architectures with multiple CPUs and GPUs, and consider a single device (CPU or GPU) as a worker. Furthermore, these runtime systems manage data dependencies on the host, which introduces significant host-device communication and synchronization overhead.

Pagoda [207] and GeMTC [109] adopt a centralized scheduling approach to execute independent application tasks on GPUs using a resident kernel, which distributes ready tasks to compute units at the wavefront granularity. However, these frameworks assume that all the dispatched tasks are ready for execution and do not support dependency tracking and resolution. Specifically, they rely on the host to send ready tasks to the GPU after their dependencies are resolved. Therefore, Pagoda and GeMTC suffer from host-device communication which is limited by the PCI-E bandwidth.

Prior software systems [62, 37, 187, 61] improve the performance of dynamic parallelism, where a GPU kernel can launch child kernels, by aggregating the *independent* work-items across child kernels to amortize the kernel launch overhead; however, these techniques are not suitable to execute application DAGs with many-to-many relationships (dependencies) between the predecessor and successor tasks. Conversely, in our ATA framework, work aggregation is used to execute irregular applications with *data-dependent* tasks both within and across GPU kernels, which requires efficient mechanisms to track the progress of active tasks and to determine when waiting tasks are ready for execution. Hence, ATA aggregates *data-dependent* work across user tasks with a strict partial execution order and then enforces this order using hierarchical dependency management and low-latency task scheduling.

Alternatively, hardware approaches [2, 147, 194, 195] aggregate and execute data-dependent computations on many-core GPUs using dedicated hardware units or specialized workgroup (thread-block) schedulers. Unlike these approaches, the proposed ATA framework works on current GPU architectures without needing special hardware modifications. In addition, it supports a finer task granularity than workgroups.

Chapter 3

Automated Estimation of Execution Cost on Heterogeneous Architectures

3.1 Introduction

With the end of Dennard scaling, the performance of sequential CPUs hit the power wall, thus making it hard to improve the performance by increasing the clock frequency [185]. To meet the ever-increasing demand for computing performance, driven by the multitude of data sets, computer architectures have shifted to parallel processing. However, unlike the sequential computing era, there is no de facto standard for hardware acceleration. Instead, the parallel architecture landscape is in flux as new platforms are emerging to meet the needs of new workloads. Therefore, current (and future) HPC systems contain a wide variety of heterogeneous computing resources, ranging from general-purpose CPUs to specialized accelerators, due to both the diversity of the computational kernels in the applications and the lack of a single architecture meeting all of their requirements [24].

Porting sequential applications to heterogeneous HPC systems for achieving high performance requires extensive software and hardware expertise to *manually* analyze the target architectures and applications not only to estimate the potential speedup, but also to make efficient use of all the different compute resources in these systems. To streamline such a daunting task, end users need appropriate tools and frameworks to automatically predict the potential application performance on heterogeneous HPC systems. Therefore, researchers have created several tools classified into two categories: automated performance modeling and machine-learning performance prediction.

Automated performance modeling tools [97, 107, 114] use static and/or dynamic analysis to construct a performance model of the target application and hardware architecture, which is then used to predict the potential speedup. However, these tools are either limited to homogenous HPC systems, with traditional multi-core processors, or require code annotations to indicate the available parallelism and data movement, which might not be possible for non-expert users. In addition, tools based only on static analysis do not work well for irregular applications, whose computation and memory access patterns are data-dependent, due to the difficulty of alias analysis [111, 156].

On the other hand, machine-learning performance prediction tools [13, 17] adopt a "black-box" approach and are heavily influenced by the training data. Thus, their prediction accuracy depends on the availability of a diverse set of training applications along with their *optimized parallel* implementations for every target architecture, which is often hard to find [115]. Apart from that, the predicted speedup is not the speedup upper-bound, and even worse it depends on which optimization techniques are applied to the reference heterogeneous implementations of training applications. Unfortunately, open-source heterogeneous applications and benchmark suites are usually *not equally-optimized* for each architecture type [115, 181]. For this reason, machine-learning approaches may not be suitable to predict the relative performance between different architecture types despite the required expertise and effort to collect the ideal training set and the significant time for training. Given all this, there is a compelling need for a more practical approach to serve a wide range of the users of HPC systems.

3.1.1 AutoMatch: The First-Order Framework

This chapter presents AutoMatch, an automated framework for matching of computational kernels to heterogeneous HPC architectures. Figure 3.1 shows the proposed framework which analyzes a given *sequential* application code and constructs *a graph-structured program representation at the instruction level* to estimate the benefits of porting such an application to heterogeneous HPC systems.

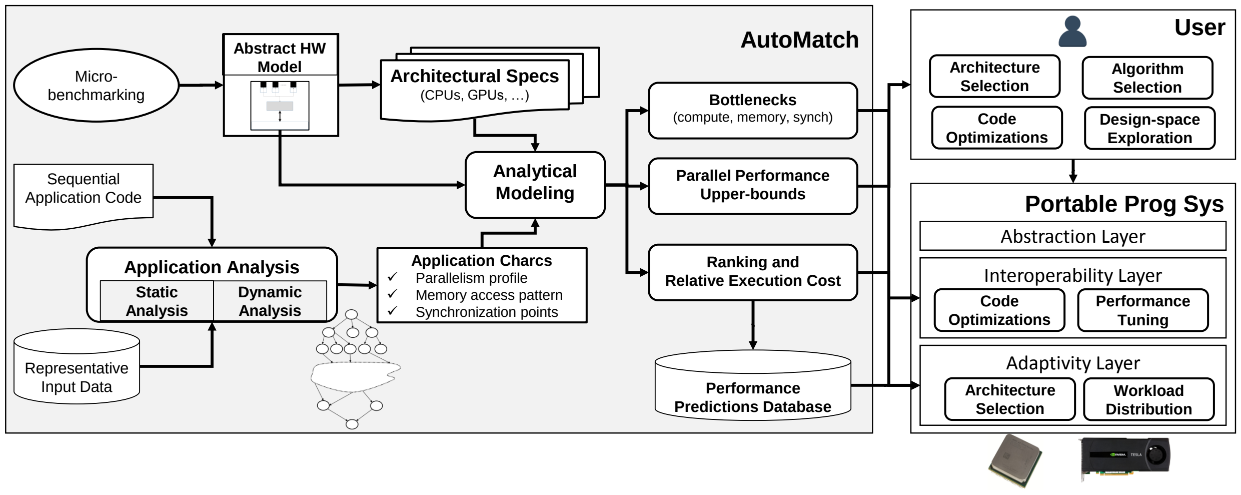


Figure 3.1: Overview of AutoMatch framework.

First, AutoMatch generates the architectural specifications via micro-benchmarking to instantiate an abstract hardware model for each architecture in the target heterogeneous system. Second, it leverages compiler-based static and dynamic analysis techniques to quantify the maximum parallelism, the maximum data locality, and the minimum synchronization of the sequential code for estimating the *upper bounds* of the parallel performance on the different architectures. Third, AutoMatch generates *high-level* analytical models which combine the abstract hardware model, application characteristics, and architectural specifications to predict the potential parallel performance on *different types* of hardware devices. This performance prediction is then used to estimate the relative execution cost across a set of different architectures, including multi-core CPUs and many-core GPUs, thereby driving a workload distribution scheme, which enables end users to efficiently exploit the available heterogeneous devices in an HPC system.

It is important to note that AutoMatch is designed as a *first-order* framework for users to estimate the potential parallel performance of their sequential applications on heterogeneous HPC systems in the early stages of the development process, i.e., without having to pay the high cost of developing the optimized parallel code (or painfully collecting training data) for every target architecture. While our automatically-generated models are simple, they work well for predicting the relative performance across different architectures and the best workload distribution strategy.

Use Cases

AutoMatch accelerates the development process of user applications and supports the emerging programming systems for performance portability and interoperability across different accelerators (such as MetaMorph [79], which is discussed in Chapter 7).

Architecture Selection. AutoMatch predicts the relative ranking of parallel heterogeneous architectures for sequential application codes. It serves not only those who either have not determined the target device or cannot afford to buy multiple candidate devices, especially when the application and inputs are often changed, but also those who lack enough expertise to develop the optimized parallel implementation for each architecture type. Furthermore, AutoMatch's ranking of target hardware architectures enables the adaptivity layer of portable programming systems to select the best performing architecture at runtime.

Algorithm Selection. AutoMatch predicts the upper bounds on parallel performance of a given sequential application code by finding the best dependency-preserving schedule of its operations, which performs the same operations as the sequential algorithm but in a different order. If AutoMatch's prediction of the original algorithm is already good enough, the user can save the time to consider other algorithms. Conversely, if the predicted performance is unsatisfactory, it motivates the user to explore/develop different algorithms. Nevertheless, AutoMatch can still play a critical role even for this case. The user can analyze the sequential code of different algorithms using AutoMatch to estimate the upper bounds on their parallel performance beforehand without developing the optimized parallel code(s).

Code Optimizations. AutoMatch provides detailed information about the inherent parallelism, data locality, and performance bottlenecks of the sequential application code to help users to decide on the best parallelization strategy and optimization techniques for their applications. Moreover, since it is often hard to find reference applications that are *equally-optimized* for each architecture type [115], AutoMatch's prediction of the performance upper bounds serve as a reference to show how close the parallel implementation is to the best possible performance. That way, AutoMatch can guide not only manual code optimization, but also the customization/tuning of the different backends of portable programming systems.

Workload Distribution. AutoMatch's estimation of the relative performance (execution cost) on heterogeneous systems promotes the development of a runtime workload distribution on top of programming systems that support the seamless execution of parallel applications on multiple heterogeneous devices (e.g., MetaMorph [79]) to efficiently exploit the available compute resources across these devices.

Design-Space Exploration. Even if the applications and/or the architectures are not yet available, AutoMatch can still be used with synthetic application features and/or architectural parameters to automatically explore the design space. Detailed discussion of the architecture selection, code optimizations, and workload distribution is provided in Section 3.3, while a study of the algorithm selection and design-space exploration is reserved for future work.

3.1.2 Contributions

Unlike the previous approaches, AutoMatch does *not* require the availability of the target platforms or the parallel application code for each platform, thereby expanding the range of users. In addition, it is automated and applicable to different types of hardware architectures with minimal effort, i.e., generating the architecture specifications. With the automated and repeatable methodology of AutoMatch, users can easily adapt it to future heterogeneous architectures. In summary, the following are the contributions of this work:

- We propose an automated framework (AutoMatch) that constructs a fine-grained, graph-structured program representation and uses a combination of compiler-based analysis, micro-benchmarking, and analytical modeling to estimate (1) the performance upper bounds of sequential applications on heterogeneous HPC systems, and (2) the relative ranking and performance of the architecture alternatives in such systems (§3.2).

- AutoMatch's estimation of the relative performance across the heterogeneous architectures enables a workload distribution runtime that simultaneously utilizes them all in a performance-proportional way, i.e., the architecture with higher performance is assigned more workload (§3.2).

- Using a diverse set of open-source HPC applications, with different parallelism profiles and memory-access patterns, we show the efficacy of the proposed white-box framework across different HPC architectures (§3.3).

- We present case studies on both regular and irregular workloads to pinpoint the issues with the black-box performance prediction approaches, e.g., profiling and machine-learning. Since they rely on the performance of a training set of parallel applications, unlike the proposed AutoMatch framework, their results can be fooled by the heterogeneous implementations that are *not equally-optimized* for each target architecture (§3.3).

3.2 AutoMatch Approach

3.2.1 Hardware Architecture Model

This work proposes an abstract hardware architecture model that can be generalized to different shared-memory architectures, including multi-core CPUs and many-core GPUs. The proposed model extends the classical external memory model [4, 49] to parallel architectures and considers important constraints on these systems, such as the on-chip memory access time and the synchronization overhead.

Figure 3.2 shows the hardware architecture model comprised of multiple compute cores that share a fast on-chip memory connected to a slow off-chip memory. The compute cores can only perform operations on data in their private on-chip memory, and each core executes floating-point operations at a peak rate of π_0 FLOPs per second. The floating-point throughput is $\Pi = n_p \times \pi_0$, where n_p is the number of compute cores. The fast on-chip shared memory is fully associative with a size of Z words, and it uses the least recently used (LRU) replacement policy. The data is transferred between the compute cores, the fast memory, and the slow memory in messages of L words. The fast on-chip shared memory has a latency α_f and a bandwidth β_f, while the slow off-chip memory has a latency α_s and a bandwidth β_s.

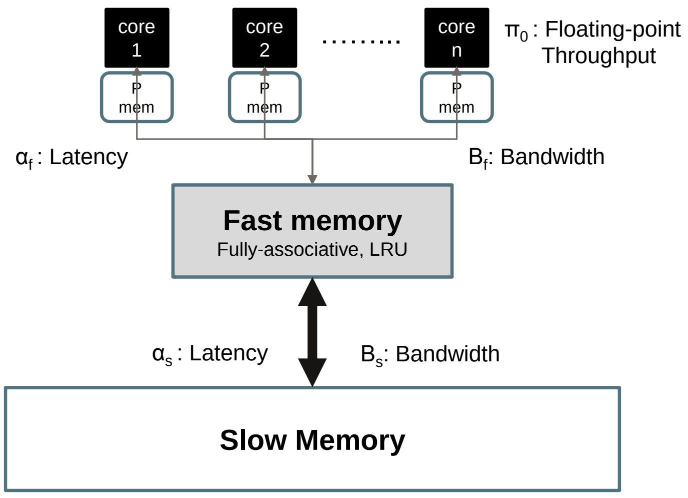


Figure 3.2: The abstract hardware architecture.

To reach a globally consistent memory state, the compute cores perform synchronization operations whose cost depends on the memory latency and the number of compute cores. Since the synchronization overhead, s_0, significantly affects the execution time on parallel architectures, especially at higher core counts [206, 65], the proposed model considers this overhead. There are two synchronization types: (1) global synchronization, between coarse-grained threads with different control units (threads on CPUs and workgroups/thread-blocks on GPUs), and (2) local synchronization, between fine-grained threads with shared control units (SIMD lanes on CPUs and threads on GPUs). Given that local synchronization overhead is negligible compared to global synchronization (usually by at least an order of magnitude) [206, 65], the proposed model ignores it.

Note, the main goal is to match the workloads to the best architecture from a set of parallel architectures that are fundamentally different, for which our proposed high-level hardware model works well. In light of this, the model abstracts away architecture-specific parameters and low-level hardware details, e.g., hardware prefetchers and complex memory hierarchies. Similarly, it ignores one-time cost overheads, such as thread creation/destruction, kernel launching, and host-device data exchanges, which are highly-dependent on the runtime environment and the system/expansion bus rather than the target architectures.

3.2.2 Inferring the Architectural Specifications

AutoMatch figures out the specifications of hardware architectures using micro-benchmarking. In particular, it uses the ERT [121], pointer-chasing [128, 202], and synchronization [206, 78] micro-benchmarks to estimate the floating-point throughput and memory bandwidth, the memory access latency, and the global synchronization overhead, respectively. To analyze the effectiveness of AutoMatch, this work uses five heterogeneous architectures (two CPUs and three GPUs) with different core counts and considers three subsets of architectures: ($ARC1$, $ARC3$, $ARC5$), ($ARC1$, $ARC2$), and ($ARC4$, $ARC5$). The first subset contains three significantly different architectures with few cores, hundreds of cores, and thousands of cores, while the second and third subsets have two slightly different CPUs and GPUs, respectively. Table 3.1 summarizes the specifications of target heterogeneous architectures.

Since modern on-chip memories support the inclusion property in their hierarchical organization [46], AutoMatch chooses the fast memory size, Z, to be the effective on-chip memory capacity. On CPUs, Z is the last level cache; on GPUs, Z is the shared (local) memory and L2 cache. While the proposed architecture model represents on-chip memory as a unified fast memory, actual on-chip memories have complex hierarchies with multiple levels and some levels are physically distributed (e.g., GPU's local memory). Therefore, AutoMatch

Table 3.1: The specifications of target hardware architectures.

Model Parameter	**Intel i5-2400**	**Intel i7-4700**	**Tesla C2075**	**Tesla K20C**	**Tesla K20X**
ID	$ARC1$	$ARC2$	$ARC3$	$ARC4$	$ARC5$
Clock (GHz)	3.1	2.4	1.15	0.732	0.732
n_p	4	4	448	2496	2688
π_0 (GFLOPS)	20	33	0.9	0.41	0.42
Z (MB)	6	6	1.6	2.3	2.3
L (Byte)	64	64	128	128	128
β_f (GB/s)	285	349	2117	2018	2424
α_f (us)	0.004	0.004	0.028	0.045	0.045
β_s (GB/s)	18.88	11.5	87.92	129.73	160.1
α_s (us)	0.065	0.052	0.71	0.68	0.68
s_0 (us)	0.2	0.44	7.22	6.5	6.5

estimates the fast memory bandwidth and latency, β_f and α_f, as the average memory bandwidth and latency of the on-chip memory hierarchy. It turns out that the fast memory of the target architectures is better than the slow memory by approximately a factor of 15 in terms of memory bandwidth and latency. The only exception is $ARC2$, where the memory bandwidth ratio between the fast and slow memories is ≈ 30.

Finally, AutoMatch estimates the global synchronization cost, s_0, using barrier synchronization between threads on CPUs and workgroups (thread-blocks) on GPUs. While there are several global synchronization methods on GPUs, AutoMatch uses the host-implicit global synchronization which is the simplest and most popular one [206]. Because the number of active threads can significantly affect the synchronization overhead, AutoMatch estimates the global synchronization cost at full occupancy, i.e., it launches one thread per logical core on CPUs and four workgroups (thread-blocks) of dimension 32×32 per compute unit (streaming multiprocessor) on GPUs.

3.2.3 Compiler-based Application Analysis

Design and Implementation

AutoMatch uses the LLVM compiler framework [113] and works on the intermediate representation (IR) of the sequential code, which makes it language-independent and applicable to any source code and programming language supported by the LLVM front-ends (e.g., C/C++, FORTRAN, and so on).

Figure 3.3 shows the design and implementation of the AutoMatch compiler. Clang and other front-ends parse the sequential code of the target application and emit its IR without any optimization. In case of multiple IR files, LLVM-LINK merges them into one file. Next, OPT performs a set of canonicalization passes on the unoptimized LLVM IR. While the most important pass is the memory-to-register translation, which promotes all temporal stack memory allocation and accesses to registers and converts IR into the single static assignment (SSA) form [132], other passes such as function inlining and constant propagation simplifies the induction variables and control flow and makes the analysis easier. In addition, the user provides the input data and target kernel name. After that, the AutoMatch compiler, which is implemented in the execution engine of the dynamic compiler LLI, statically and dynamically analyzes the optimized IR to extract the architecture-agnostic characteristics of sequential code, which are combined with the specifications of the target heterogeneous system to generate the final performance analysis and predictions.

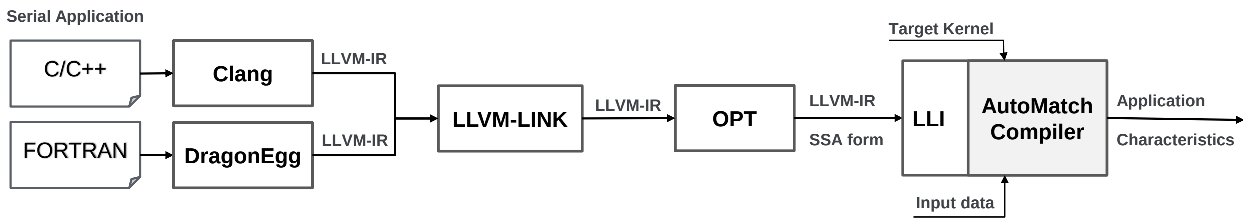


Figure 3.3: The design and implementation of AutoMatch compiler.

Parallelism Analysis

AutoMatch uses a hybrid (static and dynamic) analysis to automatically quantify the inherent parallelism in sequential applications, thereby estimating their computation time on the different architectures for a given input data. In particular, it schedules the application on a theoretical architecture with an infinite number of registers and compute units and a zero memory access latency, such that each operation is executed as soon as its true dependencies are satisfied. Figure 3.4 depicts our graph-structured program representation, where nodes are dynamic instances of floating-point instructions (operations), denoted as I_{nm}, and edges are true dependencies between operations, after scheduling the compute operations on the theoretical architecture using as soon as possible (ASAP) schedule constraints. For example, if each dynamic instance m of a floating-point instruction I_n is scheduled at an execution level j, then I_{nm} must have dependencies at the execution level $j - 1$.

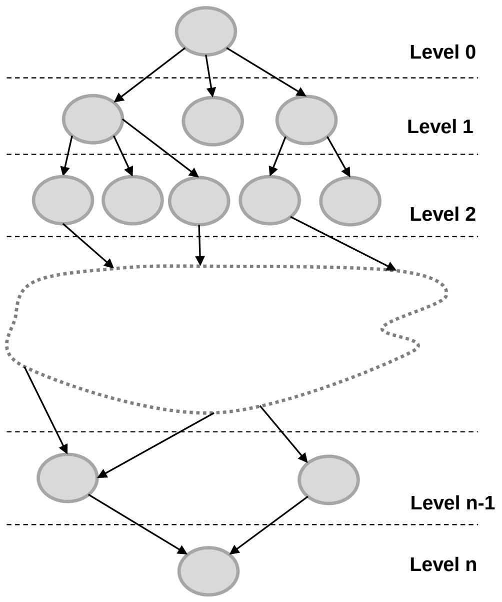


Figure 3.4: The ASAP schedule of the application operations on a theoretical architecture with infinite resources.

This ASAP schedule is similar in spirit to the classical work-depth model [26, 49], which represents the computations of a given algorithm using a directed acyclic graph (DAG) in which nodes and edges represent the compute operations and their data dependencies, respectively. While the classical work-depth model requires manual analysis to quantify the sequential part and average parallelism of a given algorithm, AutoMatch not only generates the ASAP schedule automatically to estimate the computation time, but also considers the workload imbalance, vectorization potential, instructions mix, and resource constraints of the target architectures.

To identify the true dependencies between compute operations, AutoMatch uses several static and dynamic analysis techniques. First, it constructs the def-use chains [132] at compile time to track data dependencies through registers; due to the infinite number of registers, only true dependencies exist. Second, it uses LLVM's dynamic compiler, LLI, to profile the application and collect the execution history of the compute instructions and memory operations. Third, using the application execution history, AutoMatch implements a dynamic points-to analysis [131, 106] to track data dependencies through the memory operations. Leveraging a hash table that resembles a content-addressable memory (CAM), AutoMatch dynamically detects true (read-after-write), anti (write-after-read), and output (write-after-write) memory dependencies.

Based on the detected true dependencies, AutoMatch constructs the ASAP schedule of the sequential application on the theoretical architecture and computes D, the number of execution levels (i.e., the depth of the critical path), and w_i, the total number of operations in each execution level i. In addition, AutoMatch computes f_{im}, the instruction mix of the sequential application to estimate the performance degradation factor relative to the peak floating-point throughput (π_0) on parallel architectures with FMA (Fused Multiply-Add) units. The instruction mix factor f_{im} is defined as:

$$f_{im} = \frac{W_{add} + W_{mul}}{2 \times \max(W_{add}, W_{mul})} \tag{3.1}$$

where W_{add} is the number of addition and subtraction operations, and W_{mul} is the number of multiplication operations.

Moreover, AutoMatch leverages the LLVM vectorizer to identify the loops that are amenable to vectorization, and computes W_{vec}, the number of floating-point operations that can efficiently utilize the vector (SIMD) units. Next, it estimates f_v, the performance degradation factor relative to the peak floating-point throughput on parallel architectures with vector units, as follows:

$$f_v = \frac{W_{vec}}{W} \tag{3.2}$$

where W is the total number of floating-point operations.

Data Locality Analysis

AutoMatch quantifies the inherent data locality in sequential applications by analyzing their memory access patterns on the above abstract architecture model, which assumes an ideal cache-memory model. The main goal is to estimate the number of data transfers between the compute cores and the fast memory, Q_f, and between the fast and slow memories, Q_s. Since the proposed architecture model assumes that the fast memory is fully associative and uses the LRU replacement policy, AutoMatch adopts the LRU stack distance analysis [127]. The LRU stack distance (or reuse distance) is the number of distinct memory locations accessed between two successive accesses to the same memory location; the LRU stack distance of the first reference to a memory location is ∞. Figure 3.5 shows an example of the LRU stack distance analysis on a memory access trace of 10 memory references.

Memory location accessed	a	c	d	b	c	e	g	e	d	d
LRU stack distance	∞	∞	∞	∞	2	∞	∞	1	4	0

Figure 3.5: LRU stack distance analysis example

In a fully-associative cache with the LRU replacement policy, a memory reference with an LRU stack distance larger than the fast memory size results in a miss or an access to the slow memory. Hence, Q_s and Q_f can be estimated from the number of memory references with an LRU stack distance larger than the fast memory size and the number of memory references with an LRU stack distance less than or equal to the fast memory size, respectively.

While the LRU stack distance analysis ignores the conflict and contention misses, AutoMatch assumes that the memory transfers on parallel architectures are bounded by Q_s and Q_f, as in prior work [27].

AutoMatch estimates the memory access cost of the target application as follows. First, it dynamically analyzes the LLVM IR instruction stream (execution history) to capture the load and store memory operations, and then records the referenced memory locations (addresses) along with their last access time in a self-adjusting binary search tree [184], which is sorted by the last access time. Second, whenever a memory location is referenced, AutoMatch examines the memory tree to find the last access time; if the target memory location does not exist in the memory tree, the current memory access has an LRU stack distance of ∞; otherwise, AutoMatch finds the distinct nodes accessed between the last access to the target memory location and the current access; the number of such nodes is the LRU stack distance of the current memory reference. Third, AutoMatch counts the number of memory references with a particular LRU stack distance to generate the LRU stack distance histogram. Finally, it combines this histogram with the specifications of target architectures and the ASAP schedule of the application to compute Q_f and Q_s.

Synchronization Analysis

While the parallelization overheads consist of thread creation/destruction, kernel launch, and synchronization, AutoMatch focuses on the synchronization for two reasons. First, unlike the other overheads, the synchronization is not a one-time cost and can increase with the problem size. Second, the synchronization overhead is significant on massively parallel GPU architectures; as shown in Table 3.1, their synchronization overhead is an order of magnitude higher than multi-core CPUs.

AutoMatch uses a heuristic for estimating the required global synchronization points to reach a globally consistent memory state on parallel architectures. The proposed heuristic is based on detecting loop-carried memory dependencies. AutoMatch dynamically analyzes the loop nests of the sequential application to find the inherently sequential loops (i.e., loops that cannot run in parallel due to loop-carried memory dependencies) and the parallel loops. It estimates the number of global synchronization points as the trip counts of the inherently sequential loops with inner parallel loops. Figure 3.6 shows an example of two nested loops, where the i-loop is inherently sequential and the j-loop is parallel; in this case, the number of global synchronization points is $n - 2$. In addition, AutoMatch allows users to annotate the source code to indicate the global synchronization points.

```
for(i=1; i < n; i++) {
  for(j=1; j < n; j++) {
    a[i][j] = a[i-1][j] + 2;
  }
}
```

Figure 3.6: Detection of global synchronization.

3.2.4 Analytical Modeling

Execution Cost Estimation

AutoMatch constructs the execution cost (EC) model that captures the complex interaction of the application, input data, and target architectures. In addition, it can be generalized to different types of hardware architectures. After analyzing the architecture-agnostic features of the sequential application, AutoMatch combines these features with the specifications of target architectures to generate *first-order* analytical models to estimate the computation time, memory access time, and synchronization overhead.

The computation time, T_{comp}, is estimated as:

$$T_{comp} = \frac{D}{\pi_0} + \underbrace{\sum \frac{w_i}{\min(w_i, n_p) \times (\pi_0 \times f_v \times f_{im})}}_{\forall i} \tag{3.3}$$

where D is the number of dependency levels, w_i is the total operations for each dependency level i, n_p is the number of cores, π_0 is the maximum operation throughput per core, f_v is the vectorization factor, and f_{im} is the instruction mix factor. This equation extends the classical Amdahl's law by considering the effect of instruction mix and vectorization potential on the computation throughput. The first term models the sequential execution, which depends on the inherent dependency chain, while the second term models the parallel execution that is limited by either the available cores or work in a given execution (dependency) level.

The memory access time, T_{mem}, is computed as follows:

$$T_{mem} = (\alpha_f + \alpha_s) \times D + (\frac{Q_f}{\beta_f} + \frac{Q_s}{\beta_s}) \times L \tag{3.4}$$

where α_f and α_s are the access latency of the fast and slow memories, β_f and β_s are the memory bandwidth of the fast and slow memories, Q_f is the number of memory transfers between the compute cores and the fast shared memory, Q_s is number of memory transfers between the fast memory and the slow memory, D is the depth of the application ASAP schedule, and L is the memory transfer size. This equation accounts for the memory latency once per execution (dependency) level, and assumes that memory transfers are effectively pipelined by the memory system such that they are limited by memory bandwidth.

The synchronization overhead, T_{syn}, is estimated as:

$$T_{syn} = S \times s_0 \tag{3.5}$$

where S is the total number of global synchronization points, and s_0 is the global synchronization cost.

Finally, AutoMatch evaluates equations (3.1)-(3.5) to predict the execution cost on each architecture, which is estimated as the overall computation time, memory access time, and global synchronization overhead. Moreover, AutoMatch predicts the parallel resource contention by considering the access time to shared resources, such as the fast and slow memories, assuming that they are shared fairly among threads. Next, AutoMatch combines the execution cost on the different architecture with the floating-point work of the application to predict the maximum *parallel* performance on each architecture.

Workload Distribution

AutoMatch estimates the relative execution cost across the different architectures to drive a workload distribution service for parallel compute kernels on heterogeneous CPU-GPU nodes. The main objective of this runtime service is to distribute the workload (i.e., iteration space and data) over the available heterogeneous architectures to minimize the overall execution time. Instead of distributing the workload evenly across CPUs and GPUs, AutoMatch reduces the overall execution time by considering the relative computing power of each architecture with the execution cost prediction above. For example, if AutoMatch's relative execution cost of a compute kernel on the CPU and the GPU is three to one, its workload distribution scheme partitions the workload into four parts and assigns three parts to the GPU and one part to the CPU.

3.3 Evaluation

This section demonstrates the efficacy of AutoMatch and its utility as a *first-order* performance prediction framework for sequential applications on heterogeneous HPC systems. The experiments use AutoMatch to analyze the *sequential* implementation of target applications and show its estimation in comparison to the heterogeneous parallel implementations.

The target applications are built by the following compilers: gcc 4.9, icc 13.1, and nvcc 7.5, and AutoMatch is implemented in LLVM-3.6. While AutoMatch works with any data type supported by LLVM, this work considers double-precision floating point only for brevity. Since the key evaluation point is in the relative performance of the different HPC architectures at the chip level, the reported performance is for the core computational kernels and ignores one-time cost overheads, such as I/O, data initialization (including host-device transfer), profiling, and debugging.

3.3.1 Performance Forecasting and Analysis Case study

This case study shows the effectiveness of AutoMatch in identifying the performance upper-bound of sequential applications on heterogeneous HPC architectures and how close the parallel implementation is to the best parallel performance. In addition, the study presents the sensitivity of AutoMatch to variations in the architectural characteristics and its ability to predict the relative ranking of the architecture alternatives.

We consider eight (8) HPC workloads from Rodinia [36] and Parboil [181] benchmarks with different parallelism profiles and memory access patterns. The reason for choosing Rodinia and Parboil is because they provide sequential/multi-threaded CPU implementations as well as GPU implementations, which are used as reference for several black-box performance prediction approaches [13, 17]. Table 3.2 presents the target workloads and the input data sets provided by their benchmark suites.

Table 3.2: Rodinia and Parboil workloads.

Workload	**Description**	**Input data**
CUTCP	Simulation of explicit-water biomolecular model which computes the Cutoff Coulombic Potential over a 3D grid	watbox.sl40.pqr
STENCIL	Iterative Jacobi solver on a 3D structured grid	Grid 512x512x64
SPMV	Sparse matrix-vector multiplication	Dubcova3.mtx [52]
LBM	Lid-driven cavity simulation using the Lattice-Boltzmann method	120_120_150_ldc.of
LUD	LU decomposition on a dense matrix	Matrix 512x512
LavaMD	Molecular-dynamics simulation which calculates the potential due to mutual forces between particles in a 3D space	boxes1d 10
HotSpot	Thermal simulation and modeling for VLSI designs	temp_1024 power_1024
SRAD	Image processing used to remove locally correlated noise, known as speckles	image 512x512

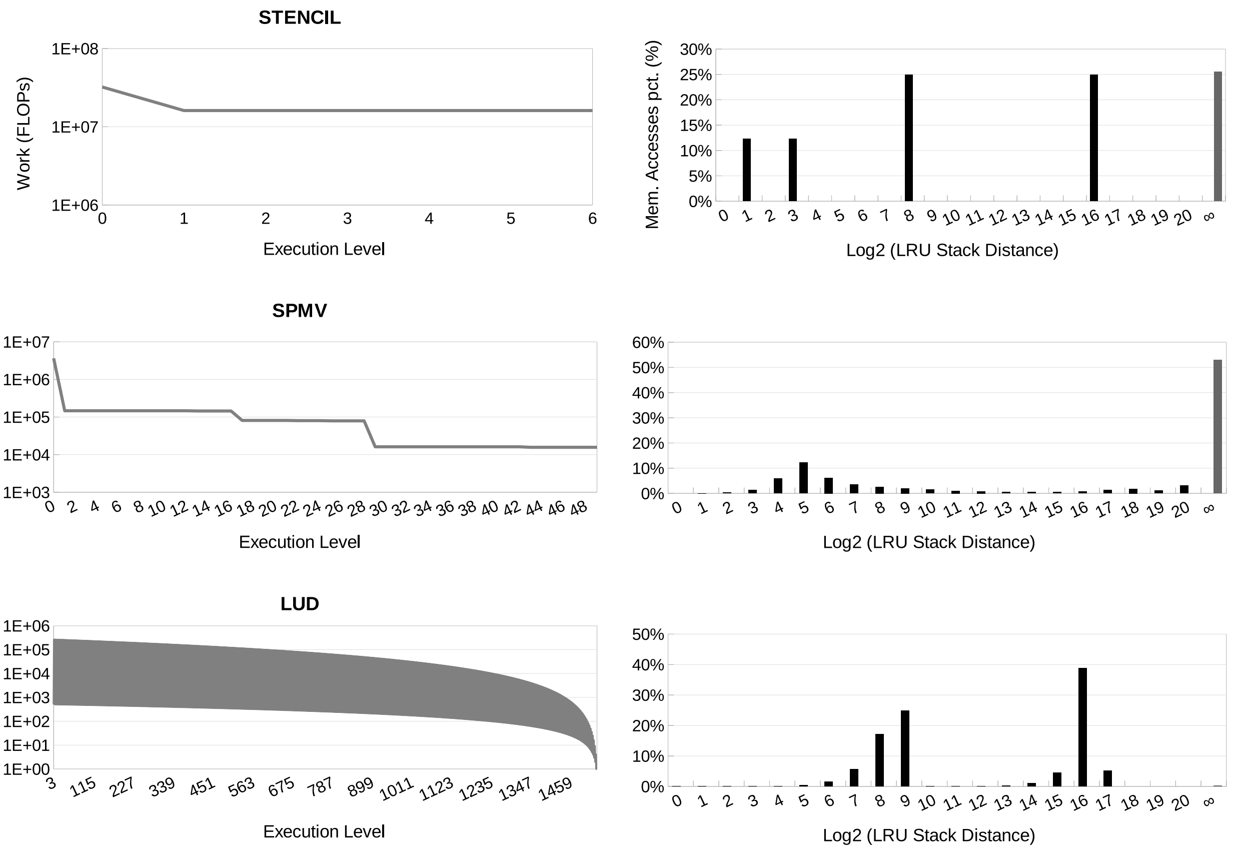


Figure 3.7: Parallelism and LRU stack distance profiles

Figure 3.7 presents the parallelism and LRU stack distance profiles of the target workloads; for brevity, it shows three workload results: STENCIL, SPMV, and LUD. AutoMatch indicates that STENCIL is inherently parallel with a few execution levels and massive amounts of work per level, and it has a uniform memory access pattern with few memory streams corresponding to the dimensions of the data grid. SPMV has a small number of execution levels; however, the work (number of compute operations) per level is significantly lower than STENCIL, due to the sparsity of the input matrices. In addition, SPMV suffers from low data locality, as the compulsory misses (memory references with LRU stack distance ∞) dominate the memory accesses. LUD has an irregular parallelism profile that alternates between two bounds corresponding to the computation of the pivot column and the update of the trailing sub-matrix, respectively. For LUD, the amount of work per execution level decreases as it moves down the critical path of the application schedule, which results in workload imbalance. Moreover, LUD has scattered memory access streams, because the data accessed decreases as the execution progresses due to workload imbalance.

Figure 3.8 shows AutoMatch's estimation of the upper bounds of parallel performance compared to the achieved performance of the OpenMP and CUDA implementations, while Figure 3.9 provides AutoMatch's analysis of the execution bottlenecks on the different architectures. The experiment considers the first subset of the target architectures ($ARC1$, $ARC3$ and $ARC5$) which contains heterogeneous architectures with significantly different hardware capabilities. The results show that AutoMatch accurately identifies the best architecture and the relative ranking of the different architectures in all test cases. Moreover, the actual parallel implementations do not exceed AutoMatch's prediction, which indicates that AutoMatch accurately predicts the performance upper-bound.

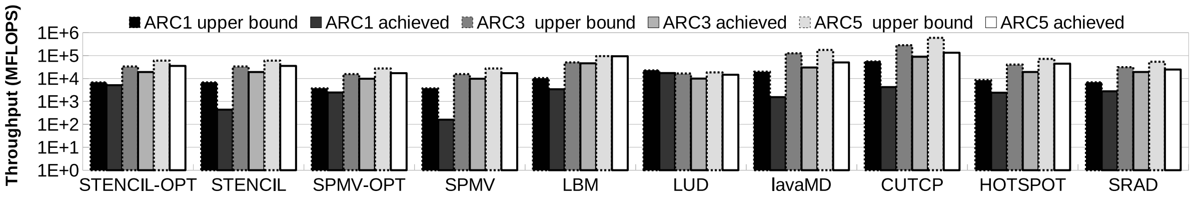


Figure 3.8: Achieved performance vs. AutoMatch's upper bounds.

Optimization Studies

AutoMatch helps the user to determine how close the parallel implementations is to the performance upper bounds. The results show that the gap between the achieved performance and the estimated upper bounds on many-core GPUs ($ARC3$ and $ARC5$) is small in most cases; however, this gap is quite large on the multi-core CPU ($ARC1$). In particular, the actual parallel implementations show that GPU architectures achieve more than two orders-of-magnitude speedup (up to 120×) over the CPU architecture, while AutoMatch reports lower relative speedup between the two architectures (up to an order-of-magnitude speedup).

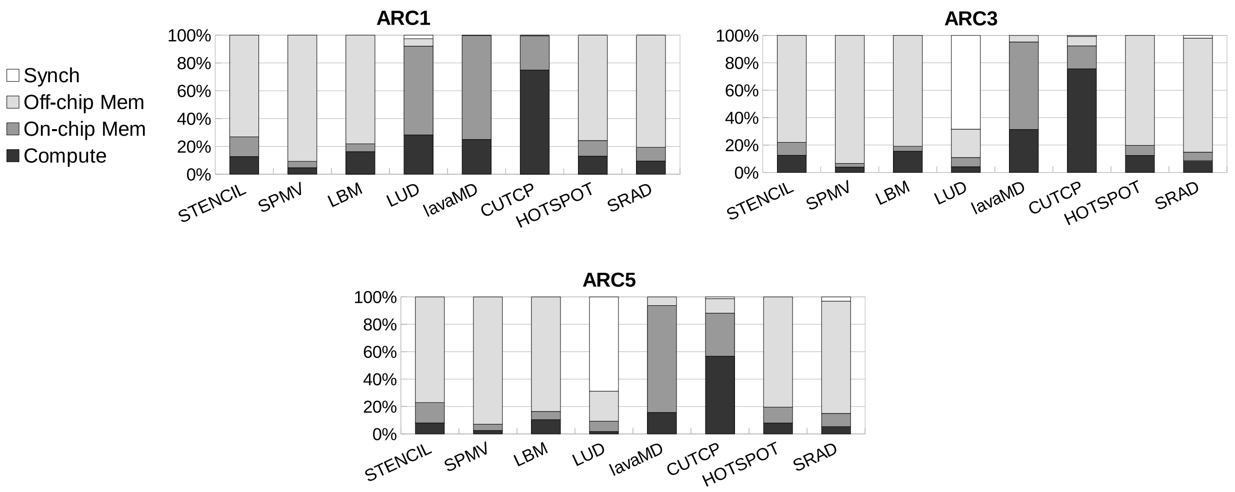


Figure 3.9: AutoMatch's prediction of execution bottlenecks.

The important question here is *whether this performance gap is due to AutoMatch's prediction error or because the benchmark suites are not equally-optimized for each architecture type.*

As Lee et al. [115] debunk the unrealistic 100X speedup of GPUs vs. CPUs and show that it results from an unfair comparison with inferior CPU implementations, our hypothesis is that *the benchmark suites are not equally-optimized for each architecture type.* After inspecting their actual parallel implementations, it turns out that the CUDA implementation is optimized whereas the OpenMP implementation is unoptimized, which is justified by the following. First, the OpenMP implementation does not utilize the vector units, which reduces the performance of compute-bound workloads (e.g., LavaMD and CUTCP). Second, the OpenMP code is cache-unfriendly, e.g., it distributes loop iterations with unit-stride memory accesses on different threads (STENCIL) and uses array of structures (LavaMD, CUTCP, and LBM). Simple data-layout optimization can dramatically improve the performance of CPU caches [42]. Third, the CPU code incorrectly uses GPU-specific optimizations, e.g., irregular applications (SPMV) use the GPU-friendly compressed format (JDS format).

To verify our hypothesis and to show that the estimated upper bounds are attainable, the experiment here considers two cases: a regular workload (STENCIL) and an irregular workload (SPMV). First, AutoMatch indicates that STENCIL is bounded by the off-chip memory access time (see Figure 3.9), and it has a few memory access streams corresponding to the dimensions of the input data grid (as detailed in Figure 3.7). We found that the original workload distribution strategy (of the baseline OpenMP implementation) partitions the input data grid along the X-axis, which has the smallest reuse distance or highest locality, and distributes chunks of Y-Z planes over the different threads. Hence, we changed the workload distribution strategy to distributes chunks of X-Y planes over the different threads.

Second, AutoMatch shows that SPMV suffers from low data locality and it has limited parallelism (see Figure 3.7), which increases the load imbalance especially for architectures with a massive number of threads. While the original OpenMP implementation uses the jagged diagonal storage (JDS) format, which is more suitable for data-parallel architectures with fine-grained parallelism [165], we use the compressed sparse row (CSR) format that outperforms JDS on coarse-grained parallel architectures with large caches. In addition, we used a dynamic workload distribution strategy that distributes chunks of 32 compressed rows over the available cores.

As shown in Figure 3.8, the performance of our implementations, named STENCIL-OPT and SPMV-OPT, is significantly better than the performance of the original implementations on $ARC1$, which means that the estimated performance upper bounds can be achieved with platform-specific optimizations and tuning. Moreover, while AutoMatch's prediction gap of the relative speedup is *91%* on average for STENCIL and SPMV, it dramatically drops to *15.5%* on average for STENCIL-OPT and SPMV-OPT.

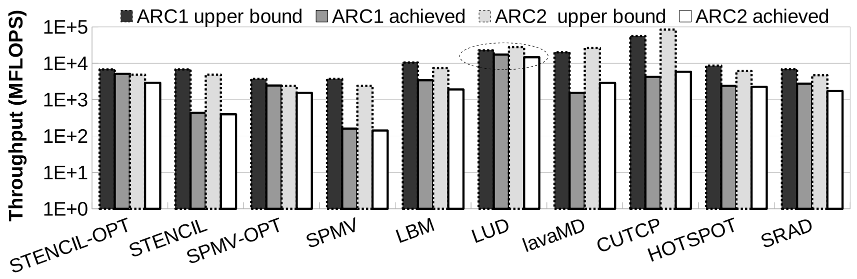
The above optimization studies pinpoint the critical issue with the "black-box" prediction approaches (e.g., profiling-driven and machine-learning). Since they rely on the performance of a training set of parallel applications, their results can be easily fooled by heterogeneous implementations that are *not equally-optimized* for each target architecture. In other words, their predicted relative speedup (and prediction accuracy) can be misleading without the availability of a diverse set of applications along with their optimized implementations for each architecture type; in general, finding such applications is another daunting task.

Finally, the gap between the the performance upper bounds and the achieved performance on many-core GPUs is relatively large for LavaMD and CUTCP, which are bounded by the compute time and on-chip memory access time according to AutoMatch's analysis. The investigation of the CUDA implementations of LavaMD and CUTCP shows that they suffer from low occupancy (37% and 27%), due to the high registers and local memory usage which limits the number of concurrently active wavefronts (warps) and workgroups (thread-blocks). Kernel fission [203] can be used to improve the occupancy by partitioning the kernel into smaller kernels with less resources usage.

Sensitivity Analysis

The experiment here considers the second and third subsets of target architectures, which contain multi-core CPUs ($ARC1$ and $ARC2$) and many-core GPUs ($ARC4$ and $ARC5$) architectures with similar hardware characteristics and capabilities (see Table 3.1).

Figure 3.10 shows AutoMatch's performance prediction and the actual performance on these architecture subsets. Surprisingly, AutoMatch accurately predicts the best architecture in all the test cases, except for the LUD benchmark on multi-core CPUs, which shows that our automatically-generated, high-level performance models are sensitive to the small variation of the target architectures. For LUD, AutoMatch indicates that it is bounded by the fast memory access time on multi-core CPUs ($ARC1$ and $ARC2$), and its parallelism and LRU stack distance profiles show a non-uniform memory access pattern, where the data being accessed decreases as the execution progresses due to workload imbalance. Hence, our hypothesis is that the higher memory bandwidth of $ARC2$ is underutilized due to the non-uniform memory access pattern of LUD, leading to the incorrect ranking. While AutoMatch' high-level memory model captures the data locality of the target applications, it does not consider the uniformity of the memory access pattern and its effect on several hardware features such as hardware prefetchers, memory coalescing units, and write buffers. In addition, the micro-benchmarking approach has the same limitation, as it uses a stream-like memory access pattern to measure the memory bandwidth of target architectures.



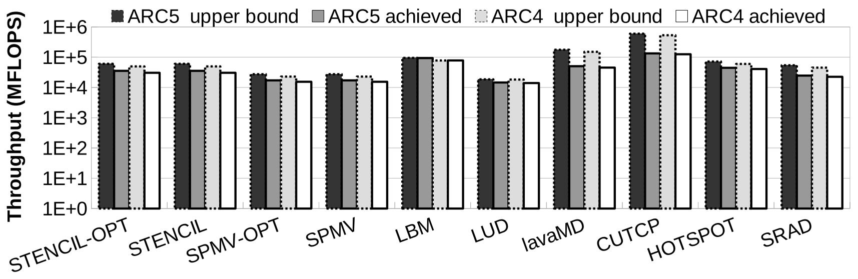


Figure 3.10: The prediction sensitivity of AutoMatch.

3.3.2 Workload Distribution Case Study

This study shows the effectiveness of our workload distribution scheme based on the execution cost model generated by AutoMatch from analyzing the sequential codes. Our scheme is compared to an oracle workload distribution scheme obtained by runtime profiling of the optimized parallel implementations.

To evaluate AutoMatch's workload distribution, we use three applications from the structured grids and sparse linear algebra design patterns, which are widely used in computational fluid dynamics (CFD). MiniGhost [19] is a representative application for multi-material, hydrodynamics simulation. The main computational kernel is the finite difference solver, which applies a difference stencil and explicit time-stepping scheme on a homogeneous 3D grid. Heat2D solves the Poisson partial differential equations (PDEs) for heat diffusion in homogeneous two-dimensional grid [148]. SPMV is a canonical sparse-matrix dense-vector multiplication. The experiment uses the implementations provided by the MetaMorph library [79], which supports the seamless execution of CFD applications on multiple heterogeneous devices, including CPUs (OpenMP back-end) and GPUs (CUDA back-end). In addition, MiniGhost and Heat2D are configured to apply 3D 7-point and 2D 5-point stencils, respectively, on a single global grid, and to use an explicit time-stepping with 100 time steps.

The target platform is a heterogeneous CPU-GPU node that includes $ARC1$ and $ARC5$ devices. Three different workload distributions are tested: default, AutoMatch, and Oracle distribution. The default strategy is to distribute the workload evenly across the available devices. The AutoMatch workload distribution uses AutoMatch to analyze the sequential implementation and to predict the execution cost on the heterogeneous devices. Next, based on the predicted execution cost, it distributes the workload to minimize the overall execution time. The Oracle distribution is similar to AutoMatch strategy; however, instead of predicting the execution cost, it profiles the parallel code on the target CPU and GPU and distributes the workload based on the *measured* execution time.

Figure 3.11 shows the overall execution time of the target applications with the different workload distribution strategies. The results demonstrate that the AutoMatch and Oracle strategies achieve comparable performance, outperforming the default strategy by a factor of 3.5× and 3.8× on average, respectively. In summary, AutoMatch's workload distribution achieves approximately 90% of the oracle performance, due to its accurate estimation of the relative execution cost across CPU and GPU architectures.

3.3.3 Discussion

While AutoMatch generates simple and intuitive models, the results show that it works well as a *first-order* framework; however, it has several limitations. First, AutoMatch ignores one-time overheads such as host-device data transfers, which depend on the runtime system and the expansion bus rather than the hardware devices, and assumes that the performance is dominated by the compute kernels. While this is a valid assumption for long-running HPC applications, extending AutoMatch to model the host-device interconnect and data transfers enables the users to explore their effect on the overall performance. Second, AutoMatch

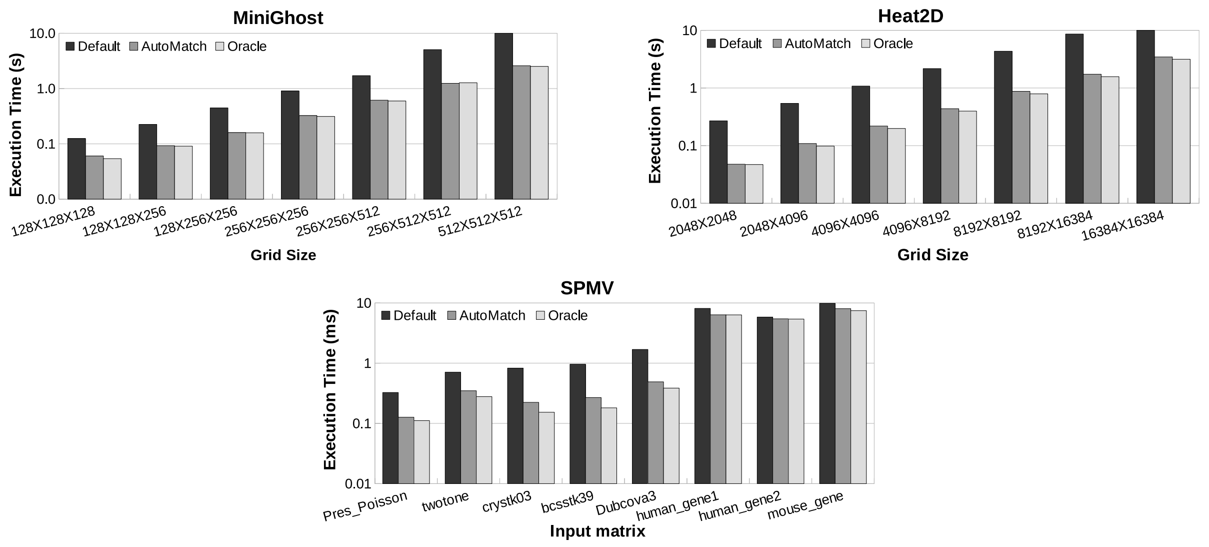


Figure 3.11: The performance of compute kernels with the different workload distribution strategies: Default, AutoMatch, and Oracle.

ignores low-level, architecture-specific features, such as hardware prefetchers, memory coalescing units, thread divergence, and resource occupancy. Although AutoMatch can be extended, beyond its main goal as a *first-order* performance prediction tool, to incorporate more sophisticated models (such as the MWP-CWP model[93]), there is a trade-off between the tighter performance bounds and both the generalization to different architecture types and the limited insight about the critical performance parameters.

3.4 Conclusion

This chapter presented AutoMatch, an automated framework that constructs a fine-grained, graph-structured program representation using various compiler-based analysis techniques to reason about the performance bottlenecks of sequential codes on heterogeneous HPC architectures. AutoMatch combines this program representation with an abstract hardware model, micro-benchmarking approach, and analytical performance modeling to project (1) the realizable performance upper bounds of sequential applications on heterogeneous parallel architectures, (2) the relative ranking and performance of the architecture alternatives, and (3) the best workload distribution strategy on HPC compute nodes with different parallel hardware devices.

We implemented AutoMatch in the LLVM compiler framework, and used various static and dynamic analysis techniques to quantify the application performance on different types of parallel architectures, including multi-core CPUs and many-core GPUs. The experimental results demonstrated the efficacy of the proposed framework across five different heterogeneous architectures and a set of HPC workloads, with different parallelism and memory access patterns. In particular, AutoMatch's workload distribution turns out to be very effective, achieving comparable performance to the profiling-driven oracle.

Chapter 4

Automated Estimation of Communication Cost and Scalability

4.1 Introduction

To deliver scalable performance to large-scale scientific and data analytic applications, HPC cluster architectures adopt the distributed-memory model. These architectures are more difficult to program than shared-memory models and require explicit decomposition and distribution of the program data and computations, due to the lack of a single global address space. The MPI programming model is the de facto standard for programming applications on HPC clusters [56, 20]. MPI uses explicit messaging to exchange data across processes that reside in separate address spaces, and it is often combined with shared-memory programming models, such as OpenMP [146], to exploit the available compute resources seamlessly both within a node and across nodes. Alternatively, the partitioned global address space (PGAS) programming models (e.g., Chapel [55] and UPC [32]) abstract explicit communication using distributed-memory objects; however, the user still needs to manage the data distribution and locality across compute nodes.

Current (and future) HPC cluster architectures suffer from an increasing gap between the computation and communication costs, i.e., the cost of data transfers can be orders of magnitude higher than the cost of compute operations [24]. Therefore, the performance and scalability of user applications on HPC clusters are limited by the communication cost across compute nodes. In particular, the asymptotic scalability/efficiency of a program on distributed-memory architectures is determined by the growth of the communication with respect to the problem size and the number of processes/nodes [76, 72].

Hence, fast and accurate prediction of the minimum communication cost and maximum scalability of user applications plays a critical role in assessing the benefits of porting these applications to HPC clusters as well as in guiding the development of efficient distributed-memory implementations. Unfortunately, estimating the communication and scalability of a given application is a complex and time-consuming process that requires extensive manual analysis and a wide array of expertise in the application domain, HPC architecture, and programming model.

Researchers have created several performance and scalability analyzers [18, 29, 33, 209, 73, 205] and communication pattern detectors [163, 96, 16] to study the effect of data transfers on the application performance and to provide valuable insights on the optimization of the communication bottlenecks. However, these tools are limited only to *MPI implementations*. That is, the estimated communication is specific to the *given MPI implementation* and its workload decomposition and distribution strategy rather than the inherent characteristics of the *original application*. Moreover, due to the parallel programming effort and time on distributed-memory systems, the MPI implementation is often not available in the early stages of the development process.

Thus, there is a compelling need for automated tools, which can analyze the sequential user applications and predict the communication cost of their parallel execution on distributed-memory HPC clusters, to estimate the scalability and performance beforehand without needing comprehensive knowledge of the target applications and cluster architectures.

4.1.1 CommAnalyzer Framework

This chapter presents CommAnalyzer, an automated framework for estimating the communication cost of sequential codes, to figure out the scaling characteristics of running them in parallel on a distributed-memory HPC system. Figure 4.1 shows the proposed framework which takes as inputs the sequential application code (written in any of the languages supported by LLVM compiler [113]), representative input data, and the number of compute nodes (e.g., 2-64 nodes).

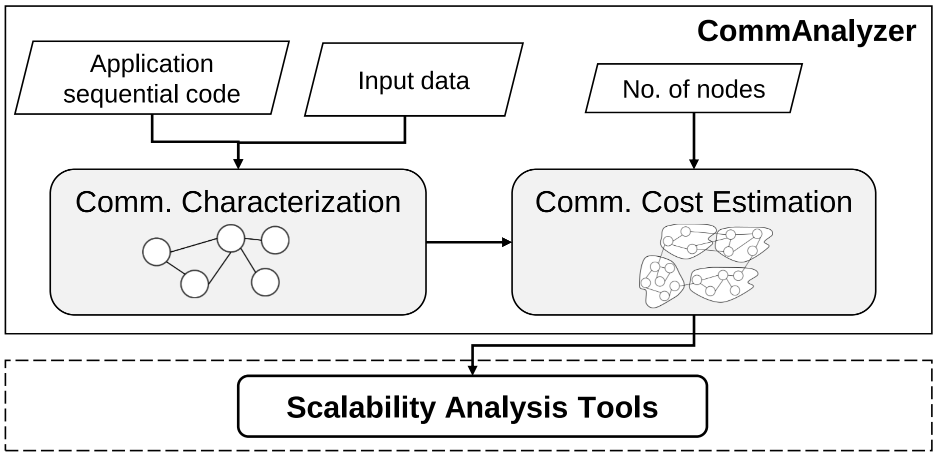


Figure 4.1: Overview of CommAnalyzer framework.

The key idea is to reformulate the problem of estimating the communication of a parallel program into that of analyzing the inherent flow of program values (information) in the sequential application. Hence, CommAnalyzer uses novel dynamic program analyses that build a *value communication graph* (VCG) from sequential codes taking into account the data-flow behavior of their program values. Since parallel programs are typically optimized to minimize the communication between compute nodes, their data transfers are likely to serve as a cut for partitioning the VCG. CommAnalyzer in turn leverages graph partitioning algorithms over the VCG to automatically identify sender/receiver entities and to estimate their communication. As a result, for a given sequential application, CommAnalyzer allows the user to project the performance upper-bound of its effective distributed-memory parallel execution, regardless of the target programming model (e.g., MPI and PGAS).

The communication cost estimation of CommAnalyzer can be used by scalability analysis tools, such as Extra-P [29], to estimate the strong and weak scaling of the communication. These tools perform regression analysis using the scaling functions (regression hypothesis) that exist in HPC applications to estimate a user-specified metric (e.g., FLOPs, communication, etc.) at a large scale from a set of small-scale measurements or predictions with different problem sizes and/or number of nodes. Furthermore, CommAnalyzer makes it possible to construct bound-and-bottleneck models of the parallel efficiency and scalability on distributed-memory clusters.

Use Cases. CommAnalyzer accelerates the application and system design in the early stages of the development process by allowing end users to figure out the communication cost and scaling behavior of their sequential applications. As such, domain scientists can quickly make informed decisions about the need to explore other solutions/algorithms to the problem at hand to attain better parallel performance. That way, CommAnalyzer enables the system designers to evaluate the HPC system design alternatives to achieve the required performance. Even if a distributed-memory implementation of the target application is available, CommAnalyzer still plays a critical role in the optimization process by generating bound-and-bottleneck scaling models that show how close the current parallel implementation is to the scalability/efficiency Roofline. In addition, by estimating the communication from the sequential code rather than the MPI parallel code, CommAnalyzer empowers existing scalability analysis tools for MPI applications to serve a wide range of end users.

4.1.2 Contributions

Unlike previous approaches that are limited by the imprecision of *compile-time* analyses [87, 201, 77], CommAnalyzer proposes a novel *dynamic* analysis approach that instruments the sequential code to precisely capture the runtime information required to detect not only static value-flow (communication) dependencies, but also dynamic value-flow dependencies through multiple levels of access indirection. Thus, CommAnalyzer is applicable for both regular and irregular problems and works also for programs that cannot be auto-parallelized.

The following are the contributions of this work:

- *A novel and automated approach for estimating the communication cost of sequential applications when ported to HPC clusters based on value-flow analysis, value liveness analysis, dynamic program slicing, and graph algorithms.* This approach quantifies the inherent communication of sequential applications regardless of the underlying data structures, and it is applicable for regular, irregular, and unstructured problems. Using the estimated communication, we can successfully project the efficiency upper-bound of the effective distributed-memory parallel implementations on HPC clusters (§4.4 and §4.5).

- *Model validation and case studies using both regular and irregular workloads (matrix multiplication and sparse matrix-vector multiplication) as well as four structured and unstructured representative applications (MiniGhost [19], Heat2D [148], LULESH [98], and K-means [103]).* The experimental results demonstrate the utility of CommAnalyzer in identifying the minimum communication cost on HPC clusters with more than 95% accuracy on average. As a result, the optimized distributed-memory implementations realize more than 92% of the estimated upper bounds on parallel scalability/efficiency (§4.6).

4.2 Background

4.2.1 Distributed-Memory Execution Model

We assume that the applications running on the target HPC clusters follow the single program, multiple data (SPMD) execution model, which is the dominant approach on such architectures [56, 20]. Figure 4.2 shows the SPMD execution model, where the program data is partitioned and mapped to different processes (compute nodes) and all processes execute the same program to perform computations on their data (i.e., owner-computes rule). The data owned by each process is stored on its private (local) address space, and when the local computations on a process involve non-local data, this data is accessed using inter-process communication.

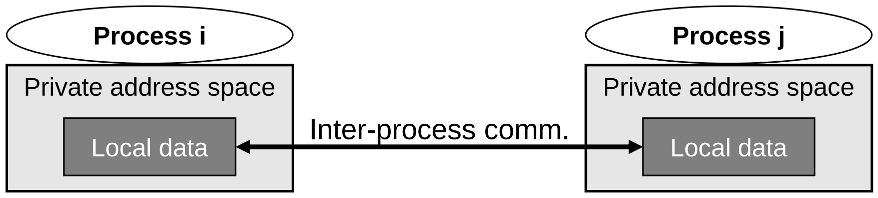
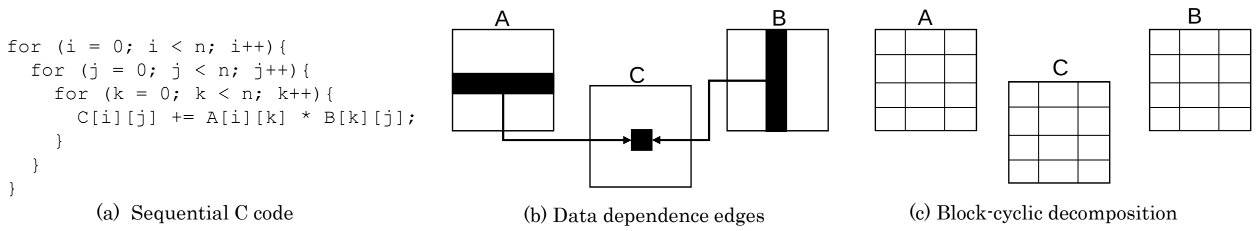


Figure 4.2: The SPMD execution model.

4.2.2 Simple Estimation of Communication Cost

In the SPMD execution model, the inter-process communication results from dependencies between local and non-local data. Therefore, a simple approach for estimating the communication cost is to partition the program data and then identify the data dependencies across different partitions.

Figure 4.3 shows a simple communication estimation for matrix multiplication on distributed-memory architectures. The sequential C code (a) is given to the compiler, and a traditional data-flow analysis is used to identify the data dependence [132] (b) between the data items. Next, a domain decomposition method such as block-cyclic (c) is used to partition and distribute the input matrices over the compute nodes. Finally, the communication cost is estimated as the number of data dependence edges between the data items that exist in different nodes. This approach has been used by the auto-parallelizing compilers for distributed-memory architectures [87, 201] in the early days of HPC. While this simple communication analysis is sufficient for regular and structured problems such as matrix multiplication, real-world scientific applications with irregular computation and unstructured memory access patterns are a lot more challenging and therefore require sophisticated analysis techniques.



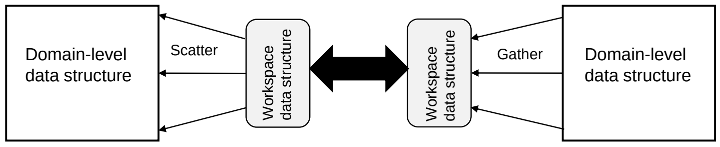
```
for (i = 0; i < n; i++){
  for (j = 0; j < n; j++){
    for (k = 0; k < n; k++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Figure 4.3: Simple estimation of communication cost for matrix multiplication.

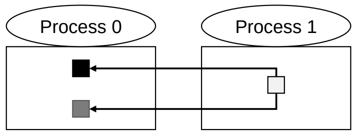
4.3 Challenges

```
for(i = 0; i < numElem; ++i){
  for(j = 0; j < numNodes; ++i){
    x[nodeA[i][j]] += y[nodeB[i][j]]*z[i];
  }
}
```

(a) Indirect memory access



(b) Workspace structures mask the true data dependencies



(c) Redundant communication

Figure 4.4: Challenges for estimating the communication cost.

4.3.1 Indirect Memory Access

Irregular and unstructured problems arise in many important scientific applications such as hydrodynamics [94, 84]. These problems are characterized by an indirect memory access pattern via index data structures, which cannot be determined at compile time. Figure 4.4(a) shows an example of such an indirect access. Here, a traditional data-flow analysis fails to precisely capture the data dependence between the x and y data arrays whose index is a value determined at runtime. Thus, it is impossible to figure out which part of x (y) is defined (used) due to the lack of runtime information.

4.3.2 Workspace Data Structures

In unstructured problems, it is common to gather data items from the program-level (domain-level) data structures into workspace (temporary) data structures. The actual computations are performed in the workspace, and then the results are scattered to the program-level data structures. Usually, there are multiple levels of workspace data structures, i.e., the intermediate results in a workspace are used to compute the values of another workspace.

Figure 4.4(b) shows an example of an unstructured application that uses such workspace data structures to perform its computations. In this common design pattern, workspace data structures mask the true data dependence edges between the data items of the program-level data structures. Hence, the data-flow analysis ends up generating a massive number of local data dependence edges between the program-level data and the workspace data, which leads to inaccurate estimation of the actual communication cost.

4.3.3 Singleton Data Items

Most scientific applications have singleton data items, i.e., data items that are used in almost all the computations (e.g., simulation parameters and coefficients) and data items that use the output of these computations (e.g., simulation error and time step). Hence, there is a massive number of data dependence edges (communication edges) between these singleton data items and the rest of the program data in the original sequential implementation. However, typical MPI implementations create local copies of the singleton data items in each process and use collective communication messages (e.g., broadcast and reduce) to update their values at the beginning and end of the program and/or each time step, instead of accessing their global copy via inter-process communication in every dependent computation. Therefore, the detection of singleton patterns is very important for an accurate estimation of the communication cost.

4.3.4 Redundant Communication

Computing the communication cost as the number of data dependence edges between data items that exist in different processes is subject to overestimation. Figure 4.4(c) shows an example of the communication overestimation, where two items in process 0 use a single data item in process 1 and its value did not change between the two uses. If there is sufficient memory space at process 0 to store the required data value, it can be read only once (instead of two times) from process 1. While the exact data-flow analysis can detect this case in regular applications and remove the redundant communication, it is not possible to do the same for irregular applications due to the indirect memory access.

4.4 CommAnalyzer Approach

It is a daunting challenge to analyze sequential codes and predict the communication cost of running them in parallel on HPC clusters without the distributed-memory parallel codes. CommAnalyzer relies on the following observation; HPC developers always optimize (minimize) the communication of their distributed-memory parallel programs across compute nodes. In particular, the communication cost of the resulting SPMD implementation cannot be smaller than the inherent data-flow (communication) cost of the original sequential program, which would otherwise break the program correctness. As a result, analyzing the data communication in the sequential codes can serve as a basis for estimating the communication cost of their distributed-memory implementations. However, this presents another challenge, i.e., how to figure out the inherent data communication of the sequential program regardless of its underlying data structures.

The main idea to tackle this challenge is to view the sequential program as an entity that consumes input values, computes intermediate values, and produces output values, as well as to analyze their behaviors. In a sense, these values are small pieces of digital information. Similar to genes, which are small pieces of heredity information, the program values are not constrained by the underlying data structures. Rather, such values can interact, replicate, and flow from one data structure to another as well as evolve to new values. Actually, the program data structures are mere value containers, i.e., placeholders of the program values. In this view, a single value can exist in more than one memory location, and it can even get killed in its original memory location (where it was generated) while it remains alive in another memory location.

Algorithm 1 CommAnalyzer algorithm

```
Input: PROGRAM, N
Output: $\mathbb{COMM\_COST}$
1: $\mathbb{VAL\_FC} \leftarrow$ ValFcDetection(PROGRAM)
2: $\mathbb{VAL\_LIVE} \leftarrow$ ValLiveAnalysis(PROGRAM)
3: $\mathbb{VCG} \leftarrow$ CommGraphFormation($\mathbb{VAL\_FC}$, $\mathbb{VAL\_LIVE}$)
4: $\mathbb{VAL\_PMAP} \leftarrow$ ValDecomposition($\mathbb{VCG}$, N)
5: $\mathbb{COMM\_COST} \leftarrow$ CommEstimation($\mathbb{VAL\_FC}$, $\mathbb{VAL\_PMAP}$)
```

To this end, CommAnalyzer uses a dynamic analysis technique, which is based on dynamic program slicing, to analyze the generation of values, the flow of values, the lifetime of values, and the interactions across values, thereby building the *value communication graph* (VCG).

Algorithm 1 shows the top-level CommAnalyzer approach which takes the sequential program (along with representative input data) and the number (or range) of compute nodes, and then computes the communication cost across these nodes when the program is ported to distributed-memory architectures. In the first place, CommAnalyzer characterizes the inherent communication of the sequential program by detecting the dynamic flow of the program values which is encoded as *value-flow chains* defined as follows:

Definition 4.1 (Value-Flow Chain (VFC))**.** *For a given program value, v, its value-flow chain consists of v itself and all the other values on which v has data dependence, where the values are defined as a set of unique data observed in the memory during program execution.*

CommAnalyzer also analyzes the live range [132] (interval) of the program values. Together with VFCs, it is used to generate the *value communication graph* (VCG) of the program. Then, CommAnalyzer decomposes the VCG into multiple partitions to map the program values that are tightly-connected, due to high-flow traffic between them, to the same compute node/process. Once the program values are mapped to the different compute nodes, CommAnalyzer uses the owner-computes rule (see §4.2) to estimate the communication cost by analyzing the *value-flow chains* across the compute nodes.

4.4.1 Communication Characterization

This section first defines the terminologies used in analyzing the input sequential program and then describes how the inherent communication is understood. CommAnalyzer defines *program* values as a set of unique values observed at runtime, and they are classified into three categories: *input, output, and intermediate* values. The *input* values are the program arguments and any memory location read for the first time at runtime, while the *output* values are the returned data or those that are written to the memory and last until the program termination. During program execution, the program values existing in memory locations can be killed with their updates, i.e., losing the original value. That way, program values can end up existing in memory locations only for a limited interval, and CommAnalyzer considers them as *intermediate* values. Note, if a value remains in at least one memory location, it is still live thus not an *intermediate* value. In addition, any program value that exists only in registers is treated as an *intermediate* value.

Algorithm 2 shows the high-level algorithm for calculating the *value-flow chains* (i.e., the inherent communication). CommAnalyzer needs to identify the unique values observed at runtime and then to investigate the flow across the values by figuring out the dependence in between. It is therefore important to know what value is currently stored in a given memory address during program execution. For this purpose, CommAnalyzer uses a shadow memory (*MemVal*) to keep track of program values that exist in the memory at the granularity of the memory word. Thus, each shadow memory location tracks the latest value stored in the corresponding location in the original memory. Since CommAnalyzer works on the static single assignment (SSA) form [132] of the sequential code (e.g., the LLVM IR [113]), there is no need to track (name) the intermediate values that exist in the registers.

For each store (write) instruction during program execution, CommAnalyzer generates a dynamic program slice from the execution history using the reduced dynamic dependence (RDD) graph method [5]. This slice is the set of dynamic IR instructions involved in the computation of the value (v) being stored in the memory. To determine those values on

Algorithm 2 Value-flow chain detection algorithm

```
Input: PROGRAM
Output: $\mathbb{VAL\_FC}$
 1: $Values = \{\}$                                          ▷ program values
 2: $MemVal = \{\}$                                          ▷ shadow memory-to-value map
 3: for each: dynamic store (write) operation $w$ do
 4:     $DS \leftarrow$ DynamicSlicing($w$, PROGRAM)
 5:     $v, s \leftarrow$ ValueFlowAnalysis($DS$, $MemVal$)
 6:     if $v \notin Values$ then                            ▷ writing new value
 7:         $Values = Values \cup v$
 8:         $\mathbb{VAL\_FC} = \mathbb{VAL\_FC} \cup (v\ ,\ s)$
 9:     end if
10:     $MemVal$[address($w$)] $= v$
11: end for
```

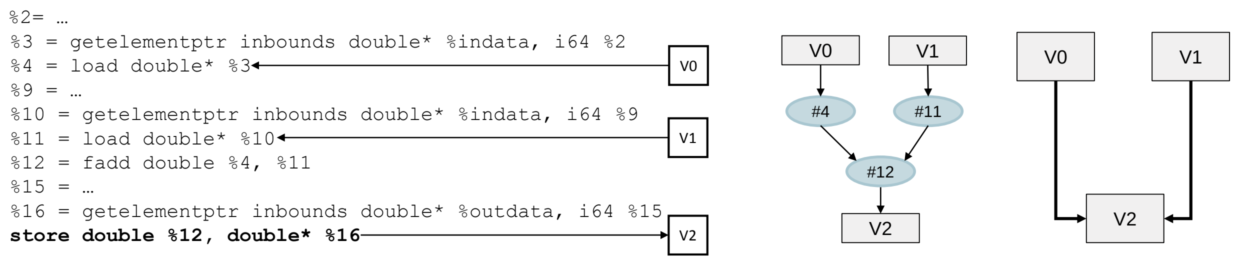
which the value v depends, CommAnalyzer traces it back inspecting the instructions and the registers along the data dependence edge of dynamic slice [1]. Such a dependence backtracking continues for each data dependence edge until it encounters another value that has been recognized using the shadow memory (*MemVal*); whenever a new value is found, CommAnalyzer keeps it in the *Values* set (line 1 of Algorithm 2). Here, the found value turns out to be used in the computation of the value (v) being stored; it is said the former flows to the latter while the former is called a *source* value. At Line 5 of Algorithm 2, CommAnalyzer calculates s which is the set of all the *source* values over the slice of v being stored.

If such a value (v) does not exist in the *Values* set (i.e., new value), CommAnalyzer adds the value-flow chains between v and s to the set of the program value-flow chains ($\mathbb{VAL\_FC}$) and updates the *Values* set; otherwise, v is not unique and already exists in *Values*, and the store instruction just replicates this value and writes it to a new memory location. Usually, the value replication happens when the dynamic program slice does not contain any

[1]The outgoing edges of circle nodes in Figure 4.5 correspond to data dependence edges.

compute instructions (e.g., direct load-store relation). Finally, CommAnalyzer updates the shadow memory *MemVal* with the new value. Further, at the end of the program execution, CommAnalyzer performs the same value-flow chain detection for the return variables.

Figure 4.5 shows a simple example of the value-flow chain detection. For brevity, the example shows the thin dynamic slice [177] instead of the actual dynamic slice, i.e., it excludes the instructions that manipulate the memory pointers. After CommAnalyzer generates the dynamic slice of the target store instruction (a), it uses the shadow memory (b) to track the program values in the memory locations, and then it detects the value-flow in the dynamic slice. Since the exact memory addresses are available in the dynamic slice, CommAnalyzer inspects the shadow memory and records that registers *#4* and *#11* have values *V0* and *V1*, respectively. Next, a new value is computed in register *#12* using the values *V0* and *V1*, and the target store instruction writes this value to the memory. Finally, CommAnalyzer adds the new value-flow chain *(V2, {V0, V1})* in the set of the value-flow chains $\mathbb{VAL\_FC}$.



```
%2= …
%3 = getelementptr inbounds double* %indata, i64 %2
%4 = load double* %3
%9 = …
%10 = getelementptr inbounds double* %indata, i64 %9
%11 = load double* %10
%12 = fadd double %4, %11
%15 = …
%16 = getelementptr inbounds double* %outdata, i64 %15
store double %12, double* %16
```

(a) Dynamic slice (IR code) (b) Value shadow memory (c) Value-flow analysis (d) Resulting value-flow chain

Figure 4.5: An example of value-flow chain detection.

4.4.2 Communication Cost Estimation

Value Communication Graph Formation

After the detection of the value-flow chains (VFCs) between program values, CommAnalyzer creates the *value communication graph* $VCG(V, E)$, where V is a set of vertices that represents the program values and E is a set of edges that represents the value-flow chains.

This is achieved by generating a communication edge between the values in each value-flow chain. Specifically, at Line 5 of Algorithm 2, a connection edge is made between v and every source value in s. $VCG(V, E)$ is a directed and weighted graph, where the weight w of a communication edge represents the number of times the sink (destination) value uses the source value.

Value Graph Compression. CommAnalyzer utilizes the value liveness [132] ($\mathbb{VAL\_LIVE}$), which is generated during the dynamic value analysis, to reduce the communication graph size. An intermediate value is killed when it does not exist in the registers or the program memory. CommAnalyzer coalesces the vertices of the intermediate values that share the same memory location at non-overlapping liveness intervals into a single vertex. Such values are multiple exclusive updates to a single memory location. This vertex coalescing does not impose artificial constraints on the program data partitioning according to the SPMD execution model, where processes compute all the intermediate values for the data items (memory locations) that they own. Finally, after the graph compression, CommAnalyzer updates the weights of the new vertex and edge sets of the value communication graph $VCG(V, E)$ to account for the communication cost before graph compression.

Singleton Detection and Removal. The singleton values appear in most scientific applications and are characterized by extreme connectivity, i.e., a massive number of incoming and/or outgoing communication edges. Example of these values are the simulation coefficients, parameters, and time step. To reduce the communication on distributed-memory architectures, the singleton values should not be mapped to a specific compute node (process). Instead, each compute node creates a local copy of the singleton data items and uses global communication to exchange their values once per the simulation execution and/or the simulation time step. In fact, automated singleton detection and removal is well known in large-scale circuit simulations [188, 78]; specifically, it reduces the overall communication by automatically detecting and duplicating the singleton power/clock circuit nodes in each process. Similarly, CommAnalyzer adopts a threshold-based scoring approach for the singleton detection, which is a simplified variant of the proximity-based outlier detection methods [88].

Algorithm 3 Singleton Detection Algorithm

```
Input: $\mathbb{VCG}$
Output: $\mathbb{S\_VAL}$
1: $\mathbb{S\_VAL} = \{\}$
2: for each: vertex $v \in \mathbb{VCG}$ do
3:     $score \leftarrow$ MAX( DENSITYS($\mathbb{VCG}$, $v$), DISTANCES($\mathbb{VCG}$, $v$) )
4:     if $score \geq HIGH$ then                                    ▷ $HIGH = 90\%$
5:         $\mathbb{S\_VAL} = \mathbb{S\_VAL} \cup v$
6:     end if
7: end for
```

Algorithm 3 shows the high-level singleton detection algorithm where the following definitions are used:

Definition 4.2 (Value Degree Centroid). *The centroid is the minimum of the mean and the median value degree over the VCG, where the value degree of v is defined as the number of value vertices adjacent to v in the VCG.*

Definition 4.3 (Value Degree Distance). *The degree distance of a value d(v) is the distance between its degree and the centroid of the value degree cluster.*

For each value vertex in VCG, CommAnalyzer computes the singleton score, which is the maximum of the density-based score and the distance-based score. When the singleton score is high, the value vertex is identified as a singleton, as shown in Figure 4.6. The density score of a value v is the density of its row and column in the value adjacency matrix, while the distance score is computed by analyzing the value degree distribution of the VCG. If the degree of v is larger than the degree centroid, the distance score of v is computed as $1-(\text{centroid}/d(v))$; otherwise, the distance score is zero. That is, the distance score estimates how positively far v is from the centroid.

Once CommAnalyzer identifies the singleton vertices, it removes them from the communication graph, and maps the remaining values to the compute nodes using graph partitioning.

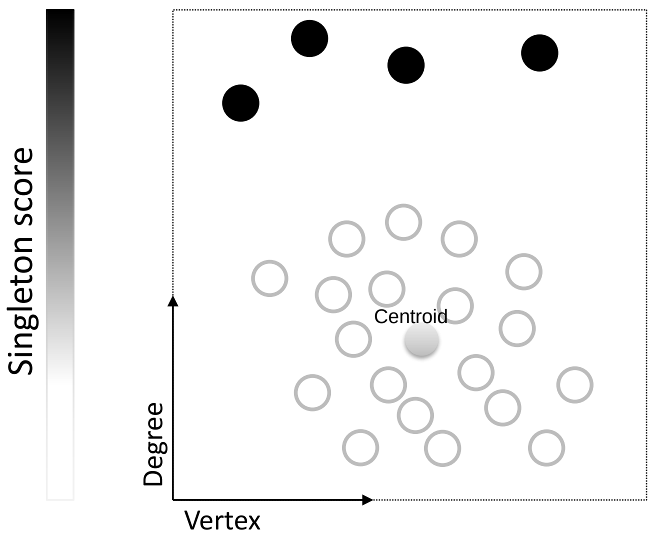


Figure 4.6: Automated detection of singleton values (with extreme connectivity) using the threshold-based scoring algorithm.

Finally, it creates a local copy of the singleton values in each compute node, and accounts for the singleton global communication to project the total communication cost.

Value Decomposition

To predict the minimum communication across compute nodes, CommAnalyzer maps the program values to the compute nodes using a customized graph partitioning algorithm. The value mapping problem has three different optimization objectives: 1) maximizing the load balance which is estimated as the weighted sum of the vertices in each compute node, 2) minimizing the communication across the compute nodes which is the weighted sum of the edge cuts, and 3) generating connected value components in each compute node. CommAnalyzer uses the multilevel partitioning heuristics [100] to solve the value mapping problem in polynomial time, and then generates $\mathbb{VAL\_PMAP}$ which maps each vertex in the value communication graph to a specific compute node. Figure 4.7 shows a simple example of the value decomposition using the graph clustering and partitioning algorithms. Finally, CommAnalyzer maps local copies of the singleton values to each compute node.

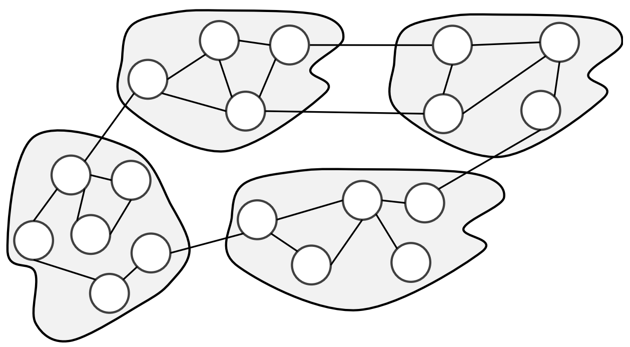


Figure 4.7: Value decomposition using graph clustering and partitioning algorithms.

Communication Cost Estimation

Once the value communication graph (VCG) is decomposed, CommAnalyzer is ready to estimate the overall communication cost and intensity across compute nodes using the value-flow chains and the obtained value decomposition.

Algorithm 4 shows the high-level communication estimation algorithm. Once the program values are mapped to the different compute nodes, all the value-flow pairs with non-local values are identified as communication edges. Then, CommAnalyzer removes the redundant communication by pruning the communication edges that carry the same value between the compute nodes, as the destination (sink) node needs to read this value once and later reuse the stored non-local values (see Figure 4.4(c)).

Algorithm 4 Communication Estimation Algorithm

```
Input: $\mathbb{VAL\_FC}$, $\mathbb{VAL\_PMAP}$, $\mathbb{S\_VAL}$
Output: $\mathbb{COMM\_COST}$
1: $commEdges \leftarrow$ CommDetection($\mathbb{VAL\_FC}$, $\mathbb{VAL\_PMAP}$)
2: $commEdges \leftarrow$ RedundantCommPruning($commEdges$)
3: $\mathbb{COMM\_MAT} \leftarrow$ CommMatGeneration($commEdges$)
4: $\mathbb{COMM\_MAT} \leftarrow$ SingletonComm($\mathbb{COMM\_MAT}$, $\mathbb{S\_VAL}$)
5: $\mathbb{COMM\_COST} \leftarrow$ TotalCost($\mathbb{COMM\_MAT}$)
```

The final set of communication edges (after pruning) are used to update the corresponding entries in the communication matrix $\mathbb{COMM\_MAT}$. An entry (i, j) in the communication matrix represents the amount of the data transferred from a compute node i to j during the program execution. Next, CommAnalyzer accounts for the singleton communication patterns to generate the final communication matrix. While there are several options to synchronize the singleton values, such as broadcast, reduce, allgather, and allreduce, the minimum communication cost to synchronize a singleton value is $O(p-1)$, where p is the number of compute nodes. In particular, source (input), sink (output), and reduction (input/output) singleton values require at least $p-1$, $p-1$, and $2(p-1)$ communication words, respectively. CommAnalyzer uses this lower bound to account for the singleton communication cost.

Finally, CommAnalyzer estimates the communication cost as the overall cost of the above communication matrix, and then computes the communication intensity of the program. The communication intensity of an application is $I_c = C/W$, where C is the communication cost in bytes and W is the work in FLOPs. CommAnalyzer computes W by counting the number of floating-point IR instructions during the program instrumentation.

4.4.3 Implementation and Complexity

We implemented the proposed approach for communication cost estimation on HPC clusters (explained above) using the LLVM compiler infrastructure [113].

CommAnalyzer uses the LLVM front-ends to parse the sequential code and to transform it to the LLVM Intermediate Representation (IR). The main dynamic analysis algorithm is implemented using the dynamic compiler LLI and works on the static single assignment (SSA) form of the LLVM IR. CommAnalyzer instruments the sequential IR and runs it with the provided input data set on top of LLI to characterize its inherent communication. Next, CommAnalyzer generates the value communication graph (VCG) to estimate the communication cost of the sequential IR code on distributed-memory models.

The time complexity of the instrumentation stage (§4.4.1) is linear, as generating the dynamic program slice using the reduced dynamic dependence (RDD) graph method requires a fixed number of operations for each dynamic instruction. The memory complexity of the instrumentation depends on the inherent communication of the sequential code. For embarrassingly-parallel code, the memory complexity is linear (shadow memory), and if the code is fully connected (each value depends on all other values), the memory complexity is quadratic in the worst case (shadow memory and adjacent values). However, in practice, each value depends on a limited number of neighboring values, i.e., the adjacency matrix representation of VCG is sparse even for structured and dense code (see §4.6). Note that the number of vertices of VCG is bounded by the number of memory words as the intermediate values in a single memory location are stored as one vertex in VCG.

The offline analysis (§ 4.4.2) has a time complexity of $O((|V| + |E|) \log p)$ and a memory complexity of $O(|V| + |E|)$, where V and E are the sets of vertices and edges of VCG, and p is the number of distributed-memory compute nodes.

4.5 Efficiency Roofline Model

To show the importance of estimating the communication cost and intensity (I_c) of a given sequential application, this section proposes a high-level model to project the efficiency upper-bound on distributed-memory architectures. Our analysis is inspired by the Roofline model [200] which shows that the performance on shared-memory systems is bounded by the computation intensity of the application, the memory bandwidth, and the floating-point throughput, and ignores several overheads such as the memory access latency, parallel/scheduling overhead, and resource contention. Such simple bound and bottleneck analysis is useful for projecting the performance in the early stages of the development process and for optimizing the actual parallel implementation to minimize the gap between the upper-bound and the achieved performance. Section 4.6 shows the effectiveness of the proposed models for projecting the performance upper-bound of real-world HPC workloads.

In the analysis, we use the following notations:

- n, p: the problem size and the number of compute nodes.
- $T_1(n)$: the single-node execution time.
- $T_o(n, p)$: the parallel overhead function.
- $E(n, p)$: the parallel efficiency, i.e. *speedup*/p.

According to the classical isoefficiency analysis [76, 72], the asymptotic efficiency of a given application on distributed-memory architectures is determined by the growth of the total work performed and the amount of data exchanged with respect to the problem size and the number of nodes. In particular, the parallel efficiency function is given by [2]:

$$E(n, p) = 1/(1 + \frac{T_o(n, p)}{T_1(n)}) \tag{4.1}$$

To project the efficiency upper-bound $E_u(n, p)$, we estimate the lower-bound on the overhead function as follows:

$$T_{o,l}(n, p) = \frac{C \times (1 - O_c)}{B_c} \tag{4.2}$$

where C is the communication in bytes, B_c is the network bisection bandwidth, and O_C is the maximum communication overlap.

The single-node execution time can be formulated as:

$$T_1(n) = W/R \tag{4.3}$$

where W is the work in FLOPs, and R is the single-node throughput.

By substituting equations (4.2) and (4.3) for $T_o(n, p)$ and $T_1(n)$ in equation (4.1), and using the communication intensity definition (i.e., $I_c = C/W$), the efficiency upper-bound is computed as:

$$E_u(n, p) = 1/(1 + \frac{R \times I_c \times (1 - O_c)}{B_c}) \tag{4.4}$$

[2] The readers can refer to [72] for understanding how equation 4.1 is derived from the ratio of the speedup to the number of nodes.

As such, this equation binds the efficiency upper-bound, single-node throughput, communication intensity, and network bandwidth. The communication intensity (I_c) is obtained by CommAnalyzer, while the single-node throughput (R) can be estimated by analytical modeling [176, 43], simulation [25, 31], or simply running the single-node implementation on one node of the target cluster. To figure out the effective bisection bandwidth (B_c) and the maximum communication overlap (O_c), we use Netgauge [89, 90] and Sandia MPI Micro-Benchmark Suite (SMB) [57], respectively. Netgauge measures the bisection bandwidth across a randomly-generated ring process topology to stress the different paths across the network. SMB measures the ability of the MPI library, runtime system, and network hardware to support independent progress of the non-blocking communication (send-recv) operations for different message sizes (8-1M Bytes). This metric is platform-specific and represents the maximum overlap that can be achieved by the application using non-blocking communication.

Finally, the lower-bound on the parallel execution time is:

$$T_{p,l}(n,p) = \frac{T_1(n)}{p \times E_u(n,p)} \tag{4.5}$$

4.6 Evaluation

We illustrate the capabilities of CommAnalyzer using two benchmarks and four real-world HPC applications. Table 4.1 presents the workloads and input data sets considered in the case studies.

In the experiments, we use strong scaling, where the total work is fixed on the different number of compute nodes, as it is the most challenging problem for the parallel efficiency prediction; in particular, at higher node counts, the average parallelism decreases, and the parallel/scheduling overhead and resource contention become the dominant efficiency bottlenecks. In addition, we explore the weak scaling problem for unstructured hydrodynamics code, which is the standard configuration for LULESH on HPC clusters [98].

Table 4.1: Target HPC benchmarks and applications.

Application	**Description**	**Input data**
MatMul	Traditional matrix multiplication	Three matrices of size 4K×4K
SPMV	Sparse matrix vector multiplication with Compressed Sparse Row (CSR) format	Dense square matrix of order 32K
MiniGhost [19]	Representative application of hydrodynamics simulation of solid materials in a 3D grid	3D Grid of size 1K×1K×1K
Heat2D [148]	Canonical heat diffusion simulation in a homogeneous 2D space	2D Grid of size 32K×32K
LULESH [98]	DARPA UHPC hydrodynamics challenge problem for Lagrangian shock simulation on an unstructured hexa-hedral mesh	Strong Scaling Problem: Mesh of size 240^3, Weak Scaling Problem: Mesh of size 120^3
K-means [103]	Unsupervised machine-learning application for data clustering	10M data objects in 34D features space clustered into 5 groups

The estimated communication and performance is evaluated in comparison with the actual MPI+OpenMP implementations. These parallel implementations utilize non-blocking communication to hide the communication latency, and use *MPI_THREAD_FUNNELED* mode, where the main thread executes the MPI calls. The applications use double-precision floating point data types; however, the analysis works with any data type supported by LLVM.

4.6.1 Experimental Setup

As depicted in Figure 4.1, to estimate the communication and scalability at a large-scale problem n, the experiments leverage CommAnalyzer to analyze the sequential code of the target workloads with different problem sizes, where the largest problem size is at most $n/64$ [3]. Using CommAnalyzer's communication cost, scalability analysis tools can project

[3]Additional experiments with small-scale input data sets are available at [81].

the strong and weak scaling of the communication. Specifically, the experiments use Extra-P [29], a scalability analysis tool from Scalasca toolset [68], with the typical settings.

Using the estimated communication cost, we project the efficiency and performance Roofline (see §4.5) on the target HPC cluster. The performance is presented as the grind time, i.e., the execution time/iteration/data element, and it has the same estimation accuracy as the parallel efficiency (see equation 4.5). The reported communication and performance is for the core application kernels and ignores the initialization, setup, and profiling code. The applications run for 100 iterations, and the experiments are repeated 5 times. The error bars show the 95% Confidence Interval.

Test Platform

The test platform is a Linux cluster consists of 196 nodes, and each node contains Intel Xeon processor E5-2683v4 (Broadwell) running at 2.10 GHz. The nodes are connected with an Intel OPA interconnect. The platform uses a batch queuing system that limits the number of nodes per user to 64 nodes. The test system runs CentOS Linux 7 distribution, and the applications are built using gcc 5.2 and OpenMPI 2.0. In the experiments, we launch one MPI process per node and limit the number of cores per process to 16, as using more cores increases the chance of high variance and reduced performance [151, 18].

To evaluate the efficiency Roofline model on the target cluster, we estimate the hardware parameters using the benchmarking approach detailed in §4.5. Specifically, we use Netgauge-2.4.6 [89, 90] and SMB-1.0 [57]. Figure 4.8 shows the effective bisection bandwidth per node on the target cluster. The average value of the communication overlap factor O_c on the test platform is 0.57, i.e., with enough computations to hide the data transfers, the application developer can overlap around 60% of the communication time on average. The application throughput per node $R(n)$ is estimated by running the single-node implementations on one node of the cluster.

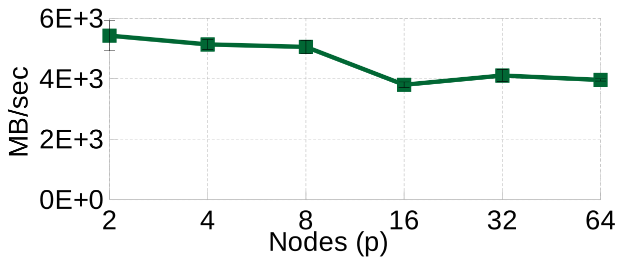


Figure 4.8: The effective bisection bandwidth per node.

Analysis Overhead

As explained in §4.4.3, the time and memory overheads of CommAnalyzer are bounded by the VCG (Value Communication Graph) size, which depends on the inherent flow of information (communication) in the sequential code and can be quadratic in the worst case (if the code is fully connected). Table 4.2 shows the growth of the VCG size with respect to the problem size for the test applications and the size of the largest VCGs used in the experiments[4]. The results show that the VCG size grows much slower than the worst case even for dense linear algebra (MatMul). In most applications, the VCG size grows linearly with the problem size (i.e., each program value is connected to a limited number of neighboring values), while the VCG size of MatMul is approximately $O(n^{1.5})$.

Table 4.2: The size of value communication graph (VCG).

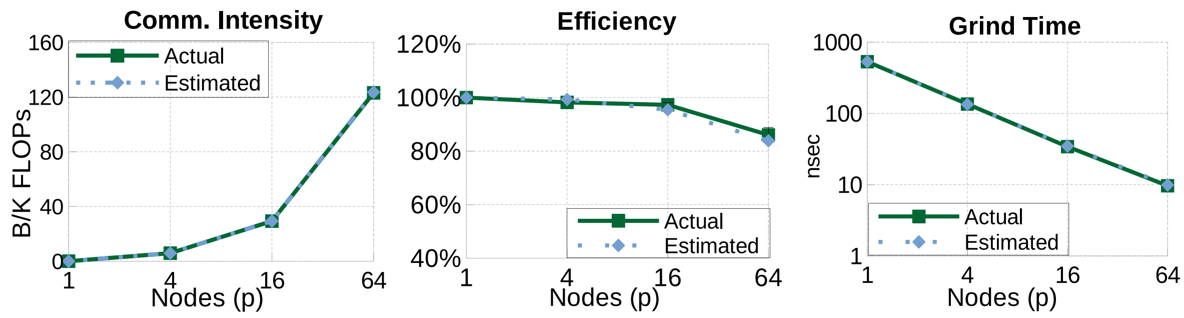
Application	**Problem size n**	**VCG size (n)**	**Largest VCG size**
MatMul	Matrix size	$O(4\ n^{1.5})$	8.01 GBs
SPMV	Sparse matrix size	$O(2\ n)$	0.5 GBs
MiniGhost	Grid size	$O(16\ n)$	4.1 GBs
Heat2D	Grid size	$O(12\ n)$	3 GBs
LULESH	Mesh size	$O(1568.2\ n)$	5.05 GBs
K-means	Data objects size	$O(2\ n)$	0.16 GBs

[4]Detailed examples are available at https://github.com/vtsynergy/CommAnalyzer.

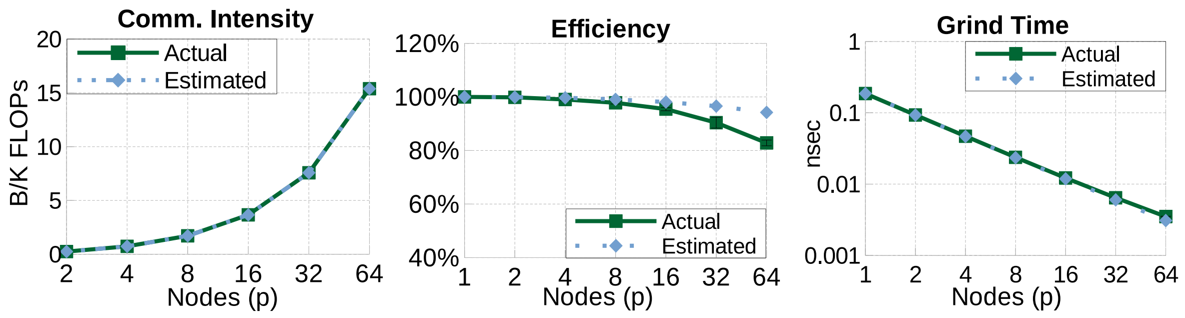
4.6.2 Benchmarks

First, we evaluate the accuracy of CommAnalyzer using two canonical regular and irregular workloads: MatMul and SPMV. MatMul is a traditional matrix multiplication benchmark. The MPI implementation of MatMul uses block-cyclic domain decomposition to distribute the matrices over the compute processes using 2D (square) process topology. SPMV is a sparse matrix-vector multiplication using the compressed sparse row (CSR) format/representation. SPMV is an irregular application with indirect memory accesses through index arrays. To minimize the communication overhead, the MPI implementation of SPMV uses 1D domain decomposition to distribute row blocks of the input sparse matrix and chunks of the input and output vectors. Each process computes a part of the output vector using its row block and the required elements of the input vector, which could exist in other processes. Therefore, the communication cost depends on the density of the input matrix. When the matrix is dense or semi-dense, each process requires $(p-1)$ remote chunks of the input vector, where p is the number of processes, to compute its part of the output vector. Finally, the processes exchange the output vector using collective communication.

Figure 4.9 shows the estimated communication and performance in comparison with the actual MPI+OpenMP implementations for MatMul and SPMV. For the two benchmarks, the actual communication intensity is bounded by 98% of the estimated value. In SPMV, CommAnalyzer detected n output singleton values, where n is the vector size, corresponding to the output vector. In addition, due to the density of the input matrix, CommAnalyzer identified n input singleton values which constitutes the input vector. The actual MPI+OpenMP implementations realize 99% and 96% of the efficiency upper-bound on average for MatMul and SPMV, respectively, while the maximum efficiency gap is 12%. At 64 nodes, SPMV attains 88% of the projected efficiency, as the execution time per iteration drops to a few milliseconds and fixed overheads, such as the parallelization, scheduling, and communication setup overheads, become the dominant efficiency bottlenecks.



(a) MatMul



(b) SPMV

Figure 4.9: The communication cost and performance of MatMul and SPMV benchmarks.

4.6.3 MiniGhost and Heat2D

MiniGhost [19, 20] and Heat2D [148] are two regular HPC applications that use the structured grids design pattern, where a physical space is mapped to a Cartesian grid of points. Typically, the value of grid points represents a material state, such as the temperature, energy, and momentum. MiniGhost is a representative application for 3D hydrodynamics simulation that models the flow and dynamic deformation of solid materials. Heat2D is a canonical heat transfer simulation that solves the Poisson partial differential equations (PDEs) of heat diffusion in a homogeneous 2D space. The two applications use the iterative finite-difference method with an explicit time-stepping scheme to solve the simulation equations. Specifically, Figure 4.10 shows the 3D 7-point and 2D 5-point finite-difference stencils used in MiniGhost and Heat2D, respectively.

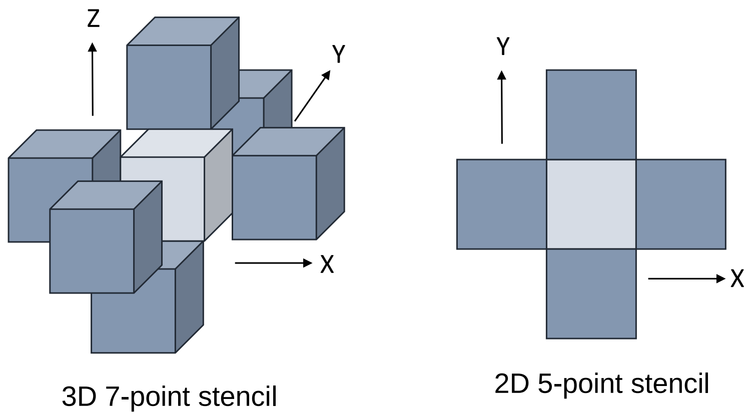
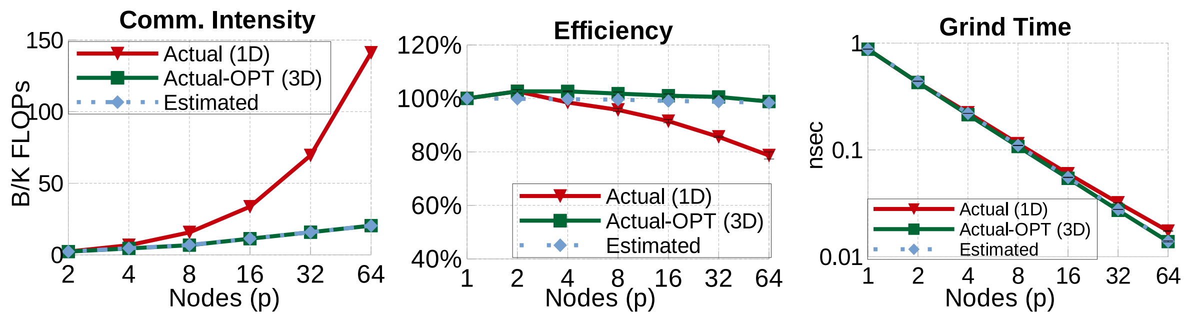


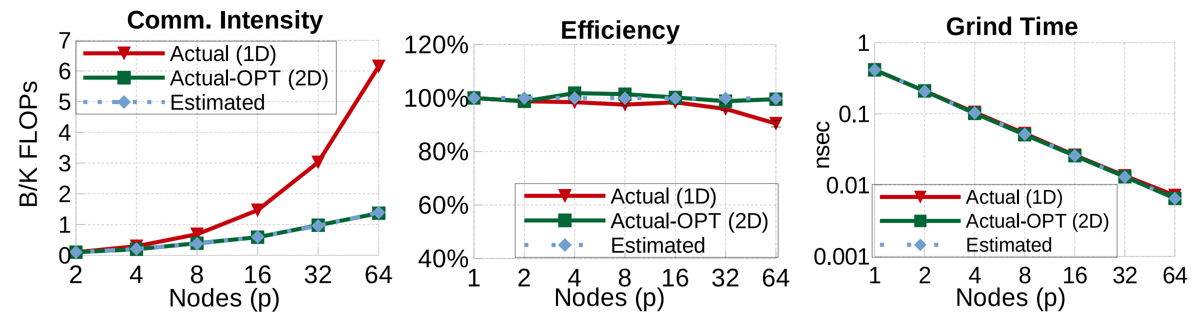
Figure 4.10: The finite difference stencils used in MiniGhost and Heat2D.

Typically, the distributed-memory implementations of structured grids applications use domain decomposition to partition the global grid into multiple sub-grids, and to map each sub-grid to a specific process. To compute the values of the sub-grids in each time step, the processes exchange boundary elements (2D faces and/or 1D lines) with neighbors, which is known as halo exchange. Therefore, the communication volume depends on the surface area of the sub-grid boundaries and the total number of sub-grids (processes).

Figures 4.11 shows the estimated and achieved communication intensity and performance of MiniGhost and Heat2D. The two applications use 1D domain decomposition/process topology by default to reduce the message packing/unpacking overhead, and to ease the programming/debugging effort. However, the results show that the projected efficiency Roofline is only attainable using 3D and 2D domain decomposition/process topologies for MiniGhost and Heat2D, respectively. In particular, when 1D domain decomposition is used by the distributed-memory implementations of MiniGhost and Heat2D, the efficiency gap can be as large as 20% with one order-of-magnitude higher communication intensity. When the optimized MPI implementations use multi-dimensional domain decomposition, the actual communication intensity reaches 97% of the estimated value on average. In some cases, the optimized MPI applications slightly exceed the projected parallel efficiency, due to the improved cache performance in comparison with the single-node implementations.



(a) MiniGhost



(b) Heat2D

Figure 4.11: The communication cost and performance of MiniGhost and Heat2D.

4.6.4 LULESH

LULESH [98] is a Lagrangian shock hydrodynamics simulation that represents more than 30% of the DoD and DoE workloads [144], and one of the five DARPA UHPC challenge problems. LULESH solves the Sedov blast wave problem [172] in 3D space using an unstructured, hexa-hedral mesh. Each point in the mesh is a hexahedron with a center element that represents the thermodynamic variables (e.g., pressure and energy) and corner nodes that track the kinematic variables (e.g., velocity and position); Figure 4.12 shows the 3D view of a point in the mesh. In the Lagrangian hydrodynamics simulation, the mesh follows the motion of the elements in the space and time. LULESH uses an explicit time stepping method (Lagrange leapfrog algorithm); in each time step, it advances the nodal variables, then updates the element variables using the new values of the nodal variables. To maintain the numerical stability, the next time step is computed using the physical constrains on the time step increment of all the mesh elements.

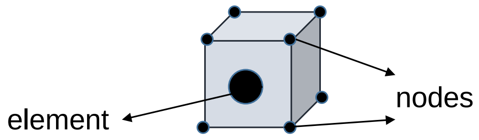
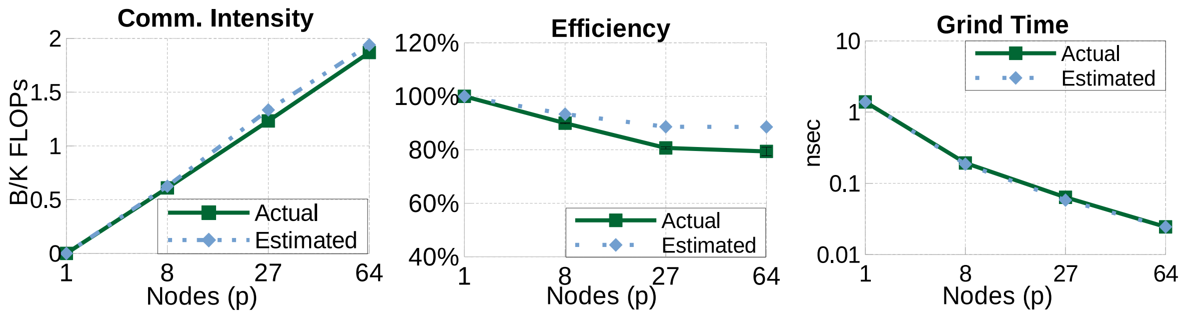


Figure 4.12: LULESH uses an unstructured, hexa-hedral mesh with two centering.

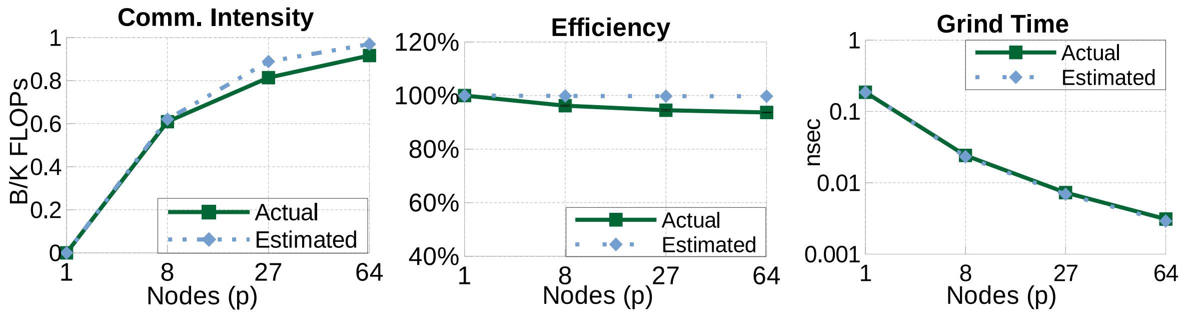
LULESH is a relatively large code with more than 40 computational kernels. It uses indirect memory access pattern via node and element lists and multiple-levels of workspace data structures. The MPI implementation of LULESH uses 3D cube process topology/domain decomposition to distribute the mesh over the available processes, where each process can communicate with up to twenty-six neighbors. In each time step, there are three main communication operations. First, the processes exchange the node-centered boundaries for the positions, acceleration, and force values. Second, they communicate the element-centered boundaries for the velocity gradients. Third, a global collective communication is used to compute the next time step based on the physical constraints of all the mesh elements. In particular, the MPI implementation of LULESH has three different communication patterns: 3D nearest neighbor, 3D sweep, and collective broadcast and reduction [163].

Typically, LULESH uses weak-scaling on HPC clusters [98] because the single-node throughput significantly drops as the problem size decreases due to the parallelization overhead, e.g., additional data motion to handle race conditions. In addition, the message setup time is relatively large due to the packing/unpacking of 12 data fields with different memory-access strides. However, in the experiment, we use both strong and weak scaling to show the efficiency gap in the presence of these overheads.

Figures 4.13 shows the estimated and actual communication intensity and performance of LULESH. The communication intensity of the distributed-memory implementation is within 95% of CommAnalyzer's prediction on average. In the weak-scaling problem, the actual implementation achieves 95% of the projected efficiency on average, and the maximum efficiency gap is 6%. The strong-scaling problem has lower efficiency (as discussed above), achieving 92% of the efficiency Roofline on average with a maximum gap of 11%.



(a) Strong Scaling



(b) Weak Scaling

Figure 4.13: The communication cost and performance of LULESH.

4.6.5 K-means

K-means [103], one of the top 10 algorithms in data mining [204], is an unsupervised machine-learning application for data clustering that has been used in many fields, e.g., pattern recognition, bioinformatics, and statistics (outlier detection). For a given set of data objects in a multi-dimensional feature space, K-means iteratively finds k groups (clusters) of data objects based on feature similarities and generates a cluster membership label for every data object. A data object is considered to be in a cluster c, if it is closer to the centroid (mean) of c than that of any other cluster. In each iteration, K-means computes the centroids (mean values) of the clusters based on their current data objects, and then generates a new membership label for each data object using similarity distance calculation.

The MPI implementation of K-means distributes the data objects evenly among the processes and uses global communication (reduction) to update the centroids of the clusters in every iteration. In particular, the computation cost of K-means is $O(n\ m\ k\ i)$, while its communication cost is $O(m\ k\ i)$, where n is the number of data objects, m is the feature vector size, k is the number of data clusters, and i is the number of iterations. Since n is generally much larger than m and k, the execution time of K-means is bounded by the computation time and achieves linear scaling on distributed-memory architectures.

Figure 4.14 shows the estimated communication and performance of K-means in comparison with the actual distributed-memory implementation. CommAnalyzer accurately detected the input/output singleton program values corresponding to the global (collective) communication required to update the centroids of the data clusters. Overall, the actual communication intensity is bounded by 97% of the predicted value on average. Further, the distributed-memory implementation of K-means attains 99% of the projected efficiency Roofline on average, and the maximum efficiency gap is 3%.

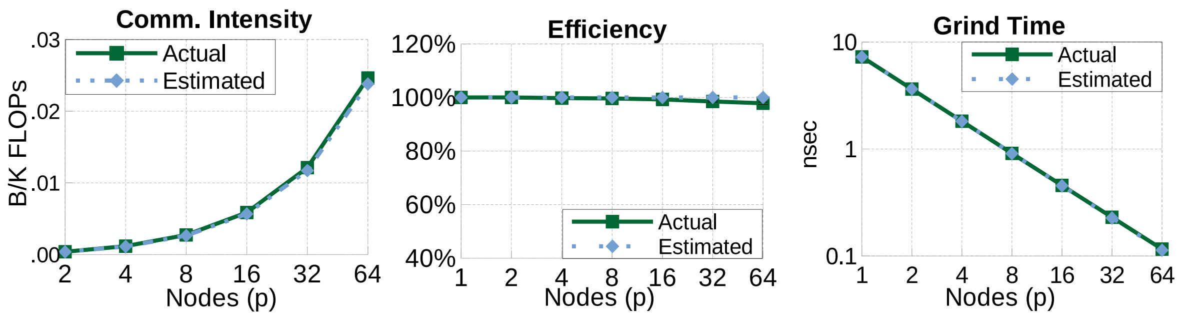


Figure 4.14: The communication cost and performance of K-means.

4.7 Discussion

CommAnalyzer estimates the communication of the distributed-memory parallel code that performs the same work (operations) as the given sequential code. Applying code optimizations that do not change how output values are computed from input values, such as loop tiling and data-layout transformations, does not affect the estimated communication cost. While the original sequential algorithm may not be the best one for parallel execution on distributed-memory systems, the end user can still use CommAnalyzer to estimate the communication cost of the algorithmic alternatives.

Furthermore, CommAnalyzer assumes that the distributed-memory code adopts the SPMD execution model which is the dominant approach on HPC clusters [56, 20]. Other parallel execution models with data migration [28], i.e., the data moves according to the distribution of computations, require a different communication estimation. One possible extension is to use a computation-centric analysis and to estimate the communication as the value-flow edges across compute instructions in different nodes.

Since CommAnalyzer instruments the sequential code to estimate its parallel communication, the problem size is limited by the available memory on a single node of the target cluster. Thanks to scalability analysis tools [29], the user can perform small-scale analysis experiments, which can be dealt with by CommAnalyzer, and project the strong and weak scaling of the communication. Even though we did not encounter any case where the scaling function could not be modeled by the existing scalability analysis tools, more sophisticated modeling techniques, such as machine learning, might be needed.

While CommAnalyzer adopts the sequential multilevel graph partitioning heuristic [100], other parallel and/or approximate graph partitioning heuristics can be used to reduce the analysis overhead. However, this might affect the quality of the VCG decomposition and the resulting accuracy of the communication estimation.

4.8 Conclusion

This chapter presents CommAnalyzer, a novel approach to predict the communication cost and scalability of the sequential code when executed on multiple compute nodes using the SPMD execution model. We implemented CommAnalyzer in the LLVM compiler framework and used novel graph-structured program representations, dynamic instrumentation techniques, and graph analytics algorithms to estimate the minimum communication and maximum efficiency/performance on HPC clusters.

The case studies, with two benchmarks and four real-world applications, demonstrated that CommAnalyzer predicts the communication intensity of sequential applications with more than 95% prediction accuracy on average. Moreover, the optimized distributed-memory (MPI+X) implementations achieve more than 92% of the projected efficiency upper-bound, while the maximum efficiency gap is 12%. The experiments demonstrate the efficacy of CommAnalyzer for regular, irregular, and unstructured problems that have different communication patterns, such as 2D/3D nearest neighbor, 3D sweep/wavefront, 2D broadcast, and collective broadcast and reduction communication.

Chapter 5

Efficient Dependency Management on Massively Data-Parallel Architectures

5.1 Introduction

Irregular computations with fine-grained data dependencies (e.g., iterative and direct solvers for sparse linear systems [51, 166]) constitute the core kernels in many important application domains, such as computational fluid dynamics (CFD), computer-aided design (CAD), data analytics, and machine learning [14, 34, 54, 78, 104, 108, 122]; thus, irregular benchmarks are used in the procurement and ranking of high-performance computing (HPC) systems [59, 134]. However, these important kernels remain challenging to execute on massively parallel architectures, due to the sequential dependencies between the fine-grained application tasks.

Representing such irregular computations as directed acyclic graphs (DAGs), where nodes are compute tasks and edges are data dependencies across tasks, exposes concurrent tasks that can run in parallel without violating the strict partial order in user applications. DAG execution requires mechanisms to determine when a task is ready by tracking the progress of its predecessors (i.e., *dependency tracking*) and by ensuring that all its data dependencies are met (i.e., *dependency resolution*). Thus, the performance of a task-parallel DAG is largely limited by its processing overhead, that is, launching the application tasks and managing their dependencies. Since the target data-dependent kernels consist of fine-grained tasks with relatively few operations, the task-launch latency and dependency-management overhead can severely impact the speedup on massively data-parallel architectures, such as GPUs. Therefore, the efficient execution of *fine-grained, task-parallel* DAGs on *data-parallel* GPU architectures remains an open problem. With the increasing performance and energy efficiency of GPUs [101, 174], driven by the exponential growth of data analytics and machine learning applications [1, 198], addressing this problem has become paramount.

Several software approaches have been proposed to improve the performance of irregular applications with fine-grained, data-dependent parallelism on massively data-parallel GPUs. Level-set methods [11, 168, 136, 137, 118] adopt the bulk synchronous parallel (BSP) execution model [189] by aggregating the independent tasks in each DAG level to execute them concurrently with barrier synchronizations between levels. Hence, these approaches are constrained by the available parallelism in the level-set DAG, which limits their applicability to problems with a short critical path. Furthermore, since the level-set execution manages all the data dependencies using global barriers, it suffers from significant workload imbalance and resource underutilization.

Self-scheduling techniques [169, 41, 119, 120, 8] minimize the latency of task launching by dispatching all the application tasks at once and having them actively wait (spin-loop) until their predecessors complete and the required data is available. However, active waiting not only wastes compute cycles, but it also severely reduces the effective memory bandwidth due to the resource/memory contention. Specifically, the application tasks at lower DAG levels incur substantial active-waiting overhead and interfere with their predecessor tasks, including those on the critical path. Moreover, the application data, along with its task-parallel DAG, must fit in the limited GPU memory, which is typically much smaller than the host memory. To avoid deadlocks, these self-scheduling schemes rely on application-specific characteristics or memory locks [116], which restrict their portability and performance.

Hence, there exists a compelling need for a scalable approach to manage data dependencies across millions of fine-grained tasks on massively data-parallel architectures. To this end, we propose *adaptive task aggregation (ATA)*, a software framework for the efficient execution of irregular applications with fine-grained data dependencies on GPUs. ATA represents such computations as hierarchical DAGs, where nodes are multi-grained application tasks and edges are their aggregated data dependencies, to match the capabilities of GPUs by minimizing the DAG processing overheads while exposing the maximum fine-grained parallelism.

Specifically, ATA ensures deadlock-free execution and performs *multi-level dependency tracking and resolution* to amortize the task launch and dependency management overheads. First, it leverages GPU streams/queues to manage data dependencies across the aggregated tasks [154]. Second, it uses low-latency scheduling and in-device dependency management to enforce the execution order between the fine-grained tasks in each aggregated task. Unlike previous work, ATA is aware of the structure and processing overhead of application DAGs. Thus, ATA provides generalized support for efficient fine-grained, task-parallel execution on GPUs without needing additional hardware logic. In all, our contributions are as follows:

- Unlike previous studies, we show that the performance of a fine-grained, task-parallel DAG depends not only on the problem size and the length of critical path (i.e., number of levels) but also on the DAG shape and structure. We point out that self-scheduling approaches [169, 41, 119, 120, 8] are even worse than the traditional data-parallel execution for problems with a wide DAG (§5.4).
- We propose the adaptive task aggregation (ATA) framework to efficiently execute irregular applications with fine-grained data dependencies as a hierarchical DAG on GPU architectures, regardless of the data-dependency characteristics or the shape of their DAGs (§5.3).
- The experimental results for a set of representative kernels, namely, sparse triangular solve (SpTS) and sparse incomplete LU factorization (SpILU0), across a gamut of problems with different data-dependency structures, show that ATA achieves a geometric mean speedup of 2.2$\times$ to 3.7$\times$ over state-of-the-art DAG execution approaches on AMD GPUs. Thus, ATA delivers substantial improvement in the end-to-end application performance (§5.4).

5.2 Background and Motivation

This section provides background information about sparse solvers as a motivating example for the target workloads, which suffer from limited performance on massively data-parallel GPUs because of their fine-grained data dependencies.

The iterative [166] and direct methods [51] for solving sparse linear systems generally consist of two phases: (1) a preprocessing phase that is performed only once to analyze and exploit the underlying sparse structure and (2) a system solution phase that is repeated several times. The system solution phase is typically dominated by irregular computations with data-dependent parallelism, namely, preconditioners and triangular solve in iterative methods and matrix factorization/decomposition and triangular solve in direct methods. Such data-dependent kernels can be executed in parallel as a computational DAG, where each node represents the compute task associated with a sparse row/column and edges are the dependencies across tasks.

Algorithm 5 and Figure 5.1 show an example of the irregular computations in sparse solvers. In SpTS, each nonzero entry (i, j) in the triangular matrix indicates that the solution of unknown i (task u_i) depends on the solution of unknown j (task u_j); hence, the DAG representation of SpTS associates an edge from node u_j to node u_i. The resulting DAG can be executed using a push or pull traversal [140]. In push traversal, the active tasks push their results and active state to the successor tasks; while in pull traversal, the active tasks pull the results from their predecessor tasks. In addition to the representative SpTS and SpILU0 kernels that are extensively discussed in this work, several sparse solver kernels (e.g., LU/Cholesky factorization, Gauss-Seidel, and successive over-relaxation [51, 166, 2]) exhibit similar irregular computations.

To execute a task-parallel DAG on GPUs using the data-parallel BSP model, the independent tasks in each DAG level are aggregated and executed concurrently with global barrier synchronization between the different levels. (This parallelization approach is often called

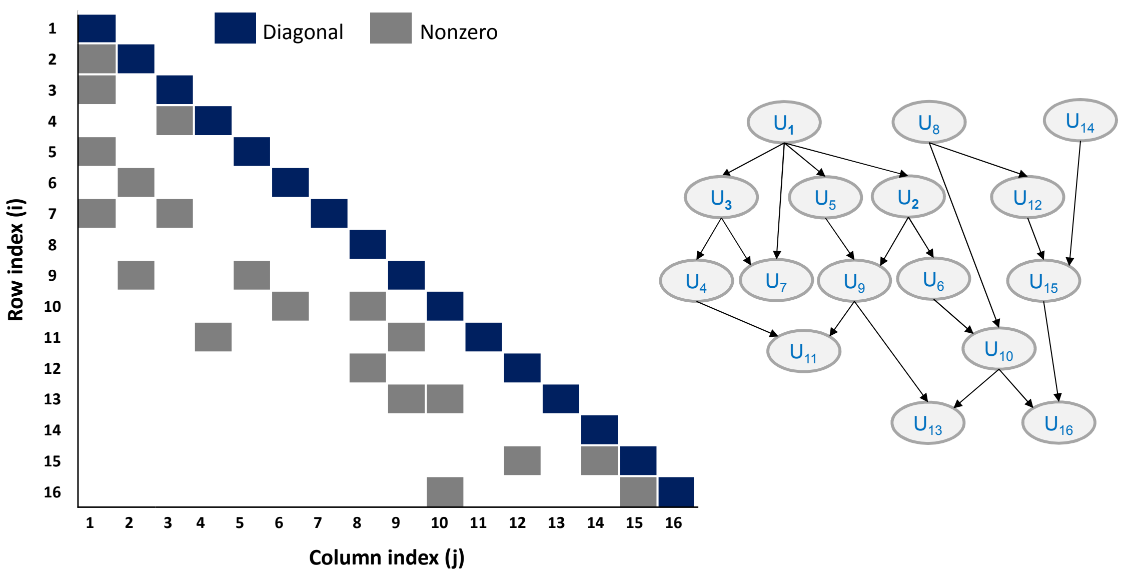


Figure 5.1: A triangular matrix and the corresponding DAG for SpTS.

Algorithm 5 Sparse Triangular Solve (SpTS)

```
Input: L, RHS                                  ▷ Triangular matrix and right-hand side vector
Output: u                                      ▷ Solution vector for the unknowns
1: for $i = 1$ to $n$ do
2:     $u(i) = RHS(i)$
3:     for $j = 1$ to $i - 1$ where $L(i,j) \neq 0$ do         ▷ Predecessor unknowns
4:         $u(i) = u(i) - L(i,j) \times u(j)$
5:     end for
6:     $u(i) = u(i)/L(i,i)$
7: end for
```

level-set execution or wavefront parallelism [166, 119].) For example, as depicted in Figure 5.2, the BSP execution of the DAG in Figure 5.1 runs tasks U_1, U_8, and U_{14} first, while the rest of tasks will wait for their completion at the global barrier. Since the local dependencies between tasks are replaced with global barriers, the BSP execution of a DAG suffers from barrier synchronization overhead, workload imbalance, and idle/waiting time. Furthermore, the GPU performance becomes even worse for application DAGs with limited parallelism and a few compute operations per task [54, 197]. At this fine granularity, the dispatch, scheduling, and dependency management overheads become the dominant bottlenecks.

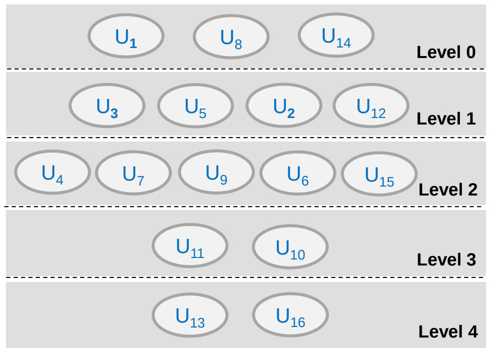

U_1 U_8 U_{14} Level 0
U_3 U_5 U_2 U_{12} Level 1
U_4 U_7 U_9 U_6 U_{15} Level 2
U_{11} U_{10} Level 3
U_{13} U_{16} Level 4

Figure 5.2: The BSP (level-set) execution of the DAG in Figure 5.1.

5.3 Adaptive Task Aggregation (ATA)

To address the limitations of the traditional data-parallel execution and previous approaches for fine-grained, task-parallel applications, we propose the *adaptive task aggregation (ATA)* framework. The main goal of ATA is to efficiently execute irregular computations, where the parallelism is limited by data dependencies, on throughput-oriented, many-core architectures with thousands of threads. On the one hand, there is a tradeoff between the task granularity and concurrency; that is, the maximum parallelism and workload balance are only attainable at the finest task granularity (e.g., a sparse row/column in sparse solvers). On the other hand, the overhead of managing data dependencies and launching ready tasks at this fine-grained level can adversely impact the overall performance.

Thus, ATA strives to dispatch the fine-grained application tasks, as soon as their data dependencies are met, to the available compute units (CUs) with minimal overhead and regardless of the DAG structure of the underlying problem. First, ATA represents the irregular computations as a hierarchical DAG by means of dependency-aware task aggregation for high-performance execution on GPUs (§5.3.1). Second, it ensures efficient, deadlock-free execution of the hierarchical DAG using multi-level dependency management and sorted eager-task (SET) scheduling (§5.3.2). Furthermore, ATA supports both the push and pull execution models of task-parallel DAGs and works on current GPU architectures without the need for special hardware support. While any input/architecture-aware task aggregation can be used to benefit from ATA's hierarchical execution and efficient scheduling and dependency management, we propose concurrency-aware and locality-aware aggregation policies to provide additional performance trade-offs (§5.3.3).

5.3.1 Hierarchical DAG Transformation

The first stage of our ATA framework analyzes the given fine-grained DAG and then generates a hierarchy of tasks to better balance the processing overheads, that is, task launch and dependency management overheads, while exposing the maximum parallelism to many-core GPUs. This transformation can be incorporated in the preprocessing phase of irregular applications, such as sparse solvers, with negligible additional overhead (see §5.4).

Consider an application DAG, $G(U, E)$, where U is a set of nodes that represents user (or application) tasks and E is a set of edges that represents data dependencies. Further, let n be the number of user tasks and m be the number of dependency edges across user tasks. ATA aggregates user tasks into adaptive tasks such that each adaptive task has a positive integer number S of the fine-grained user tasks, where S is an architecture-dependent parameter that can be estimated and tuned using profiling (as detailed in §5.3.3). The resulting set A of adaptive tasks partitions the application DAG such that $A_1 \cup A_2 \cdots \cup A_p = U$ and $A_i \cap A_j = \phi \ \forall i$ and j, where p is the number of adaptive tasks, $p \leq n$, and $i \neq j$.

This task aggregation delivers several benefits on many-core GPUs. First, the resulting adaptive tasks incur a fraction $(1/S)$ of the launch overhead of user tasks. Second, adaptive tasks reduce the execution latency of their user tasks by dispatching the irregular computations to CUs as soon as their pending coarse-grained dependencies are resolved. Third, task aggregation eliminates dependency edges across user tasks that exist in different adaptive tasks, such that an adaptive task with independent user tasks does *not* require any dependency management. Hence, ATA generates a transformed DAG with c coarse-grained dependencies across adaptive tasks and f fine-grain dependencies across user tasks that exist in the same adaptive task, where $c + f < m$.

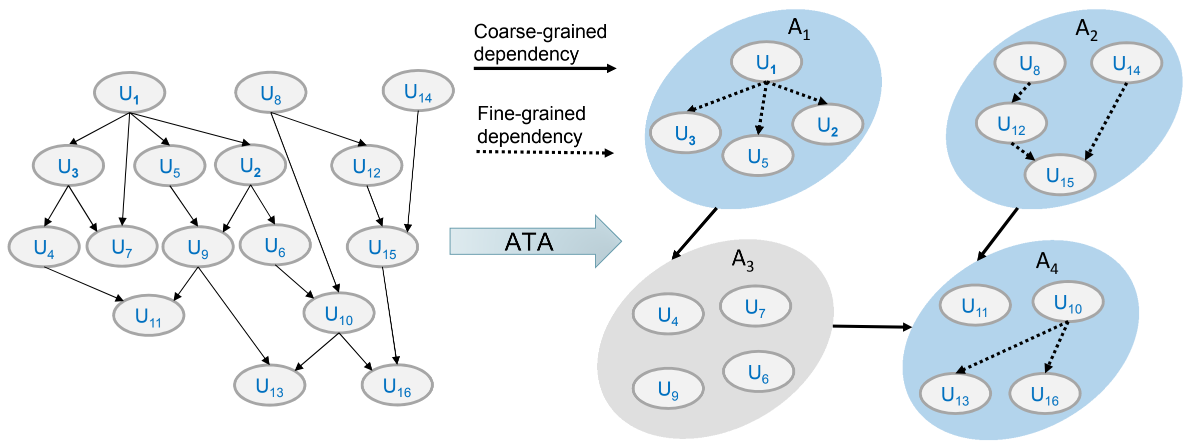


Figure 5.3: ATA transformation of the application DAG in Figure 5.1 for hierarchical execution and dependency management. The adaptive tasks A_1, A_2, and A_4 require fine-grained dependency tracking and resolution, while A_3 can be executed as a data-parallel kernel.

Figure 5.3 shows an example of the DAG transformation with an arbitrary task aggregation policy (see §5.3.3 for our proposed policies). The original DAG consists of 16 user tasks with 20 dependency edges; after the DAG transformation such that each adaptive task has four user tasks, ATA generates a hierarchical DAG that consists of four adaptive tasks with only three coarse-grained dependency edges and eight fine-grained dependency edges. Since the DAG processing overhead depends on the number of tasks and dependency edges, the transformed DAG is more efficient for execution on GPU architectures. Specifically, unlike level-set execution, which is constrained by managing all data dependencies using global barriers, ATA can launch more tasks per GPU kernel to amortize the cost of kernel launch and to reduce the idle/waiting time. Most importantly, compared to self-scheduling approaches, ATA adjusts to the underlying dependency structure of target problems by executing adaptive tasks without dependency management when it is possible and by dispatching the waiting user tasks when there is limited concurrency to efficiently utilize the GPU resources. That way, ATA dispatches the ready adaptive tasks rather than the whole DAG, and as a result, the waiting adaptive tasks along with their user tasks do not incur any active-waiting overhead.

Previous work showed the benefits of aggregating the fine-grained application tasks on CPU architectures [149]; however, each aggregated task was assigned to one thread/core to execute *sequentially* without the need for managing data dependencies across its fine-grained computations. In contrast, GPU architectures (with their massive number of compute units) demand *parallel* execution both within and across aggregated tasks, which in turn introduces several challenges and requires an efficient approach for managing the data dependencies and executing the irregular computations at each hierarchy level of the transformed DAG.

5.3.2 Hierarchical DAG Execution on GPUs

The ATA framework orchestrates the processing of millions of fine-grained user tasks, which are organized into a hierarchical DAG of adaptive tasks. The adaptive tasks execute as GPU kernels on multiple compute units (CUs), while their user tasks run on the finest scheduling unit defined by the GPU architecture (such as wavefronts) to improve workload balance and to expose maximum parallelism.

ATA performs hierarchical dependency management by tracking and resolving the data dependencies at two levels: (1) a coarse-grained level across adaptive tasks and (2) a fine-grained level across user tasks in the same adaptive task. The coarse-grained dependency management relies on host- or device-side streams/queues to monitor the progress of GPU kernels that represent adaptive tasks and to dispatch the waiting adaptive tasks to the GPU device once their coarse-grained dependencies are met. Currently, ATA leverages the open-source ATMI runtime [154] to dispatch the adaptive tasks and to manage the coarse-grained (kernel-level) dependencies.

The fine-grained dependency management requires a low-latency approach with minimal overhead, relative to the execution time of the fine-grained user tasks. Thus, ATA manages the fine-grained dependencies using lock-free data structures, where each user task tracks and resolves its pending dependencies using active waiting (i.e., polling on the shared data structures) to enforce the DAG execution order. Most importantly, ATA ensures forward progress and minimizes the active waiting overhead by assigning the waiting user tasks that are more likely to meet their dependencies sooner to the active scheduling units (wavefronts) on a GPU using SET scheduling.

SET Scheduling

To efficiently execute adaptive tasks on many-core GPUs, we propose sorted eager task (SET) scheduling, which aims to minimize the processing overhead by eliminating the launch and dependency resolution overheads using eager task launching and by minimizing the dependency-tracking overhead using an implicit priority scheme.

Figure 5.4 shows the SET scheduling of a hierarchical DAG with 16 user tasks and four (4) adaptive (aggregated) tasks. First, SET dispatches all the user tasks in an adaptive task as a GPU kernel to eliminate the task launch overhead. That way, the entire adaptive task can be processed by the GPU command processor (CP) to assign its user tasks to CUs before their predecessors even complete. However, user tasks with pending dependencies check that their predecessors finish execution using active waiting to prevent data races. Once the predecessors of a waiting user task complete, it becomes immediately ready and proceeds for execution, which eliminates the dependency-resolution overhead. To ensure forward progress, the waiting user tasks cannot be scheduled on a compute unit before their predecessors are active. While hardware memory locks [116] can be used to avoid deadlocks, they are not suitable for scheduling *large-scale* DAGs with *fine-grained* tasks because of their limited number and significant scheduling latency. In contrast, SET proposes a priority scheme for deadlock-free execution with minimal overhead and without needing specialized hardware.

SET prioritizes the execution of the waiting user tasks that are more likely to be ready soon to minimize the dependency-tracking (active waiting) overhead and to prevent deadlocks. However, current many-core architectures do not provide a priority scheme with enough explicit priorities to handle a large number (potentially millions) of tasks. Thus, SET uses a more implicit technique and exploits the knowledge that hardware schedulers execute

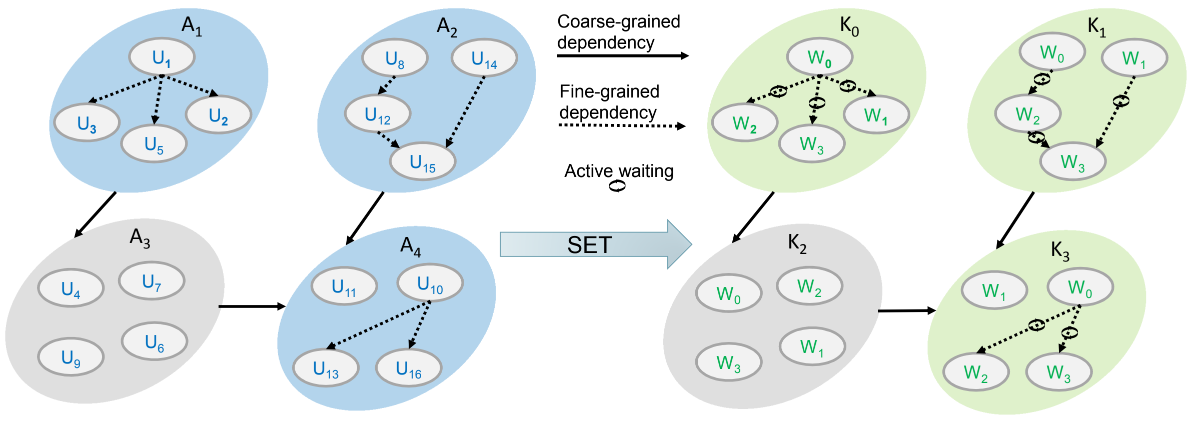


Figure 5.4: SET scheduling of the hierarchical DAG in Figure 5.3. Each adaptive (A) task executes as a GPU kernel (K) with fine-grained dependency management using active waiting when deemed necessary. SET scheduling ensures forward progress by mapping the user (U) tasks to the worker wavefronts (W).

wavefronts and workgroups with lower global ID first. According to GPU programming and execution specifications, such as the HSA programming manual [161], only the oldest workgroup (and its wavefronts) is guaranteed to make forward progress; hence, the workgroup scheduler dispatches the oldest workgroup first when there are enough resources on target CUs. Moreover, the wavefront scheduler runs a single wavefront until it stalls and then picks the oldest ready wavefront [162]. As a result, the oldest hardware scheduling units (wavefronts) with the smallest global IDs are implicitly prioritized.

Therefore, SET assigns user tasks with *fewer number of dependency levels* to *older wavefronts*. Since GPU hardware schedules concurrent wavefronts to maximize resource utilization as noted in §5.3.3, the dependency level of a user task approximates its waiting time for dependency resolution. If there are multiple user tasks with the same number of dependency levels, SET assigns neighboring user tasks to adjacent wavefronts to improve data locality. For example, in Figure 5.4, U_1 is a root task (no predecessors), while U_3, U_5, and U_2 have one level of data dependency; hence, SET assigns U_1, U_3, U_5, and U_2 to the worker wavefronts W_0, W_2, W_3, and W_1 in kernel K_0. Since all U tasks in A_3 are independent, SET executes A_3 without any dependency tracking and resolution and assigns the neighboring U_4, U_6, U_7, and U_9 tasks to the adjacent W_0, W_1, W_2, and W_3 wavefronts in K_2.

Push vs. Pull Execution within Adaptive Tasks

Algorithm 6 Push or pull execution of adaptive tasks on GPU architectures.

```
Require: app_data                         ▷ For example, a sparse matrix.
Require: SET_sched                        ▷ Schedule of U tasks on worker wavefronts.
Require: u_deps                           ▷ No. of pending dependencies for each U task.
Require: u_done                           ▷ The state of each U task.
 1: for ∀ U tasks in parallel do
 2:     $i$ = GET_UTASK(SET_sched)
 3:     while ATOMIC(u_deps($i$) $\neq 0$) do          ▷ Active waiting.
 4:         NOOP
 5:     end while
 6:     Compute task $i$ on the worker SIMD units
 7:     for each $j$ successor of task $i$ do
 8:         ATOMIC(u_deps($j$) = u_deps($j$) $- 1$)
 9:     end for
10:     for each $j$ predecessor of task $i$ do
11:         while ATOMIC(u_done($j$) $\neq 1$) do      ▷ Active waiting.
12:             NOOP
13:         end while
14:         Perform ready ops. of task $i$ on worker SIMD units
15:     end for
16:     ATOMIC(u_done($i$) = 1)
17: end for
```

ATA supports both the push and pull execution models of a computational DAG. Algorithm 6 shows the high-level (abstract) execution of adaptive tasks with fine-grained data dependencies using push or pull models (as indicated by the different gray backgrounds[1]).

In push execution, ATA uses an auxiliary data structure (*u_deps*) to manage the fine-grained data dependencies by tracking the number of pending dependencies for each user task. Once

[1]The lines with white background are required for both push and pull models.

all dependencies are met, user tasks can proceed to execute on the SIMD units of their worker wavefront. (The assignment of user tasks to worker wavefronts is determined by the SET schedule.) When a user task completes its operations, it pushes the active state to its successors by decreasing their pending dependencies. Hence, push execution often needs many atomic write operations.

Conversely, the pull execution model tracks the active state of user tasks using the *u_done* data structure. As such, each user task pulls the state of its predecessors and cannot perform the dependent computations until the predecessor tasks finish execution. Once a user task completes, it updates the corresponding state in *u_done*. Thus, pull execution performs more read operations compared to the push model. However, it can pipeline the computations (lines 10 and 14 in Algorithm 6) to hide the memory access latency.

5.3.3 Task Aggregation Policies

Finding the optimal granularity of a given application's DAG on a many-core GPU is a complicated process. First, the active waiting (dependency tracking) overhead increases with the size of aggregated tasks. In addition, a user task on the critical path can delay the execution of its aggregated task, including the other co-located user tasks. On the other hand, as the size of aggregated tasks becomes larger, the cost of managing their coarse-grained dependencies and launching user tasks decreases; moreover, increasing the size of aggregated tasks reduces the idle/waiting time, including dependency resolution time, which improves the resource utilization. Therefore, optimal task aggregation requires detailed application and architecture modeling as well as sophisticated tuning and profiling. However, by leveraging the knowledge of the target hardware architecture and application domain, simple cross-layer models and heuristics can achieve near-optimal performance.

Unlike CPU architectures, GPUs are throughput-oriented and rely on massive multithreading (i.e., dispatching more threads/wavefronts than the available compute resources) to maximize resource utilization and to hide the execution and memory-access latencies [66]. This massive multithreading is possible due to the negligible scheduling overhead between stalled wavefronts and other active wavefronts. Thus, the GPU hardware can be efficiently used, if and only if, enough concurrent wavefronts are active (or in-flight). Hence, if each user task executes on a wavefront, the minimum size of an adaptive task, S_{min}, is limited by the number of CUs and the occupancy (active wavefronts per CU) of the GPU device:

$$S_{min} = num\_CU \times occupancy \tag{5.1}$$

As detailed before, increasing the size of an adaptive task has several side effects. However, any aggregation heuristic should ensure that the size of an adaptive task is large enough to amortize the cost of launching the aggregated tasks and tracking their progress. On GPUs, such cost is typically dominated by launching the aggregated tasks as GPU kernels (T_l). If the average execution time of a user task is $\overline{T_u}$, the size of an adaptive task can be tuned as follows:

$$S = R \times (T_l/\overline{T_u}), \quad S > 1 \text{ and } R > 0 \tag{5.2}$$

The above equation indicates that the execution time of an aggregated task should be much larger than its launch cost. Typically, R is selected such that T_l is less than 1% of the average time of an adaptive task, while the execution time of user tasks can be estimated by profiling them in parallel to determine $\overline{T_u}$. Since the dependency management overhead can be several orders of magnitude higher than the execution time of user tasks (as shown in §5.4), and the profiling is performed only once in the preprocessing phase, the additional profiling overhead is negligible.

In summary, the proposed heuristic for tuning the granularity/size (S) of adaptive tasks, using equations (5.1) and (5.2), ensures that the performance is limited by the inherent application dependencies rather than resource underutilization, idle/waiting time, or kernel launch cost. Once the granularity of adaptive tasks is selected, different task aggregation mechanisms can be used with additional performance trade-offs. In particular, we propose the following concurrency-aware and locality-aware aggregation techniques, which are formally detailed in Algorithms 7 and 8.

Algorithm 7 Concurrency-aware (CA) Aggregation

```
Require: u_levels                              ▷ User tasks in each DAG level.
Require: S                           ▷ Granularity/size of adaptive tasks.
Ensure: a_tasks                                         ▷ Adaptive tasks.
 1: $a\_task$ = GET_CURRENT_ATASK(a_tasks)
 2: for ∀ U levels do
 3:     $i$ = GET_LEVEL(u_levels)
 4:     ADD_UTASKS(a_task, u_levels(i))                ▷ Aggregate U tasks.
 5:     if SIZE(a_task) $\geq S$ then
 6:         $a\_task$ = CREAT_ATASK(a_tasks)
 7:     end if
 8: end for
```

Concurrency-aware (CA) Aggregation. ATA aggregates user tasks starting from the root DAG level before moving to the next levels. If the current DAG level has more than S user tasks, where S is the size (granularity) of adaptive tasks, ATA launches this level as an adaptive task. Otherwise, it merges the next DAG level in the current adaptive task and continues aggregating. That way, adaptive tasks end up having at least a size of S user tasks. Such an aggregation mechanism increases concurrency among user tasks in the same adaptive task and minimizes the overall critical path of the resulting hierarchical DAG; however, it ignores data locality.

Locality-aware (LA) Aggregation. This aggregation policy improves data locality across the memory hierarchy by merging neighboring user tasks into the same adaptive task, which can benefit applications with high spatial locality. The task locality information is based on knowledge of standard data formats (e.g., sparse matrix representations) and it can also be incorporated as a programmer hint. Unlike the CA approach, LA aggregation may increase the overall critical path of hierarchical DAGs, as a user task on the critical path can delay the execution of neighboring user tasks.

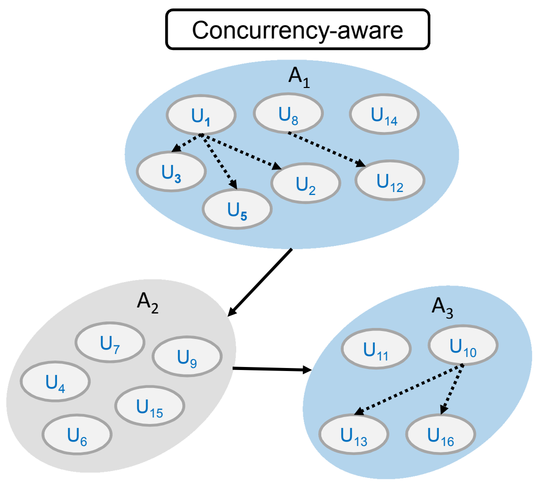
Algorithm 8 Locality-aware (LA) Aggregation

```
Require: u_tasks, u_loc                              ▷ User tasks and their locality info.
Require: S                                           ▷ Granularity/size of adaptive tasks.
Ensure: a_tasks                                      ▷ Adaptive tasks.
1: $a\_task$ = GET_CURRENT_ATASK(a_tasks)
2: for ∀ U tasks do
3:     $i$ = GET_U_ID(u_loc)
4:     ADD_UTASK(a_task, u_tasks(i))                 ▷ Aggregate U tasks.
5:     if SIZE(a_task) $\geq S$ then
6:         $a\_task$ = CREAT_ATASK(a_tasks)
7:     end if
8: end for
```

Figure 5.5 shows an example of the different aggregation policies, where the adaptive task granularity is four (4) user tasks. Due to the limited concurrency at the root DAG level, CA aggregation combines this level and the next one into the adaptive task A_1. Next, it encapsulates the third DAG level in A_2 which does not require any fine-grained dependency management. Finally, CA aggregation merges the fourth and fifth DAG levels in A_3 to reach the required granularity. In contrast, LA aggregation merges the neighboring user tasks into four adaptive tasks. While CA aggregation achieves the same critical path as the original application DAG, that is, five user tasks ($U_1 \rightarrow U_2 \rightarrow U_6 \rightarrow U_{10} \rightarrow U_{16}$), the resulting hierarchical DAG from LA aggregation has a longer critical path of nine user tasks ($U_1 \rightarrow U_3 \rightarrow U_4 \rightarrow U_5 \rightarrow U_9 \rightarrow U_{11} \rightarrow U_{14} \rightarrow U_{15} \rightarrow U_{16}$).

We also considered greedy aggregation, which combines the maximum number of user tasks that can fit on the GPU[2] in a single adaptive task. Compared to other aggregation policies, greedy aggregation does *not* adapt to the DAG structure, leading to excessive active waiting for application DAGs with high concurrency, as adaptive tasks are unlikely to execute without needing a fine-grained dependency management.

[2]This number is limited by the available memory and maximum number of active wavefronts on the GPU.



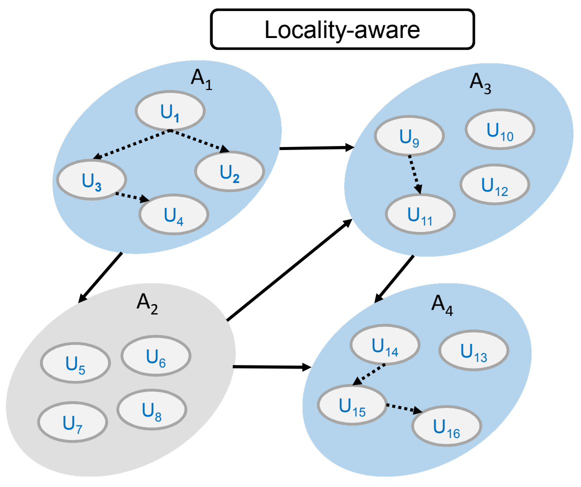


Figure 5.5: Concurrency- and locality-aware aggregations of the application DAG in Figure 5.1. The adaptive task granularity is four.

5.4 Evaluation

We evaluate the performance of the proposed ATA framework using a set of important and representative kernels with fine-grained data dependencies. These kernels implement the sparse triangular solve (SpTS) and sparse incomplete LU factorization with zero level of fill in (SpILU0) algorithms, which are detailed in Algorithms 5 and 9. Specifically, we consider the push and pull execution variants of SpTS using the compressed sparse column (CSC) and compressed sparse row (CSR) formats, respectively, and the left-looking pull execution of SpILU0 using the CSC format [51, 166]. In addition, we evaluate the end-to-end application performance using the preconditioned conjugate gradient (PCG) sparse solver [166].

We compare ATA to the level-set execution [11, 168, 136, 137, 118] and self-scheduling approaches [169, 41, 119, 120]. The target GPU kernels are implemented in OpenCL, while the host code is written in C++ and leverages the open-source ATMI runtime [154] to dispatch GPU kernels. Using double-precision arithmetic, we report the performance and overhead numbers for the system solution phase as an average over 100 runs[3]. It is important

[3]The reported performance for SpTS (push traversal) with self-scheduling approach is based on executing the OpenCL code from Liu et al. [119, 120].

Algorithm 9 Sparse Incomplete LU Factorization with zero level of fill in (SpILU0)

```
Require: A                                      ▷ Input matrix that will be decomposed into L and U
 1: for $j = 1$ to $n$ do                                                        ▷ Current column
 2:     for $k = 1$ to $j - 1$ where $A(k, j) \neq 0$ do                          ▷ Predecessors
 3:         for $i = k + 1$ to $n$ where $A(i, k)$ & $A(i, j) \neq 0$ do
 4:             $A(i, j) = A(i, j) - A(i, k) \times A(k, j)$                        ▷ Elimination
 5:         end for
 6:     end for
 7:     for $i = j + 1$ to $n$ where $A(i, j) \neq 0$ do
 8:         $A(i, j) = A(i, j)/A(j, j)$                                          ▷ Normalization
 9:     end for
10: end for
```

to note that the different DAG execution approaches, namely, ATA, level-set, and self-scheduling, generate identical results using the same computations and only differ in the data-dependency management, as detailed in the previous sections.

5.4.1 Experimental Setup

Input Data

The experiments consider representative problems with different sizes and data dependency structures that cover a wide range of application domains, including computational fluid dynamics, electromagnetics, mechanics, atmospheric models, structural analysis, thermal analysis, power networks, and circuit simulation [52]. Table 5.1 presents the characteristics of the test problems, where the problem ID is assigned in an ascending order of the number of unknowns. Further, to clarify the experimental results, we classify the resulting application DAG of the input problems into wide DAG, L-shape DAG, and parallel DAG. Figure 5.6 shows an example of these different DAG types. The parallel DAG has a short critical path

Table 5.1: Characteristics of the sparse problems.

Prob. ID	**Name**	**#unknowns**	**#nonzeros**
P1	onetone2	36,057	222,596
P2	onetone1	36,057	335,552
P3	TSOPF_RS_b300_c3	42,138	4,413,449
P4	bcircuit	68,902	375,558
P5	circuit_4	80,209	307,604
P6	ASIC_100ks	99,190	578,890
P7	hcircuit	105,676	513,072
P8	twotone	120,750	1,206,265
P9	FEM_3D_thermal2	147,900	3,489,300
P10	G2_circuit	150,102	726,674
P11	scircuit	170,998	958,936
P12	hvdc2	189,860	1,339,638
P13	thermomech_dK	204,316	2,846,228
P14	offshore	259,789	4,242,673
P15	ASIC_320ks	321,671	1,316,085
P16	rajat21	411,676	1,876,011
P17	cage13	445,315	7,479,343
P18	af_shell3	504,855	17,562,051
P19	parabolic_fem	525,825	3,674,625
P20	ASIC_680ks	682,712	1,693,767
P21	apache2	715,176	4,817,870
P22	ecology2	999,999	4,995,991
P23	thermal2	1,228,045	8,580,313
P24	atmosmodd	1,270,432	8,814,880
P25	G3_circuit	1,585,478	7,660,826
P26	memchip	2,707,524	13,343,948
P27	Freescale2	2,999,349	14,313,235
P28	Freescale1	3,428,755	17,052,626
P29	circuit5M_dc	3,523,317	14,865,409
P30	rajat31	4,690,002	20,316,253

(typically less than 100 user tasks) such that the performance is bounded by the execution time rather than the data dependencies. In L-shape DAGs, most of the user tasks are in the higher DAG levels and the number of concurrent user tasks significantly decreases as we move down the critical path. Conversely, in wide DAGs, the majority of DAG levels are wide with enough user tasks to utilize at least the available SIMD elements in each compute unit (i.e., four wavefronts per CU in target GPUs).

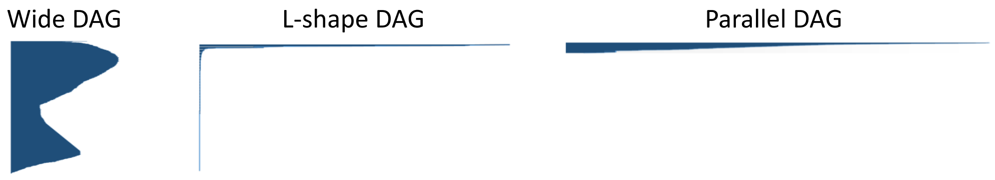


Figure 5.6: An example of the different DAG classes. The x-axis shows the number of user tasks, while the y-axis represents the DAG levels (critical path).

Test Platform

The test platform is a Linux server with an Intel Xeon E5-2637 CPU host running at 3.50 GHz and multiple GPU devices. The server runs the Debian 8 distribution and ROCm 1.8.1 software stack, and the applications are built using GCC 7.3 and CLOC (CL Offline Compiler) 1.3.2. In the experiments, we consider two different generations of AMD GPU devices: Radeon Vega Frontier Edition [10] (VEGA-FE) and Radeon R9 Nano [125] (R9-NANO). Table 5.2 details the specifications of the target GPUs. For brevity, we only show the detailed results for the VEGA-FE GPU. In addition, we use micro-benchmarks to profile the overhead of atomic operations and kernel launch.

Table 5.2: Target GPU architectures.

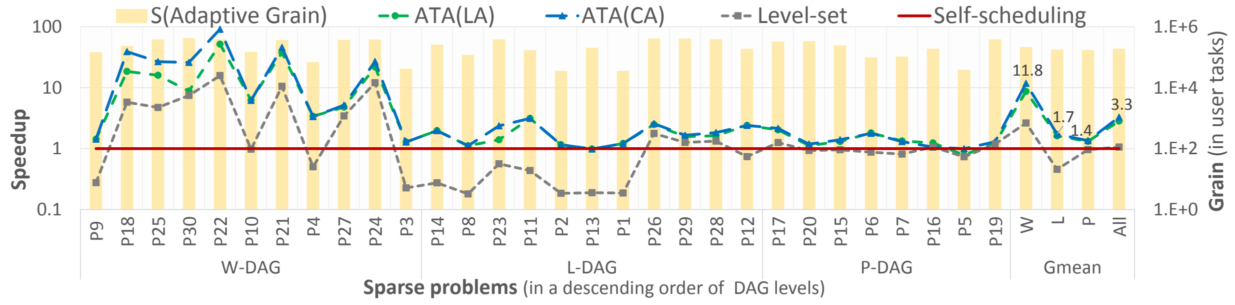
GPU	**Max. freq.**	**Memory**	**Mem. BW**	**#cores**
R9-NANO	1000 MHz	4 GB	512 GB/s	4,096
VEGA-FE	1600 MHz	16 GB	483 GB/s	4,096

5.4.2 Experimental Results

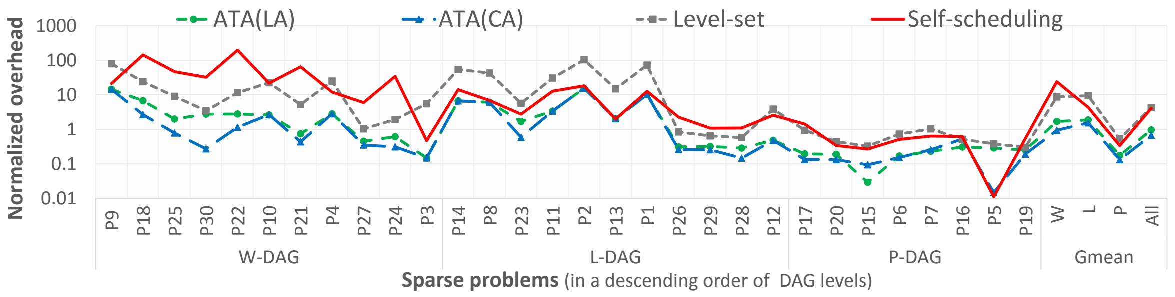
The results reported here demonstrate the capabilities of the ATA framework with its different aggregation policies, where the adaptive task granularity (S) is selected using the profiling-based heuristic from Eq. (5.1) and (5.2). In the experiments, we set R to 100 in Eq. (5.2) to ensure that the overhead of coarse-grained dependency management across adaptive tasks is less than 1% of their average execution time. To measure the overhead of managing the data dependencies of the application DAGs, we execute these DAGs without any dependency management and with the different DAG execution approaches. Such overhead represents the kernel launch and workload imbalance (global synchronization) for level-set execution and the active waiting for self-scheduling methods, while it shows the processing cost of hierarchical DAGs for ATA execution as illustrated in §5.3.2.

Figure 5.7 shows the performance and overhead of SpTS kernel using push traversal and CSC format. The results demonstrate that the ATA framework significantly outperforms the other approaches, achieving a geometric mean speedup of 3.3× and 3.7× on VEGA-FE and R9-NANO GPUs, respectively. Due to the higher cost of active waiting on the R9-NANO GPU, ATA achieves better performance compared to self-scheduling. Furthermore, the results indicate that concurrency-aware (CA) aggregation outperforms locality-aware (LA) aggregation, as sparse applications tend to have a limited spatial locality and LA aggregation can also increase the critical execution path (see §5.3.3). In addition, the pull variant of SpTS shows a similar trend to the push execution, as detailed in Figure 5.8. However, ATA has a slightly lower geometric mean speedup of 3.0× and 3.3× on VEGA-FE and R9-NANO GPUs, respectively, compared to the push execution as the pull execution requires lower dependency management overhead (see §5.3.2).

Most importantly, ATA has better performance across the different types of application DAGs due to its hierarchical dependency management and efficient mapping of user tasks to the active wavefronts using SET scheduling. In particular, the self-scheduling approach is even worse than level-set execution for wide DAGs because the large number of user tasks



(a) Speedup and adaptive grain

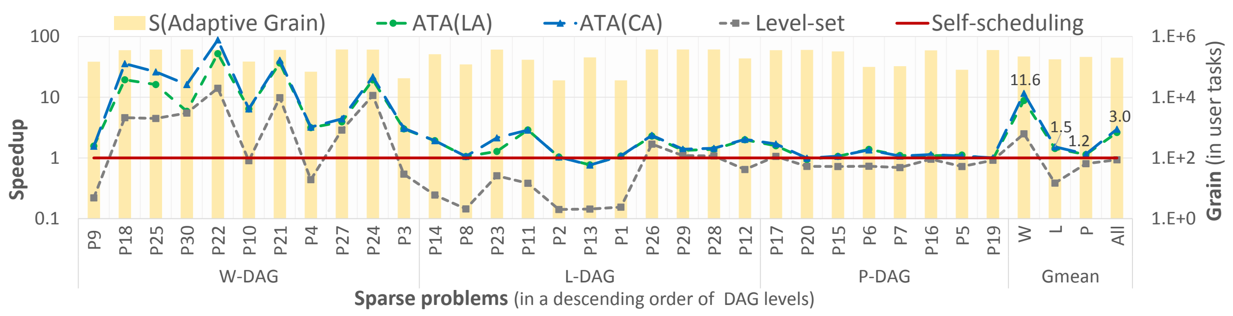


(b) Overhead relative to execution time without dependencies

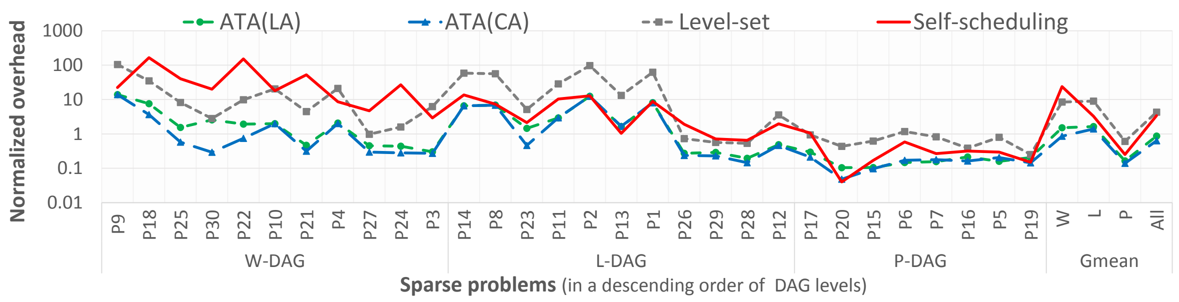
Figure 5.7: The performance and overhead of SpTS (push traversal) kernels using the different execution approaches on VEGA-FE.

at the lower DAG levels incur significant active-waiting overhead; such overhead can be higher than the computation time by more than two orders of magnitude for large-scale problems with long critical paths, as explained in Figures 5.7 and 5.8. For L-shaped DAGs, the average performance of level-set execution is significantly worse than the other methods because of the limited number of concurrent user tasks in the majority of DAG levels; hence, the overhead of global barrier synchronization becomes prohibitive, especially for problems with deeper critical paths. On the other hand, the results for L-shaped and parallel DAGs show that level-set execution achieves comparable (or better) performance to self-scheduling as the length of the critical path (i.e., number of DAG levels) decreases, due to the higher concurrency and the lower overhead of global barrier synchronization.

Figure 5.9 shows the performance and overhead of the pull execution of SpILU0 kernel using the different DAG execution methods. Since SpILU0 performs more operations than SpTS,



(a) Speedup and adaptive grain

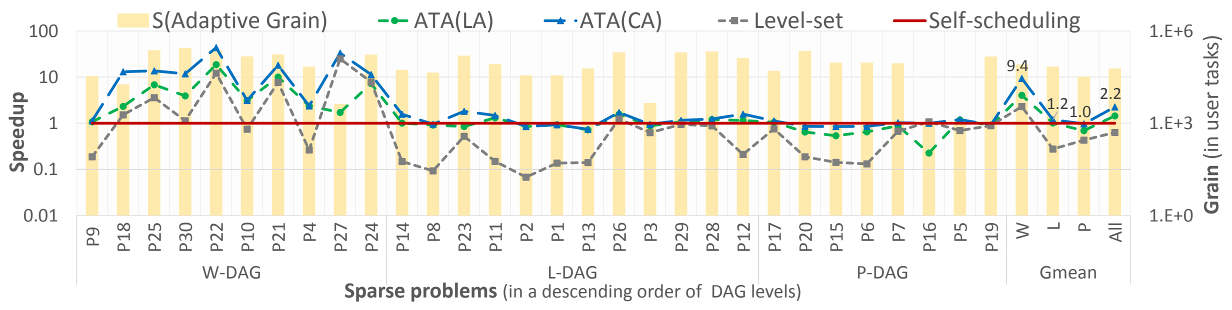


(b) Overhead relative to execution time without dependencies

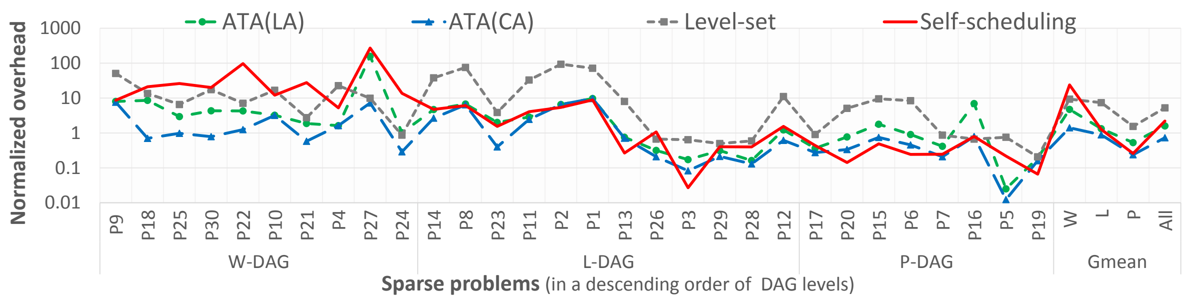
Figure 5.8: The performance and overhead of SpTS (pull traversal) kernels using the different execution approaches on VEGA-FE.

the dependency-management overhead is relatively smaller compared to the computation time. Specifically, in SpILU0, the number of operations is relative to the number of non-zero elements of each user task and also the non-zero elements of its predecessors, which results in roughly an order of magnitude smaller adaptive grain size (S) compared to SpTS. Hence, ATA achieves a geometric mean speedup of 2.2× for the SpILU0 kernel in comparison with a geometric mean speedup of 3.0×–3.7× for the different variants of SpTS.

Finally, Figure 5.10 presents the preprocessing cost required to generate ATA's hierarchical DAG from the fine-grained application DAG for each sparse problem. Since such a cost depends on the number of user tasks and data dependencies, it increases with the problem size; however, the maximum cost is approximately 0.1 second in the target benchmarks, which include sparse problems with millions of tasks (i.e., unknowns) and tens of millions of data dependencies (i.e., nonzeros). LA aggregation has a higher cost than CA aggregation



(a) Speedup and adaptive grain



(b) Overhead relative to execution time without dependencies

Figure 5.9: The performance and overhead of SpILU0 (pull traversal) using the different execution approaches on VEGA-FE.

because it typically uses a larger number of data dependencies to enforce the DAG execution order, compared to the CA policy, as explained in §5.3.3. Specifically, the geometric mean cost of generating the hierarchical DAG is 18 ms and 22 ms for the CA and LA aggregation policies, respectively.

It is important to note that once the hierarchical DAG is generated, it can be used many times during the application run. Target user applications, such as CFD and CAD applications, typically solve a nonlinear system of equations at many time points; each *nonlinear* system solution requires several iterations of a *linear* system solver, which in turn needs tens to hundreds of iterations to converge [166]. Thus, in practice, such a preprocessing cost is negligible. In addition, the preprocessing phase can be overlapped with other operations, including the system solution phase. Optimizing the preprocessing cost is outside the scope of this work and a further reduction of this cost is feasible.

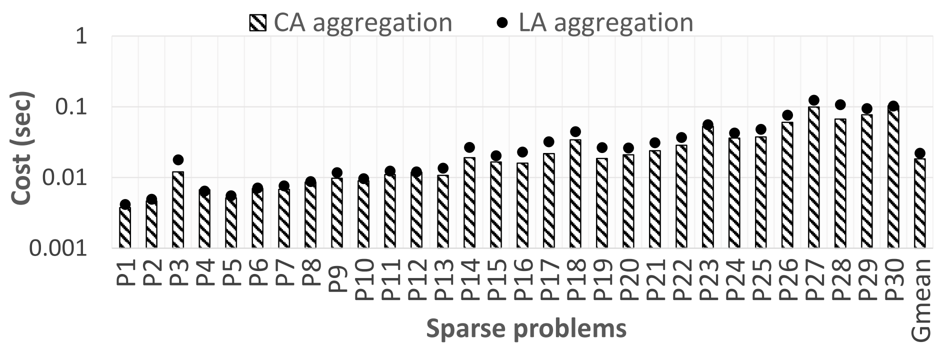


Figure 5.10: The cost of hierarchical DAG transformation using the different task aggregation policies.

5.4.3 End-to-End Application Performance

To evaluate the end-to-end application performance, we use the preconditioned conjugate gradient (PCG) method for solving sparse linear systems with a symmetric and positive-definite (SPD) matrix [166]. We implemented a PCG solver, based on Algorithm 9.1 from Saad [166], using the data-dependent kernels discussed in this work (namely, SpTS and SpILU0) and open-source SpMV and BLAS kernels from clSPARSE library [74]. Specifically, the data-dependent kernels of PCG solver perform pull traversal of application DAGs to execute the compute tasks. In the experiment, the right-hand side is a unit vector and the maximum number of iterations allowed to find a solution is 2000. The PCG solver converges when the relative residual (tolerance) is below one millionth (10^{-6}), starting from an initial guess of zero. We evaluate three versions of the PCG solver; each version uses different SpTS and SpILU0 kernels, based on ATA and prior level-set and self-scheduling approaches, and the same SpMV and BLAS kernels.

Figure 5.11 presents the execution time and number of iterations of the PCG solver for the set of SPD problems in Table 5.1. The detailed profiling indicates that the data-dependent kernels constitute the majority of execution time, ranging from 76% to 99% of the total runtime across PCG solver versions and input problems. As a result, the performance of the PCG solver shows a similar trend to the data-dependent kernels discussed in §5.4.2, where

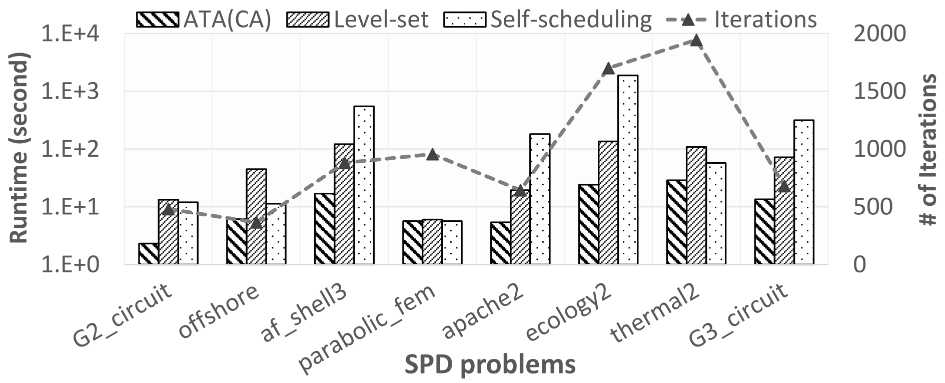


Figure 5.11: The performance of PCG solver on VEGA-FE.

ATA significantly outperforms previous methods across the different sparse problems. The performance gain depends on the characteristics of input problems, as detailed in §5.4.2. Overall, this experiment demonstrates the efficacy of the proposed framework to greatly improve the end-to-end application performance. Specifically, ATA's PCG solver achieves a geometric mean speedup of 4.4× and 8.9× compared to PCG solvers implemented using prior level-set and self-scheduling methods, respectively.

5.5 Discussion

In addition to the widely adopted level-set [11, 168, 136, 137, 118] and self-scheduling [169, 41, 119, 120, 8] techniques, which we discussed in previous sections, several approaches have been proposed to improve the performance of sparse solvers on GPUs.

Graph-coloring methods [183, 138] can increase the parallelism of sparse solvers by permuting the rows and columns of input matrices; however, such a permutation breaks the original data dependencies and the corresponding DAG execution order, which affects the accuracy of the system solution and typically increases the number of iterations needed for convergence in iterative solvers [166]. In addition, finding the color sets of a DAG is an NP-complete problem that requires significant preprocessing overhead.

Prior work [44, 12] used approximation algorithms to solve the target sparse system without data dependency management. Similar to graph-coloring methods, these approximation algorithms affect the solution accuracy and convergence rate of sparse solvers.

Various approaches [69, 158, 112, 208] exploit dense patterns in the underlying sparse problems and use dense BLAS [60] routines/kernels to improve data locality and to reduce the indirect addressing overhead; however, such techniques are limited to problems with structured sparsity patterns. Recently, Wang et al. [197] proposed sparse level tile (SLT) format, which is tailored for locality-aware execution of SpTS on Sunway architecture. Nevertheless, end users need to either re-factor existing applications to use such a specialized format or convert their sparse data before and after each call to SLT-based solvers.

5.6 Conclusion

This chapter presented the adaptive task aggregation (ATA) framework to greatly reduce the dispatch, scheduling, and dependency management overhead of irregular computations with fine-grained tasks and strong data dependencies on massively data-parallel architectures, such as GPUs. Unlike previous work, ATA is aware of the processing overhead on target architectures and it adapts the parallel execution to the dependency structure of underlying problems using (1) hierarchical dependency management at multiple levels of computational granularity, and (2) efficient sorted eager task (SET) scheduling of the application tasks based on their expected dependency-resolution time. As such, the ATA framework achieves significant performance gains across the different types of application problems. Specifically, the experiments with various sparse solver kernels demonstrated a geometric mean speedup of 2.2× to 3.7× over the existing DAG execution approaches and up to two orders-of-magnitude speedups for large-scale problems with a wide DAG and long critical path.

Chapter 6

Scalable Execution of Irregular Data-Dependent Computations

6.1 Introduction

This chapter tackles the challenges of scaling irregular applications with limited parallelism and fine-grained data dependencies on heterogeneous HPC cluster and cloud architectures. Such applications suffer from a substantial synchronization and communication overhead on distributed-memory systems, relative to the computation time, which traditionally restricted their scalability to a single homogenous architecture.

To this end, we propose an adaptive parallel execution framework for a representative irregular application, namely, SPICE simulation using the direct method [135], on a cloud-based heterogeneous system. SPICE simulation of very large scale integrated (VLSI) circuits, with millions of elements, can take days of execution time, which makes it the most critical bottleneck in the design flow. The on-demand availability of computing power on the cloud has made it possible to economically utilize a large number of hardware resources, such as multi-core CPUs and many-cores GPUs, once the problem has been algorithmically decomposed.

Parallel SPICE simulation has been studied by many researchers in the last four decades [167, 117]. While the direct method is well-known of being robust and the most accurate, it is extremely challenging for parallel execution. Most efforts to parallelize its fine-grained computations has had limited results in terms of scalability of speedup because of the inherent synchronization and communication cost, which limit their applicability to a single compute node [117, 40, 39, 75, 22]. The approach of circuit partitioning using node-tearing [152] has been extensively investigated in the early adoption phases of parallel processing for SPICE, as it can decompose the problem into several pseudo-independent partitions [35, 45]. However, increasing the number of these circuit partitions was always limited by solving the connectivity matrix. Moreover, these efforts utilized a single computation platform. Since SPICE simulation is a large problem with a diversity of components, a single platform is unlikely to satisfy the needs of all computational tasks. Therefore, integrating different architectures in a heterogeneous platform is a promising alternative to achieve scalable speedup.

This work proposes SPICE-H, an adaptive parallel execution framework for circuit simulation on heterogeneous distributed-memory systems. Specifically, we attack the problem of efficiently parallelizing the direct method for SPICE simulation on a heterogeneous HPC cloud using circuit partitioning. This problem is challenging for several reasons. First, finding the optimal workload partitioning and load balancing is nontrivial, especially for tightly-coupled circuits such as post-layout designs. Second, achieving optimal scalability is very difficult due to the parallelization and scheduling overhead and the limited amount of parallel operations in the original algorithms. Finally, assigning the partitioned computational tasks to the most suitable architecture that matches their characteristics is complicated, and requires extensive performance modeling and runtime analysis.

As such, SPICE-H generates a runtime adaptation model to optimally partition and distribute the circuit across the compute nodes based on a coarse-grained, graph-structured program representation. In addition, it uses compensation-based algorithms to greatly reduce the synchronization and communication overhead between the fine-grained computations across circuit partitions. Furthermore, SPICE-H divides the simulation process into four major computational kernels and it generates another runtime adaptation model to assign each of these kernels to the most suitable execution platform of the Amazon EC2 heterogeneous cloud. Unlike alternative iterative-based parallel techniques [102], SPICE-H does not impose any restriction on the nature of the simulated circuit and it achieves the same accuracy level as the referencer Berkeley SPICE3f5 [155]. In summary, our contributions are as follows:

- We propose the SPICE-H framework to tackle the scalability challenges of analyzing and simulating general VLSI circuits with SPICE-level accuracy. SPICE-H generates graph-structured program representations and runtime adaptation models to transform the target fine-grained, data-dependent computations for efficient parallel execution on heterogeneous distributed-memory clouds (§6.4).
- The experimental results, with diverse real-world problems, show that SPICE-H delivers up to an order-of-magnitude speedup over the optimized multi-threaded implementations of SPICE [139] with state-of-the-art KLU [53] and NICSLU [39] solver packages (§6.5).

6.2 Background

SPICE (Simulation Program with Integrated Circuit Emphasis) [135] maps the target circuit into a set of partial differential equations (PDEs) using Kirchhoff's current law (KCL) and modified nodal analysis (MNA) [152]. These equations are discretized using a numerical integration method such that the time domain is divided into a set of time points. The circuit response (waveforms) at every time point is obtained using the Newton-Raphson (NR) nonlinear solver [152].

Figure 6.1 shows the typical computational kernels in the NR solver during the transient (time domain) circuit analysis. The NR solver starts with an initial guess and iteratively solves the circuit equations ($[Y][V] = [I]$) using a direct linear solver [51] until convergence is achieved. Specifically, in each iteration, the circuit equations are linearized by evaluating

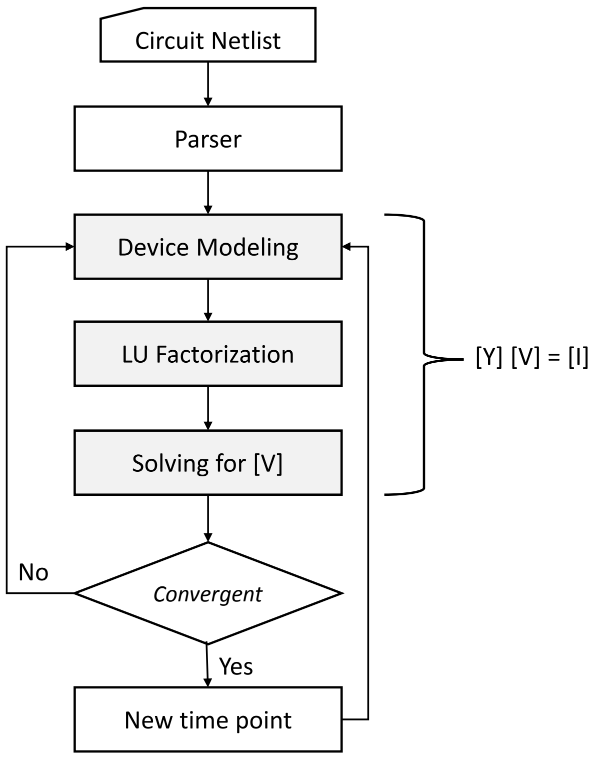


Figure 6.1: SPICE transient analysis.

the circuit device models and the resulting sparse Jacobian matrix (Y-matrix) is solved using a direct method, such as LU factorization [51]. Alternative circuit simulators use iterative methods to solve the Jacobian matrix. In modern circuit designs, the decreasing feature size introduces massive parasitic coupling that makes the convergence harder to achieve. Therefore, post-layout circuit simulations are not viable for the iterative-based circuit simulators. In addition, the direct method of SPICE is widely known to achieve a golden accuracy level.

Typically, the most time consuming kernels in SPICE transient analysis are the device model evaluation and the Jacobian matrix (Y-matrix) solution [135]. The distribution of execution time between the device model evaluation and matrix solution depends on the nature of the circuit under simulation. The device model evaluation is the dominant part of the simulation in pre-layout circuits with large number of transistors, while the matrix solution is the major execution time component in post-layout circuits with a large number of parasitic elements.

6.3 SPICE Profiling and Modeling

6.3.1 Performance Modeling Methods

Our goal is to derive analytical models that capture the complex interaction of the application (SPICE), input data, and target architectures. These models are evaluated to identify the most efficient architecture for the major computation kernels in SPICE. We adopt the analysis approach introduced in Chapter 3. For simplicity, we ignore the on-chip memory access time as the target sparse kernels typically have limited data locality.

Assuming the computations and memory accesses are perfectly overlapped [48], the execution time of a SPICE kernel on a given architecture, T_{exe}, is calculated as:

$$T_{exe} = \max\{T_{comp},\ T_{mem}\} + T_{syn} \tag{6.1}$$

where T_{comp} is the computation time, T_{mem} is the memory access time, and T_{syn} is the synchronization overhead. T_{comp} and T_{mem} are given by:

$$T_{comp} = (D(n) + \frac{W(n)}{n_p}) \times \frac{1}{\pi_0} \tag{6.2}$$

$$T_{mem} = \alpha\ D(n) + \frac{Q(n) \times L}{\beta} \tag{6.3}$$

where n_p is the number of cores, π_0 is the maximum operation throughput per core, α is the memory access latency, β is the peak memory bandwidth, and we assume that the total work, critical path, and memory transfers on the parallel architecture are bounded by $W(n)$, $D(n)$, and $Q(n)$, respectively.

The synchronization time, T_{syn}, is estimated as:

$$T_{syn} = S(n) \times s_0 \tag{6.4}$$

where $S(n)$ is the total number of synchronization points, and s_0 is the synchronization cost.

6.3.2 Hardware Architecture Model

We target heterogeneous HPC cluster and cloud architectures with compute nodes that are connected via low-latency, high-bandwidth network. This work considers Amazon EC2 cloud and uses its CGI heterogeneous instances, which consist of a dual-socket CPU host and two GPUs and are connected via 10 Gbps Ethernet with a latency of 60 μs [63]. The CPU host has two Intel Nehalem x5570 multi-core processors coupled via QPI links with speed of 25.6 GB/s. The GPU accelerator is NVIDIA Tesla M2050, which is connected to the host through a 12 GB/s PCI express link [150, 70, 143].

Table 6.1: Performance parameters of cg1.4xlarge instances.

HW Parameter	**Intel Nehalem x5570**	**Tesla Fermi M2050**
Clock (GHz)	2.93	1.15
n_p	4	448
$n_p \times \pi_0$ (GFLOPS)	46.88	515
Z (MB)	8	1.6
L (Byte)	64	128
β (GB/s)	32	148.42
α (ns)	65 (on socket) 106 (off socket)	347.8
s_0 (us)	0.4 (single socket) 0.6 (dual socket)	0.25 (local) 11.5 (global)

Table 6.1 summarizes the hardware performance parameters of the target Amazon EC2 cloud instances (cg1.4xlarge). We use vendors' data to obtain the parameters n_p, π_0, α, and β [150, 70, 143], and a micro-benchmarking approach to measure the synchronization cost s_0. As detailed in Chapter 3, there are two main synchronization mechanisms for GPUs: local synchronization and global synchronization. The local synchronization leverages CUDA/OpenCL primitives, while global synchronization is achieved using kernel termination and re-launching from the host. The GPU global memory can be used to accelerate global

synchronization by a factor of 4-8× [206]; however, the results can be incorrect without the use of memory fences to enforce memory consistency. With memory consistency, these techniques are slower than kernel termination and re-launching [65]. While CUDA dynamic parallelism may reduce the overhead of global synchronization by launching kernels on the GPU, it is not supported on the target Tesla M2050. The fast memory size, Z, is chosen to be the effective on-chip memory. On CPUs, Z is the last level cache; while on GPUs, Z is the shared (local) memory and L2 cache.

6.3.3 SPICE Performance Models

Device Model Evaluation

The evaluation of circuit device models consists of several independent tasks, which can be trivial to parallelize as they scale linearly with the number of threads. However, only one thread at a time can perform the matrix stamping, i.e., loading the evaluated device models in the Jacobian matrix (Y-matrix). While several techniques can decrease the synchronization overhead, the number of synchronization points in matrix loading is only reduced by a constant factor [167]. Based on our analysis, the following models are derived:

$$W(n) = O(n_d\, d_0), \quad D(n) = O(n_d + d_0) \tag{6.5}$$

$$Q(n) = O(n_d + \frac{nnz}{\sqrt{Z}}) \tag{6.6}$$

$$S(n) = O(n_d) \tag{6.7}$$

where n_d is the total number of circuit devices, d_0 is the average operations per device, nnz is the total number of nonzero elements in Y-matrix. For device models such as BSIM4, the value of d_0 is around 500-1000. If we exclude the matrix loading, the new values of $S(n)$, $D(n)$, and $Q(n)$ are $O(1)$, $O(d_0)$, and $O(n_d)$, respectively.

By substituting in equation 6.2, the execution time of device evaluation only is given by:

$$T_{exe} = \max\{\frac{d_0}{\pi_0} + \frac{n_d\, d_0}{n_p\, \pi_0}, \alpha\, d_0 + \frac{n_d\, L}{\beta}\} + s_0 \tag{6.8}$$

Assuming $n_d >> d_0$, equation 6.8 can be simplified as:

$$T_{exe} \approx \max\{\frac{d_0}{n_p\ \pi_0}, \frac{L}{\beta}\}\ n_d \tag{6.9}$$

Using the hardware performance parameters in Table 6.1, the above execution time is minimized by performing the device evaluation on the target GPU architecture.

By substituting in equation 6.1, the execution time of matrix loading is calculated as:

$$T_{exe} = \alpha\ n_d + \frac{nnz\ L}{\beta\ \sqrt{Z}} + n_d\ s_0 \tag{6.10}$$

Similarly, using the hardware parameters in Table 6.1, the execution time of the matrix loading (stamping) is minimized on the target CPU due to the larger cache, and smaller latency and synchronization overhead.

Jacobian Matrix Solution

The matrix solution consists of LU factorization and solving the factored matrix using forward-backward substitutions. The LU factorization is the major execution component as its complexity increases nonlinearly with the matrix size. Further, it is inherently sequential due to the large number of synchronization points, which results in a limited number of independent operations that can be performed in parallel. The left-looking factorization algorithms were shown to be efficient because of the extremely sparse nature of the Jacobian Y-matrix [53]. Based on our analysis, the following models are derived:

$$W(n) = O(k^2\ n), \quad D(n) = O(n) \tag{6.11}$$

$$Q(n) = O(\frac{k^2\ n}{\sqrt{Z}}), \quad k^2 < n \tag{6.12}$$

$$S(n) = O(n) \tag{6.13}$$

where k is the average number of nonzero elements per column, and n is the matrix dimension (i.e., the number of circuit equations).

By substituting in equation 6.1, the execution time of the matrix solution is given by:

$$T_{exe} = \max\{\frac{n}{\pi_0} + \frac{k^2\ n}{n_p\ \pi_0},\ \alpha\ n + \frac{k^2\ n\ L}{\beta\ \sqrt{Z}}\} + n\ s_0 \tag{6.14}$$

Given that $n >> k$ and using the hardware performance parameters in Table 6.1, the execution time of matrix solution is minimized on the target CPU because of the larger cache, and lower latency and synchronization overhead.

6.4 SPICE-H Approach

Section 6.3 demonstrated that the synchronization overhead across the fine-grained computations of SPICE kernels, which results from the sequential data dependencies, limits the scalability of the fine-grained parallelization approaches and restricts their applicability to a single computing node. As such, SPICE-H employs compensation-based program transformations to increase the number of independent compute operations and to limit the synchronization and communication to a constant number (O(1)) of points, which enables the use of a heterogeneous distributed-memory architecture.

Specifically, SPICE-H uses a coarse-grained parallelization approach based on circuit node-tearing (Diakoptics), which is a form of domain decomposition, first introduced by Kron [110]. It is adopted in SPICE simulation by partitioning the circuit into several subcircuits and by independently obtaining the response of each one. Then, the resulting solution is corrected to account for the effect of the circuit interface nodes (ports). The correction is performed by evaluating the interconnection matrix and solving it to get the interconnection vector, which is used with the solution of subcircuits to obtain the final circuit solution [152, 45]. Thus, the circuit simulation using SPICE-H can be divided into four major computational kernels: partition device-model evaluation (PME), partition matrix factorization (PMF), interconnection matrix evaluation (IME), and interconnection matrix factorization (IMF). PME and PMF independently obtain the solution of each subcircuit, while IME and IMF measure and account for the interconnection effect between the subcircuits.

6.4.1 Adaptive Parallel Execution on Heterogeneous Distributed-memory Systems

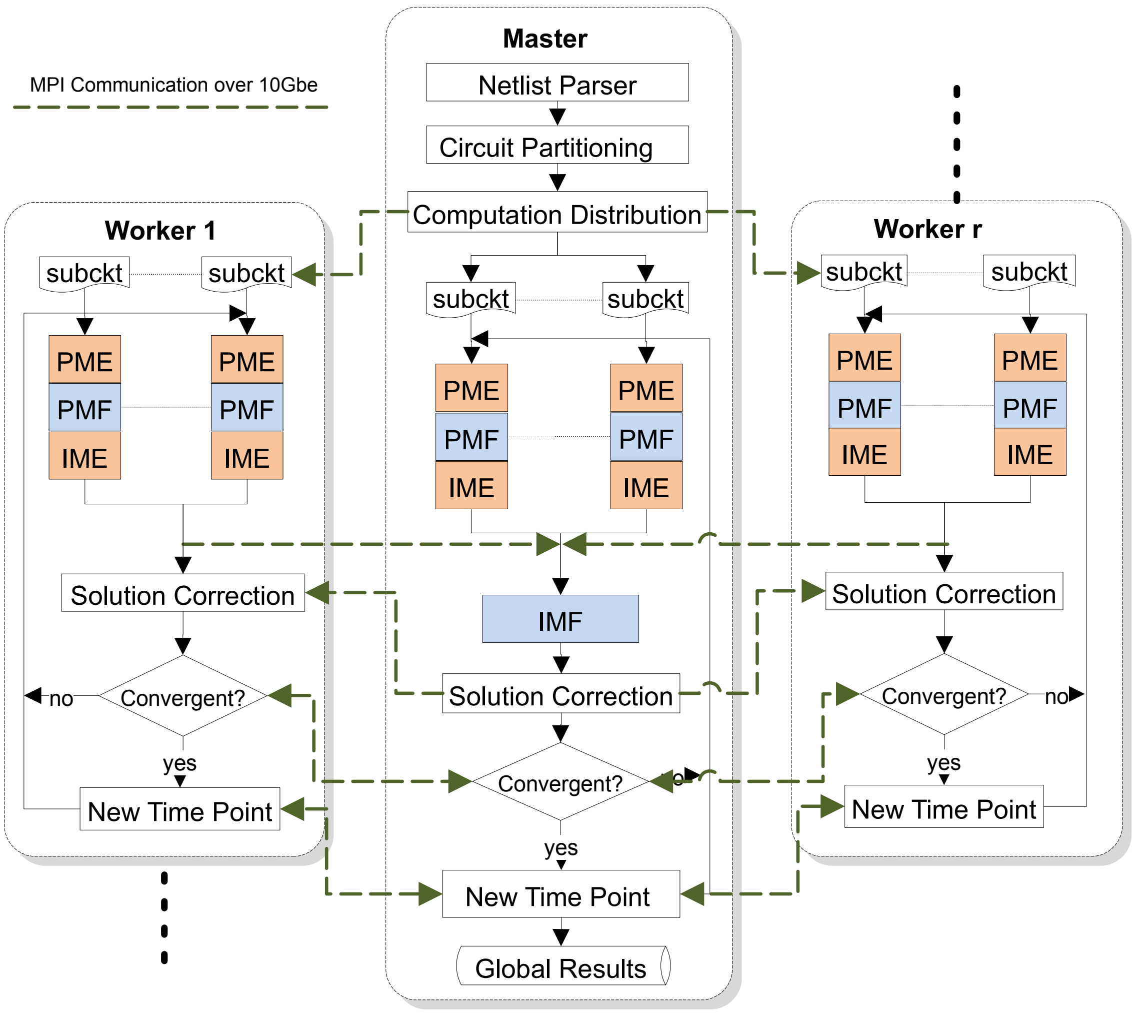


Figure 6.2: An overview of SPICE-H framework.

Figure 6.2 shows the proposed SPICE-H framework for adaptive parallel execution of circuit simulation. The simulator consists of one or more processes arranged as a master process and multiple worker processes. Each process generates several CPU threads and launches GPU kernels to work on the computational tasks assigned to it based on the runtime adaptation models. The master process is responsible for initializing the application and performing the circuit partitioning. Moreover, it uses the computation distribution algorithm to divide the subcircuits over the available processes. Each process is responsible for solving a set of subcircuits along with their contribution to the interconnection matrix, and the master process has an additional task of solving the interconnection matrix and broadcasting the solution such that each process corrects the results of its subcircuits. The master process receives the Newton-Raphson convergence results from the workers and either iterates on the same time point or advances to a new one. In addition, it receives the time step suggestions from the workers, which is calculated based on the local truncation error from the integration method. Then, the master process computes the next global time step and broadcasts it to the worker processes. Due to the latency of MPI communication over the 10Gbe links, SPICE-H uses redundant computations to avoid communication whenever the execution time of the duplicated computations is smaller than the latency penalty (which is approximately 60 μs on the target heterogeneous cloud). Finally, the master process is responsible for writing the waveforms (circuit response) to an output file.

Unlike the prior fine-grained parallelization approaches, the number of synchronization points required by SPICE-H is constant and does not grow with respect to the problem size (i.e., the number of circuit equations). Compared to the iterative-based simulation methods, SPICE-H adopts the direct method to analyze and simulate *general* VLSI circuits regardless of their different types. Specifically, it uses the convergence criteria, integration methods, and time step control of the reference Berkeley SPICE3f5 [155] to achieve the same accuracy level.

Most importantly, we derive a runtime adaptation model to optimally partition the input circuit across the compute nodes based on its inherent communication and execution costs. Another model is derived to assign each of the simulation kernels to the most suitable execution platform of the Amazon EC2 heterogeneous cloud. As indicated in §6.4.3, the GPU is the most suitable architecture for device model and interconnection matrix evaluation, while CPUs are better in loading and factorizing the Jacobian Y-submatrices and the interconnection matrix. Therefore, SPICE-H assigns PME and IME kernels to GPUs, and PMF and IMF kernels to CPUs. However, if the target GPUs are fully committed, SPICE-H distributes the workload by using the CPUs to execute the waiting PME and IME subtasks.

6.4.2 Adaptive Circuit Partitioning and Distribution

The SPICE-H framework constructs a graph-structured program representation to reason about the execution bottlenecks on distributed-memory HPC systems and to guide the target compensation-based algorithm for circuit simulation. This program representation is aware of the input data as well as the execution platform; thus, it is used to drive an adaptive model-based circuit partitioning that maximizes the overall performance by reducing the communication and synchronization overhead and the idle/waiting time.

In SPICE simulation with circuit node tearing, each of the torn nodes accumulates a three-fold overhead. First, every cut increases the order of the interconnection matrix, which is a tightly-coupled, semi-dense matrix. Second, two forward-backward substitutions of the LU-factored submatrices are required to form each row of the interconnection matrix [152]. Third, the communication cost is proportional to the number of interface/port nodes (J). Therefore, minimizing J is essential to achieve scalable performance, which is achieved using the following model-based runtime adaptation.

The sequential execution time of SPICE transient analysis for any given circuit is a function of the execution platform capabilities and the circuit characteristics, such as the number of unknowns, the number of circuit elements, and the type of circuit elements. It can be modeled as follows:

$$T_s = n^{\gamma}, \quad \gamma > 1 \tag{6.15}$$

where n is the number of circuit equations (unknowns), and γ is a coefficient calculated empirically. The value of γ depends on the circuit type and the execution platform.

The parallel execution time of SPICE transient analysis with circuit node tearing depends on the interface nodes, J, and the workload balance. The parameter J models both the communication cost and the additional computation overhead, as detailed above. Since the execution time depends on the largest circuit partition, the workload balance sets an upper bound on the performance scalability. Assuming a circuit partitioning algorithm that generates balanced partitions, the execution time of the parallel SPICE simulation with circuit node tearing on a distributed-memory architecture, with sufficient resources to run all the subcircuits in parallel, can be modeled as follows:

$$T_p = (\frac{n}{m})^{\gamma} + m^{\sigma}, \quad \gamma > 1, \quad \sigma > 1 \tag{6.16}$$

where m is the number of subcircuits, and σ is a coefficient calculated empirically. The value of σ depends on the circuit type, the partitioning algorithm, and the execution platform.

To find the optimal number of partitions, m^*, such that the overall execution time is minimized, we calculate the derivative of equation 6.16 and solve the resulting equation, as follows:

$$m^* = e^{\frac{ln\ \epsilon}{\sigma+\gamma}}, \quad \epsilon = \frac{\sigma}{\gamma}\ n^{\gamma} \tag{6.17}$$

Figure 6.3 shows the trends of the optimal number of partitions (m^*) for $n = 100,000$ circuit unknowns with the typical values of γ and σ obtained from the benchmark circuits. As γ increases, the computational cost of circuit simulation increases and SPICE-H can generate more circuit partitions to reduce the overall execution time. In contrast, when σ increases, the communication and synchronization overhead increases which leads to a smaller number of circuit partitions to make efficient use of the compute resources. Equation 6.17 shows that m^* can only be estimated at runtime, after n is given and γ and σ are calculated. Since m^* is calculated once for a given circuit and execution platform, while the circuit is typically simulated hundreds of times under different operating conditions and inputs, the overhead of estimating m^* is negligible with respect to the simulation time.

The adaptive circuit partitioning is performed in two stages. First, SPICE-H represents the input circuit as a hypergraph, as shown in Figure 6.4, where vertices are circuit element and hyperedges are circuit nodes. Each vertex and hyperedge has an associated weight to model its computational cost. The highly-connected areas in the hypergraph representation are clustered and collapsed into macro vertices, and the optimal number of partitions m^* is estimated as detailed above (see equation 6.17).

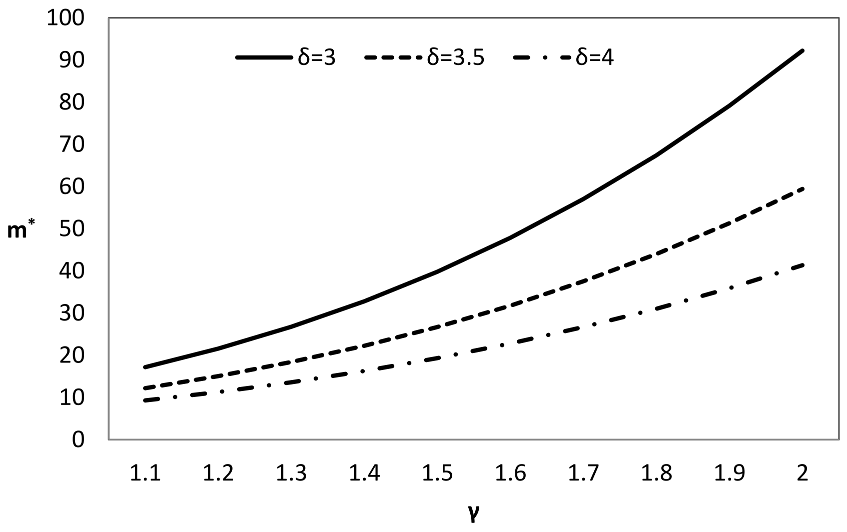


Figure 6.3: The trend of the optimal number of partitions, m^*, for $n = 100,000$ circuit equations/unknowns.

Second, SPICE-H uses the multilevel hypergraph partitioning algorithms [99] to partition the target hypergraph into m^* partitions, such that each partition has almost the same computational cost with a minimum number of cuts. Then, a custom module is used to analyze the output of the hypergraph partitioning stage and modify it to ensure that each partition has a valid reference circuit node and there are no floating circuit nodes. Finally, the resulting partitions are mapped to subcircuits.

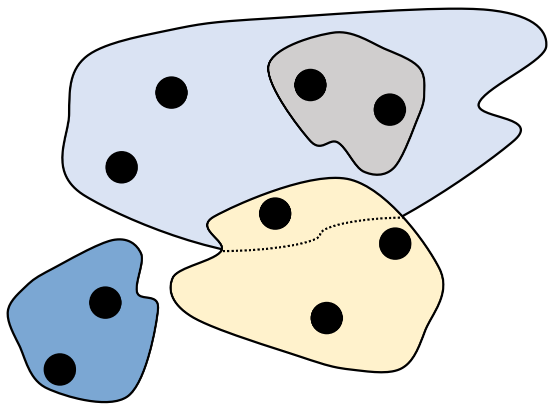


Figure 6.4: SPICE-H generates a hypergraph program representation for adaptive parallel execution of the target application.

Computation Distribution

Once the input circuit is partitioned and mapped to valid subcircuits, SPICE-H distributes these subcircuits over the worker processes. We developed a simple algorithm for dividing the overall computational cost evenly over the available processes, after excluding the cost associated with solving the interconnection matrix. This workload distribution produces balanced groups (clusters) of subcircuits, and then assigns each group to a worker process. The computation distribution problem can be stated as follows: given r processes and m subcircuits, where a subcircuit S_i has a computational cost C_i; assign the subcircuits to the compute processes such that the maximum load is minimized. The proposed mechanism for computation distribution sorts the subcircuits in a decreasing order according to the cost function (C_i), which is chosen to represent the number of interface nodes per subcircuit, and then assigns the subcircuit with the largest cost to the least-loaded worker process.

6.4.3 Kernel-Architecture Matching

The proposed SPICE-H framework generates affinity prediction models to match its diverse computational kernels, namely, PME, PMF, IME, and IMF kernels, to the different CPUs and GPUs of the target heterogeneous HPC system. At runtime, SPICE-H adapts the ongoing parallel execution to the dynamic system activities by modifying the affinity of the waiting compute subtasks when deemed necessary, as detailed in §6.4.1. For example, if the GPUs are fully committed, SPICE-H uses the CPUs to execute the waiting compute subtasks by changing their original affinity.

Partition Model Evaluation (PME) and Matrix Factorization (PMF)

The PME kernel is responsible for evaluating the device models and loading the Jacobian Y-submatrices of subcircuits, while the PMF kernel factorizes the Y-submatrices. Since the subcircuits are completely independent, the PME and PMF kernels consist of m independent subtasks (where m is the number of subcircuits) that can execute on a distributed-memory heterogeneous architecture without synchronization. Assuming balanced subcircuits and reusing the analysis in §6.3.3, SPICE-H partitioning of the input circuit and transformation of the circuit simulation modify the original performance models as follows:

- For each PME and PMF subtask, the synchronization overhead (the main sequential bottleneck) is reduced by a factor of m.
- For each PME subtask, the work and depth is reduced by a factor of m, while the memory accesses are reduced by a factor of m^2.
- For each PMF subtask, the work and memory accesses are reduced by a factor of m^3, while the depth is reduced by a factor of m.

Similar to the original circuit, the execution time of subcircuits is minimized by performing the partition device model evaluation (PME kernel) on the target GPU and the partition matrix loading and factorization (PMF kernel) on the CPU.

Interconnection Matrix Evaluation (IME)

The interconnection matrix consists of J equations, which correspond to the circuit interface (port) nodes. To form each equation, two forward-backward substitutions (solutions) of the factorized LU submatrices are required. Since all the forward-backward substitutions are independent, IME consists of $2 \times J$ independent LU-solve that can execute on a distributed-memory architecture without synchronization. Assuming balanced subcircuits and using the analysis methodology in §6.3, the execution time of each forward-backward substitution (IME subtask) is given by:

$$T_{exe} = \max\{\frac{n}{m\ \pi_0} + \frac{nnz}{m^2\ n_p\ \pi_0}, \frac{\alpha\ n}{m} + \frac{nnz\ L}{\beta\ m^2\sqrt{Z}}\} + \frac{n\ s_0}{m} \tag{6.18}$$

Using the hardware performance parameters in Table 6.1, the execution time of each IME subtask is minimized on the target CPU, due to the larger cache and the smaller latency and synchronization overhead. As detailed above, there are $2 \times J$ independent IME subtasks and typically $J >> m$, such that each submatrix is solved for tens to hundreds of times. Therefore, the overall execution time of the IME subtasks that belong to a single submatrix (subcircuit) is minimized on the GPU by using a single wavefront (warp) to perform each individual subtask and benefit from the implicit intra-wavefront synchronization (with zero overhead).

Specifically, the implementation of the IME kernel employs several transformations to match the GPU architecture. Each factorized submatrix is divided into lower (L) and upper (U) halves. All of the submatrices representing the different subcircuits are put in sequence into a continuous array. As for the right-hand side (RHS) stimulus vectors, at the interface nodes, are stored into one long vector. Other arrays for indexing the data are also prepared. Since each of these submatrices are used for tens to hundreds of solutions and to minimize the number of read operations for their arrays, all the solutions that belong to the same submatrix are performed in parallel before moving to another submatrix. A second dimension of parallelization uses a wavefront (warp) to perform the fine-grained operations in a single solution (IME subtask).

Interconnection Matrix Factorization (IMF)

Unlike the other computational kernels, IMF kernel is dominated by a single tightly-coupled task, which is LU factorization of the interconnection matrix. Therefore, SPICE-H executes it on a single compute node, due to the synchronization and communication overhead on the target distributed-memory HPC system. Using the analysis methodology in §6.3, the execution time of the IMF kernel is given by:

$$T_{exe} = \max\{\frac{J}{\pi_0} + \frac{k_J^2\ J}{n_p \pi_0}, \alpha\ J + \frac{nnz_J\ J\ L}{\beta\ \sqrt{Z}}\} + J\ s_0 \tag{6.19}$$

where k_J is the average number of nonzeros per column, and nnz_J is the total number of nonzeros in the interconnection matrix. Similar to the Y-matrix and using the hardware performance parameters in Table 6.1, the execution time of IMF kernel is minimized on the target CPU, due to the larger cache and the smaller latency and synchronization overhead.

In the IMF kernel, the interconnection matrix is solved using the right-looking factorization algorithms [51]. The Left-looking algorithm was shown to be inefficient because of the semi-dense nature of the matrix. Specifically, this interconnection matrix is solved using a dynamic allocation of threads depending on the required computational operations. Both the L and U half matrices are divided into slices, with one thread at the head of each slice, together with child threads working on the fine-grained computations within this slice. These numbers are determined during the symbolic analysis phase of SPICE simulation (see §6.2).

6.5 Evaluation

Since no single programming model fits the target architecture, we implemented the host code in C and used a hybrid MPI/OpenMP programming model to combine the inter-node and intra-node parallelism. Both CUDA and OpenCL have been equally used to program the GPU accelerators. Here, we present the CUDA results.

The experiments use two different types of circuits: analog mixed-signal and digital transistor-dominant circuits. Table 6.2 presents the detailed characteristics of the test cases. CKT1 is a large-scale, memory-like circuit with a structured design, which is the best case for the SPICE-H framework. CKT2 is a tightly-coupled analog circuit with a large number of parasitic elements, which is the worst case for the SPICE-H approach. CKT3, CKT4, and CKT5 are medium size circuits with various characteristics.

We compare SPICE-H with three different simulators: single-threaded SPICE [139], multi-threaded SPICE with KLU solver [53], and multi-threaded SPICE with NICSLU solver [39]. Table 6.3 details the characteristics of these simulators. Unlike the previous approaches, SPICE-H works at two different parallelism levels: a coarse-grained (circuit partitioning) level, and a fine-grained (subcircuit operations) level.

Table 6.2: Characteristics of test circuits.

Each BSIM4 device adds 1 resistor for gate (Rgatemod=1), and 5 resistors for substrate (Rbodymod=1), in addition to the original elements listed below.

Circuit	**Type**	**#unknowns**	**#linear elements**	**#devices**
CKT1	Transistor-dominant	1,410,464	2,880	222,080
CKT2	Post-layout analog mixed-signal	85,188	303,771	11,740
CKT3	Pre-layout analog mixed-signal	35,741	185	7,804
CKT4	Transistor-dominant	45,570	7,505	10,112
CKT5	Transistor-dominant	67,044	52,846	10,112

Table 6.3: The characteristics of target simulators.

ID	**Description**	**Devices**	**Matrix Loading**	**Matrix LU/Solve**	**Interconnection**
SPICE	Single-thread SPICE	Sequential	Sequential	Sparse1.3	n/a
SPICE-K	Multi-threaded SPICE with KLU	OpenMP	OpenMP	Sequential KLU	n/a
SPICE-N	Multi-threaded SPICE with NICSLU	OpenMP	OpenMP	Parallel NICSLU	n/a
SPICE-H	Proposed multi-node heterogeneous SPICE	MPI/ CUDA	MPI/ OpenMP	MPI/ OpenMP	MPI/OpenMP/ CUDA

Table 6.4: Performance of SPICE-H.

SU1, SU2 and SU3 are speedup vs. SPICE, SPICE-K and SPICE-N.

(The resources used by SPICE-K and SPICE-N are 8 cores.)

Circuit	**#subcircuits**	**Resources for SPICE-H**	**SU1**	**SU2**	**SU3**
CKT1	32	4 × cg1.4xlarge instances (32 cores, 8 GPUs)	n/a	12.08×	12.29×
CKT2	24	3 × cg1.4xlarge instances (24 cores, 6 GPUs)	75.65×	7.76×	2.73×
CKT3	8	1 × cg1.4xlarge instances (8 cores, 2 GPUs)	12.67×	4.00×	3.87×
CKT4	16	2 × cg1.4xlarge instances (16 cores, 4 GPUs)	19.69×	7.38×	6.77×
CKT5	16	2 × cg1.4xlarge instances (16 cores, 4 GPUs)	13.77×	4.92×	4.69×

The runtime results are obtained by performing transient analysis for [0, 100] ns with a timestep of 1ps. The circuit device models are based on BSIM4 with full parasitic, including gate and substrate resistances. The MIT StarCluster framework [159] is used to automate the configuration and management of the test platform, which is a virtual cluster consists of several cg1.4xlarge instances of the Amazon EC2 cloud. The virtual cluster uses a single placement group to ensure that all instances are connected over low-latency, high-bandwidth interconnection links.

Table 6.4 summarizes the speedup of our SPICE-H approach in comparison with the prior simulators. In CKT1, the best case, the proposed SPICE-H simulator achieves 12.1× and 12.3× speedup over SPICE-K and SPICE-N, respectively, while SPICE could not perform the simulation. In CKT2, the worst case, SPICE-H outperforms SPICE, SPICE-K, and SPICE-N by a factor of 75.7×, 7.7×, and 2.7×, respectively. In medium circuits with less than 100,000 unknowns (circuit equations), our approach achieves on average 15.4×, 5.4×, and 5.1× speedup over SPICE, SPICE-K and SPICE-N, respectively, which indicates that the overhead of the SPICE-H simulation framework is insignificant.

6.6 Conclusion

This chapter demonstrated the development of a scalable framework for SPICE circuit simulation using the direct method on a high-performance heterogeneous cloud. The proposed SPICE-H framework constructs a coarse-grained, graph-structured program representation to reason about the performance bottlenecks on the target platform. As a result, SPICE-H uses compensation-based program transformations and different levels of parallelism to maximize the performance: (1) a coarse-grained parallelism level using circuit partitioning, and (2) a fine-grained parallelism level to solve each partition and the interconnection matrix. Unlike prior work, SPICE-H generates runtime adaptation models for optimal workload partitioning and mapping on heterogeneous HPC systems. The combination of these techniques dissolved the execution bottlenecks that have limited the scalability of parallel circuit simulation.

An implementation of SPICE-H on the Amazon EC2 heterogeneous cloud proved to be two orders-of-magnitudes faster than the basic sequential SPICE, and around one order-of-magnitude faster than the optimized multi-threaded implementation using KLU and NICSLU matrix solvers. The test circuits used in the evaluation are real-world, industrial-grade circuits. They covered a large simulation spectrum, ranging from structured circuits, with a large number of transistors, to post-layout, analog-mixed signal circuits.

Chapter 7

Interoperation on Heterogeneous HPC Systems

7.1 Introduction

In the last decade, parallel computing architectures that span a wide range of execution models have emerged to meet the increasing demand for high-performance applications driven by the multitude of large-scale datasets. Examples of these architectures are multi-core CPUs, many-core GPUs, and Intel Many Integrated Cores (MICs). To attain scalable performance efficiently — relative to power, energy, and cost — the high-performance computing community expects future (exascale and beyond) HPC systems to consist of heterogeneous compute nodes, with different types of hardware accelerators, connected over a high-speed, low-latency network infrastructure. In addition to the expected many-core accelerators, namely GPUs, that populate the top end of the Green500 and TOP500 Lists [130], other candidate accelerators for these heterogeneous HPC systems include low-power embedded multiprocessors, custom hardware accelerators (potentially emulated on FPGAs), and even FPGAs themselves [24].

Such heterogeneous computing systems require hybrid programming models to exploit their potential performance and energy efficiency, and dealing with this interoperation between different compute devices and parallel programming models is a tedious and error-prone task. In addition, the abundance of parallel architectures has complicated the design and development of high-performance applications even more. Scientists face several design choices and have to decide which architecture, programming model, algorithm, and implementation technique are the most suitable for their applications. For example, NVIDIA GPUs can be programmed via CUDA, OpenCL, OpenMP, OpenACC, PTX, and several other research programming models. Each option has a different learning curve and a potential performance, but typically the best performance requires a low-level implementation approach and significant architecture expertise, which is in short supply.

Furthermore, parallel architectures change faster than parallel programming models and software, and scientists should not have to spend their time re-learning and rewriting code for the new architectures. Thus, to effectively use future exascale computing systems, end users need appropriate tools to make efficient use of the available compute resources, both within and across compute nodes, and to do so without needing to have extensive architectural expertise and with minimal development time.

This chapter presents MetaMorph, a middleware that resides between user applications and hardware platforms to enable seamless execution and interoperation across heterogeneous parallel architectures. MetaMorph is designed as a library framework to (automatically) extract as much computational capability as possible from exascale computing systems with three core design principles: *abstraction, interoperability and adaptivity.*

- *MetaMorph abstracts current and future hardware accelerators behind a single interface.* It does so without complicated installations or extensive application refactoring that existing solutions require. This, in turn, supports the development and upgrade of accelerated applications by end users with minimal development effort and time. MetaMorph provides not only high-level abstraction, but also high performance, comparable to hand-coded and manually-tuned accelerated kernels (§7.2.2).

- *MetaMorph promotes interoperability across different accelerators and with existing software.* MetaMorph's communication interface supports data exchange between *different* hardware accelerators not only in a single compute node, but also across multiple nodes. Moreover, unlike many existing solutions that use proprietary data types, MetaMorph's APIs are designed, using standard data types, to be as close to pure C language constructs as possible and to expose internal device contexts to promote interoperability with hardware vendors' libraries, domain-specific libraries, and existing code. This interoperability is crucial in the adaptation to any new library framework (§7.2.1).
- *MetaMorph is designed to be adaptive to the capabilities of the execution platform and runtime environment.* It is built to be modular and allow end users to only select the components and features relevant to the application and hardware in hand. Furthermore, it provides simultaneous access to all compute devices and accelerators present in a system, promoting the development of an overarching runtime scheduler to map the accelerated computations to the best hardware platform(s) available and to reduce the execution time. Additionally, transparent interfaces to both intra- and inter-node data transfers provide an opportunity for intelligent partitioning and pipelining to increase overlap of computation and communication and to hide the data transfer latency (§7.2.3).

To demonstrate the efficacy of the proposed MetaMorph library framework, we present a case study with the *structured grids* design pattern, which is heavily used in computational fluid dynamics (CFD). Specifically, we evaluate MetaMorph with benchmarks and a larger application, namely MiniGhost, which is a representative CFD application for solving partial differential equations with the finite difference method. The experiments demonstrate that MetaMorph significantly reduces the development effort and time for heterogeneous systems without performance penalty and it can be used to seamlessly utilize *all* the available hardware accelerators across multiple compute nodes, which include multi-core CPUs, Intel MICs, AMD GPUs, and NVIDIA GPUs. In addition, we show MetaMorph's interoperability with hardware vendors' libraries and third-party libraries such as clBLAS [9], Intel MKL [193] and MAGMA libraries [58] (§7.4).

7.2 Design Philosophy

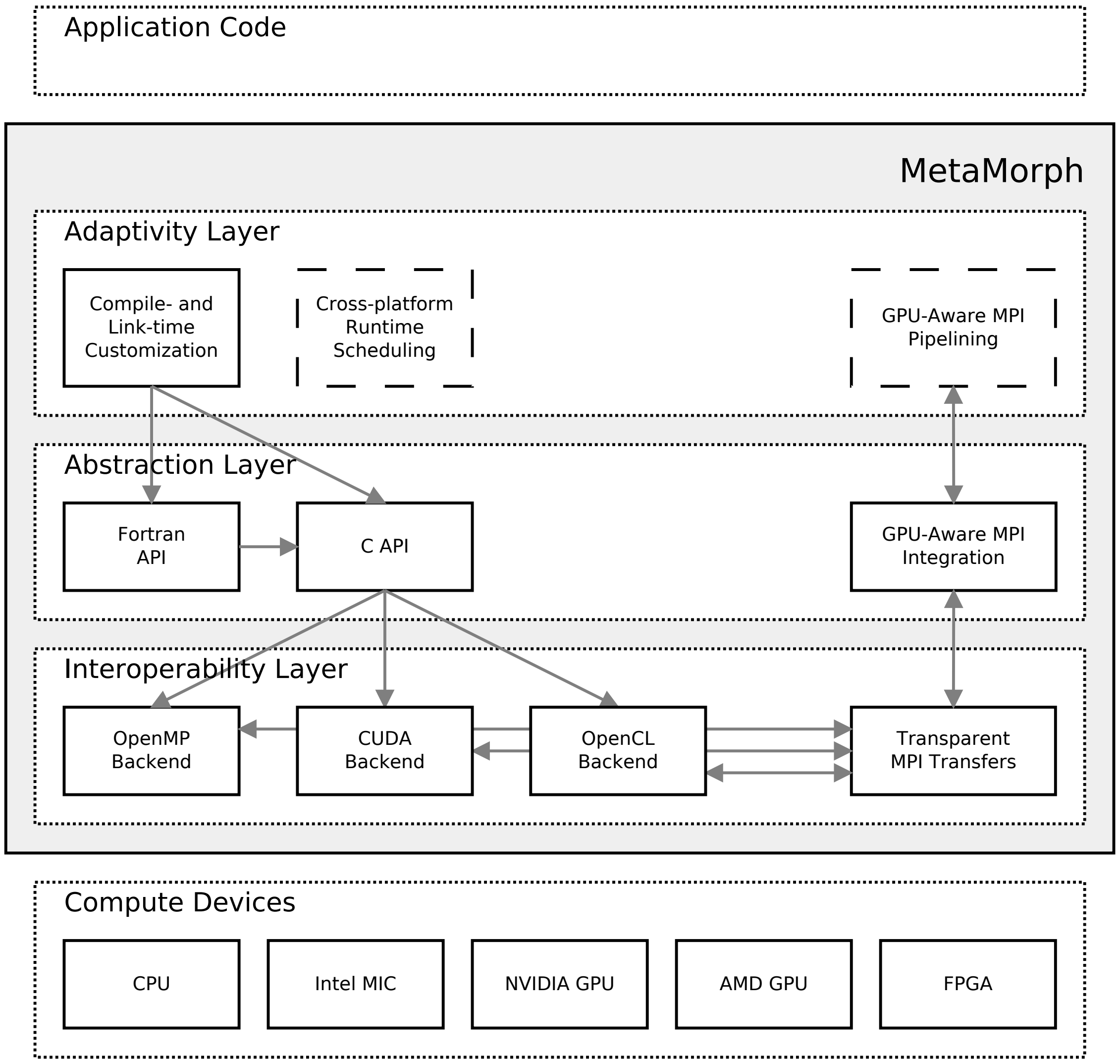


Figure 7.1: MetaMorph uses a modular, layered design to hide the complexity of executing on and interoperating across a range of compute platforms.[1]

MetaMorph is designed to effectively utilize HPC systems that consist of multiple heterogeneous compute nodes with different hardware accelerators. Figure 7.1 shows the proposed MetaMorph library framework, which acts as *a middleware* between the application code

[1]Dashed lines and boxes indicate components that are in development or not yet fully integrated into the prototype.

and compute devices, such as multi-core CPUs, many-core GPUs, Intel MICs, and FPGAs. It hides the complexity of developing code for and executing on these heterogeneous platforms by acting as a unified "meta-platform." The application developer needs only to call MetaMorph's computation and communication APIs, and these operations are transparently mapped to the proper compute devices. MetaMorph uses a modular layered design, where each layer supports one of its core design principles and each module can be used relatively independently.

7.2.1 Interoperability Layer

When designing complex scientific applications for HPC clusters, two types of interoperability are critical: (1) interoperability between compute nodes that may have different hardware capabilities and (2) interoperability with existing code and external computation and communication libraries. However, existing solutions only provide one of the two types. Many frameworks provide multi-platform portability via complex data types or custom compute languages (often utilizing obtuse template meta-programming), thus satisfying the first type of interoperability, but falling short of the second, as the entire application must be ported to the framework's types and/or language. Conversely, it is relatively easy to add a platform-specific library to an existing application that already uses that platform (e.g., adding cuBLAS [142] calls to a CUDA application), but these are *not* portable to other platforms, thus satisfying the second type of interoperability, but not the first.

MetaMorph, on the other hand, is designed from the ground up to support both types of interoperability. Figure 7.2 provides a sketch of this interoperatbility, and a concrete example is detailed in §7.4.

Interoperability across Different Accelerators

MetaMorph satisfies this type of interoperability with its unified API design and transparent communication interface. When a user application calls a MetaMorph API, the call is transparently mapped to a back-end accelerator supported by the underlying platform and

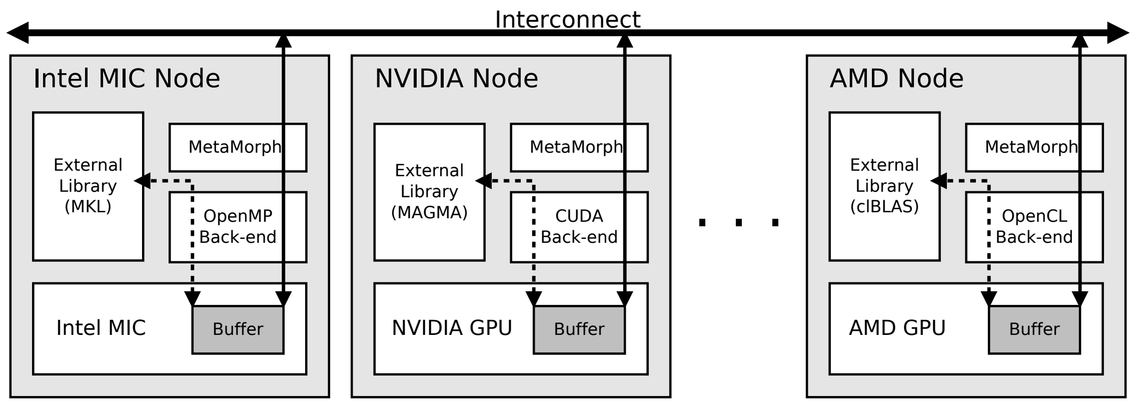


Figure 7.2: MetaMorph provides interoperability both with external libraries in a node (dashed lines) and across nodes with varying hardware (solid lines).

the MetaMorph library running on the node, which dramatically simplifies the programming required to coordinate multiple processes running on *different* hardware platforms. Further, MetaMorph provides a communication interface to transfer back-end-resident data, agnostic of the underlying execution platform, allowing for seamless interoperation across multiple compute nodes with a range of different hardware configurations.

Interoperability with Existing Software

MetaMorph satisfies this type of interoperability by the careful design of its internal representation of data buffers and platform-specific back-ends. MetaMorph data buffers are specified in the top-level APIs as simple C `void` pointers, with an enumerator identifying the primitive data type that the back-end implementation should use. The back-end that a given void pointer actually resides on is inferred at runtime from the state of the `run_mode` variable. To support incremental porting of existing applications developed for a specific platform, MetaMorph's internal context is exposed via additional API functions, so that back-end state can be shared directly with the host application (i.e., an application can share a context with MetaMorph via `meta_get_state` and `meta_set_state`). Thus, application developers can use MetaMorph, while they incrementally port and validate their applications, which significantly eases the transition process.

7.2.2 Abstraction Layer

Achieving (and more importantly, improving) the application performance in the real and changing world has become a function of portability; non-portable application codes stop gaining performance if (or more accurately, *when*) their target platform reaches end of life. Application codes can be manually ported to new generations of hardware architectures, but both functional and performance portability are often difficult to achieve without extensive expertise. Therefore, a need exists to future-proof programming solutions that obviate the demand for users to manually port performance-critical codes to new hardware devices. The choice to provide this capability as a library framework was natural, given the proliferation of libraries in use in many scientific applications and the vast range of domain- or platform-specific libraries for common operations, such as dense and sparse linear algebra.

MetaMorph provides programmability, functional portability, and performance portability by abstracting the software back-ends (currently, OpenMP, CUDA, and OpenCL) behind a single interface. As such, it bridges the performance-programmability gap by decomposing the problem space into two parts: high performance and high-level abstraction. First, MetaMorph achieves high performance by providing low-level implementations of common operations, based on the best-known solutions for a given compute platform. Moreover, the software back-ends are instantiated and individually tuned for the different heterogeneous and parallel computing platforms (currently, multi-core CPUs, Intel MICs, AMD GPUs, and NVIDIA GPUs).

Second, MetaMorph achieves high-level abstraction by hiding all device- and platform-specific details behind the unified interface, which enables the end user to write the application once and to run it on any supported device. The portability criterion is further satisfied by providing an infrastructure for adding software back-ends for future compute devices — without end-user intervention or modifying the application. This provides the small population of early adopters and architecture experts with a framework that enables them to dramatically extend the impact of their expertise to the wider community by expanding

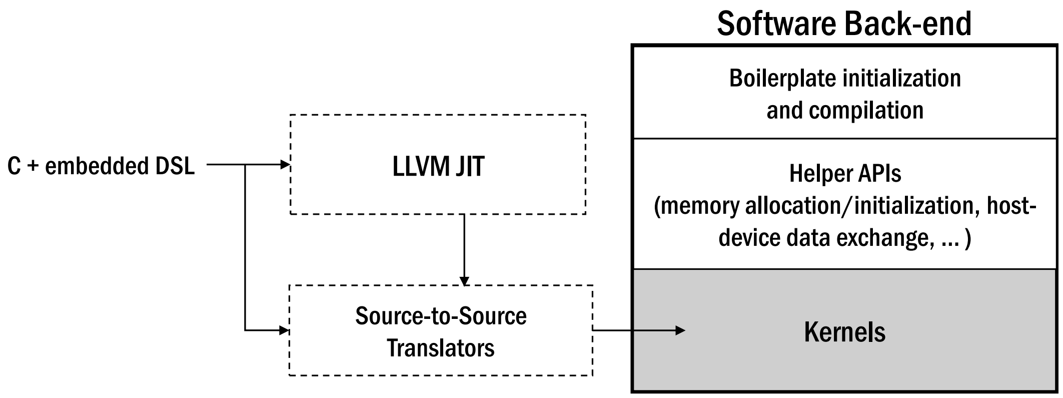

Software Back-end
Boilerplate initialization and compilation
Helper APIs (memory allocation/initialization, host-device data exchange, ...)
Kernels
C + embedded DSL
LLVM JIT
Source-to-Source Translators

Figure 7.3: MetaMorph accelerates the development of new operations and kernels.

the library with new design patterns. So, rather than writing a kernel once for a single application, these experts can write that same kernel within the MetaMorph framework, provide it to the community, and allow it to be used across many applications.

Further, MetaMorph accelerates the development of new operations and computation/communication patterns, as shown in Figure 7.3. It provides a compilation infrastructure and helper APIs that handle the boilerplate initialization and compilation and simplify data exchange between the host and accelerators, such that MetaMorph developers can focus on developing the new kernels. In addition, we used our source-to-source translators [126, 170] to largely automate the generation of MetaMorph kernels, and in the future, we aim to release a family of such source-to-source translators and leverage the LLVM Just-In-Time compiler to simplify the generation of the application host code and the expansion of the MetaMorph library with new design patterns.

Finally, existing kernels (e.g., CUDA kernels) can be included in MetaMorph without refactoring by adding their implementation directly to the interoperability layer (e.g., CUDA back-end) and their C/Fortran interface in the abstraction layer. However, advanced features, like the seamless execution on different accelerators within a node and across nodes, will work only if these kernels are implemented in all the different back-ends. Contributing a kernel for even a single back-end is valuable as it provides the architecture-expert community a baseline, from which to implement and integrate kernels for the remaining back-ends.

7.2.3 Adaptivity Layer

Software written for the heterogeneous and coming exascale era must be highly adaptive, as the expanding range of compute platforms and the increasing performance demands continually reshape the computing landscape. Accordingly, any library framework intended to provide high performance through such changes must itself be able to quickly respond to new opportunities to improve the performance. Therefore, MetaMorph has taken a modular layered approach that makes upgrading performance-critical components simple, without affecting user applications. First, the adaptivity layer provides compile-time and link-time customization and optimization for performance-critical components. All the performance-critical code for a given compute platform is encapsulated in a shared library object and separated from the core MetaMorph library and all other back-ends. This library-of-libraries construction promotes the optimization of back-ends to target specific compute devices, by making them disjoint code objects that can be tuned in isolation for new devices and then shared among the community. Further, the back-ends can be customized for different types of compute devices. For example, the OpenMP back-end can be customized for multi-core CPUs and Intel MICs, and the OpenCL back-end can be customized for CPUs, AMD GPUs, NVIDIA GPUs, Intel MICs, and FPGAs.

Second, the adaptivity layer provides runtime services to accelerate computations and data transfers. The accelerator-aware communication infrastructure is inspired by MPI-ACC [6], which allows automatic partitioning and pipelining of device-resident data buffers to hide data-transfer latencies. Another runtime service that is built into the MetaMorph design philosophy is cross-platform runtime scheduling, which intelligently maps the computations kernels to the best hardware platform(s) available, based on their relative performance, to reduce execution time. Our cross-platform scheduler is based on CoreTSAR [171], an adaptive runtime system. In summary, we highlight that our library framework facilitates the upgrading and development of such runtime services without significant modification to user applications, thus allowing end users to enjoy a “free ride” to better performance.

7.3 Prototype Implementation

We realize MetaMorph[2] as a layered library of libraries. Each tier implements one of the core principles of abstraction, interoperability, and adaptivity. The top-level user APIs and platform-specific back-ends exist as separate library objects, with interfaces designated in shared header files. Primarily, this encapsulation supports custom tuning of back-ends to a specific device or class of devices, as detailed in §7.2. In addition, This design allows the back-ends to be separately used, distributed, compiled, or even completely rewritten, without interference with the other components.

7.3.1 Programming Models

The core API, library infrastructure, and communication interface are written in standard C language for portability and performance. Individual accelerator back-ends are generated in C with OpenMP and optional SIMD extensions (for CPUs and MICs), CUDA C/C++ (for NVIDIA GPUs), and C++ with OpenCL (for GPUs, APUs, and other devices). In addition, a wrapper around the top-level API is written in polymorphic Fortran 2003 to simplify interoperability with Fortran applications prevalent in some fields of scientific computing.

7.3.2 Top-Level User API

The top-level API, shown in Figure 7.4, improves the programmability of user applications by abstracting the back-ends, which provide accelerated kernels for each platform. Specifically, it implements the offload/accelerator computation model[3], in which data is explicitly allocated, copied, and manipulated via kernels within the MetaMorph context. In addition, the front-end interface supports seamless execution of user applications on different accelerators. It intercepts calls to the MetaMorph communication and computation kernels and

[2]The prototype implementation can be downloaded from https://github.com/vtsynergy/MetaMorph

[3]For API cohesiveness, the OpenMP back-end mimics the offload model. However, redundant data transfers can be eliminated, under the user's control, via the *USE_UNIFIED_MEMORY* option.

```
// Memory/Context Management
meta_alloc(void **ptr, size_t size);
meta_free(void *ptr);
meta_copy_h2d(void *dst, void *src, size_t size...);
meta_copy_d2h(void *dst, void *src, size_t size...);
meta_copy_d2d(void *dst, void *src, size_t size...);
meta_set_acc(int acc, meta_mode mode);
meta_get_acc(int *acc, meta_mode *mode);
meta_flush(); //finish any outstanding work
// share meta_context with with existing software
meta_get_state(meta_platform *plat, meta_device *dev, meta_context *context,
    meta_command_queue *queue);
meta_set_state(meta_platform plat, meta_device dev, meta_context context, meta_command_queue
     queue);
// Communication Interface
meta_comm_init(int *argc, char *** argv);
meta_comm_finalize();
meta_packed_send(int dst, void *packed_buf, size_t len, meta_type_id type...);
meta_packed_recv(int src, void *packed_buf, size_t len, meta_type_id type...);
meta_pack_send(int dst, meta_face *face, void *buf, void *packed_buf, meta_type_id type...);
meta_recv_unpack(int src, meta_face *face, void *buf, void *packed_buf, meta_type_id type...
    );
//data marshaling
meta_pack_face(void *packed_buf, void *buf, meta_face *face, meta_type type...);
meta_unpack_face(void *packed_buf, void *buf, meta_face *face, meta_type type...);
meta_transpose_face(void *ind, void *outd, dim2 *size, meta_type type...);
meta_face *make_slab(meta_slab_pos position, void *buf, dim3 *size, int thickness...);
// Timers
meta_timers_init();
meta_timers_finalize();
// Compute Kernels
meta_kernel_name(...);
...
```

Figure 7.4: Overview of the main user API exposed by MetaMorph.

transparently maps them to a back-end accelerator supported by the underlying platform. The only times that a user application needs to explicitly manage platforms are as follows:

- at compile time, when the library is advised what back-ends might be available (via conditional compilation with `-D WITH_`*`BACK-END`* definitions), and
- at run time, when the execution mode is set to one of the compiled-in back-ends (via the `METAMORPH_MODE` environment variable or a call to `meta_set_acc()`).

Memory Management

The top-level user API provides memory management functions for allocating/freeing a MetaMorph buffer and for transferring data to/from the host. MetaMorph implements its own implicit buffer types for a set of primitive data types — currently single-precision and double-precision floating points, signed and unsigned integers, and unsigned 64-bit integers — using a type enumerator and void pointer(s). Thus, individual API wrappers can dynamically map the provided void pointer to the correct type for the back-end implementation by inferring the backend-native type at runtime from the global `run_mode` variable. In addition, these API wrappers take standard C types for scalars and `size_t[N]` vectors for multi-dimensional (N-D) problem size variables, which are transmuted to appropriate types when the back-end implementation performs the actual kernel launch.

Context Management

A number of functions are exposed to exert high-level control on MetaMorph, for example, getting/setting the current execution back-end, forcing outstanding asynchronous work to complete, and sharing MetaMorph context with the existing software. To ensure compatibility and to minimize performance overhead, when only a subset of capabilities is needed, the top-level API code uses conditional compilation, such that the library users only pay the memory and performance overhead for the back-ends and options that are needed by "opting-in" at compile time.

Communication Interface

MetaMorph has been designed from the start for heterogeneous HPC clusters, hence ensuring convenient data exchange between its processes is critical.

Domain-decomposition. In many scientific application domains, including CFD, when the problem domain is sufficiently large and cannot fit on a single compute device or node, the domain is decomposed into multiple smaller sub-domains that can be computed relatively independently. However, once decomposed, each sub-domain needs access to a current copy of data from its logical neighbors. This results in a compute-then-communicate iterative loop, where neighboring sub-domains residing in separate memory spaces must synchronize their boundary data in each iteration, i.e., exchange a face or slab of the sub-domain with neighbors, as shown in Figure 7.5.

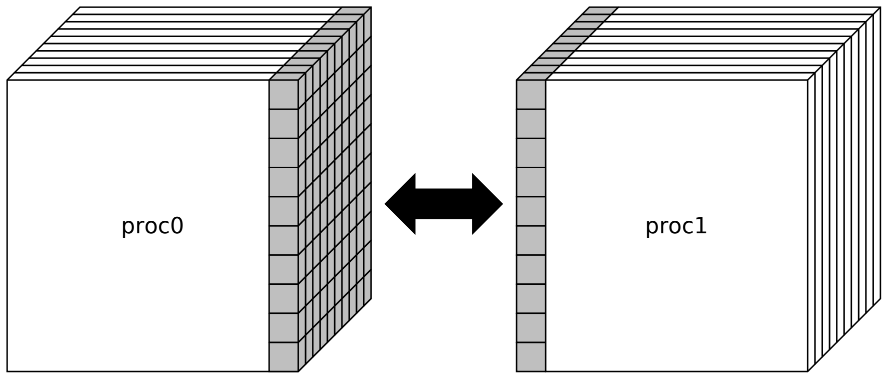
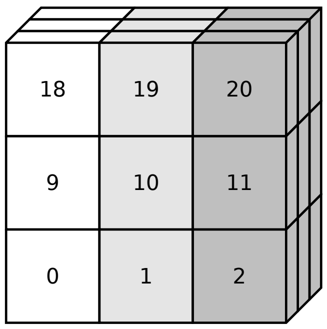


Figure 7.5: Neighboring processes must exchange boundary data every iteration.

Face Specification. As noted above, the boundary (ghost) element exchange is an important communication pattern in many scientific domains. When multi-dimensional grids are stored as standard C arrays, only two faces are stored contiguously, while the other faces have their elements scattered at well-defined stride offsets. Therefore, a concise representation is needed for defining the set of memory addresses that make up a face, so that they can all be appropriately read and exchanged. Rather than using a pointer list, we re-purpose the *gslice* data structure from `libstdc++`, which is designed specifically for recording such structured offset information about a multi-dimensional array. The data structure represents the offset computation as a tree of height N (number of dimensions), in which each successive level from the root is a finer stride through memory. That is, the leaf nodes represent the unit-stride dimension, their parents represent the dim_0-stride dimension, grandparents the $dim_0 * dim_1$-stride dimension, and so on. Figure 7.6 shows an example of this data structure, which requires $O(D)$ memory space (where D is the dimension of the data grid).

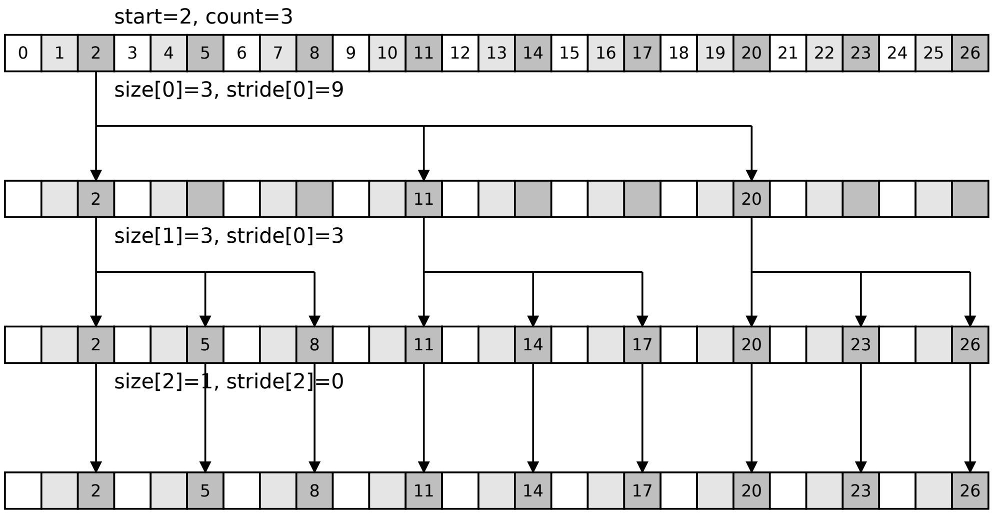


(a) A row-major 3D region with unit-dimensional indices

```
count = 3
start = 2

size[3] = {3,3,1}
stride[3] = {9,3,0}
```

(b) Specification of the right-most 2D face with compressed indexing structure



(c) Hierarchical computation of strided layout of face indices

Figure 7.6: Face description data structure and hierarchical index computation.

Data Marshaling. On-device packing and unpacking of a multi-dimensional dense grid can significantly reduce the data transfer and synchronization overhead. So, instead of using host-side data marshaling, we provide back-end variants of parallel gather/scatter operations to support the exchange of portions of a back-end-resident data buffer. In these kernels, the threads compute data indices into the unpacked buffer in parallel from the face specification, then perform a direct copy between the packed and full buffers.

Communication Meta-Operations. We expose a family of four communication operations that support exchanging data across MetaMorph processes and back-ends, in both blocking and non-blocking modes. We provide simple `packed_send` and `packed_recv` APIs, which transparently move data between the back-ends via a host-side staging buffer. In addition, two more operations are provided: `pack_send` and `recv_unpack`. These operations allow the user to specify face exchanges at a higher level of abstraction to concisely describe the exchange being performed. Currently, the user only provides the target process, the back-end-resident, multi-dimensional buffer, and the packed buffer, along with a valid face specification (as described above). Moreover, MetaMorph's communication interface has preliminary GPU-Direct support for exchanging buffers directly between two MetaMorph processes with the CUDA back-end and a GPU-aware MPI, such as MVAPICH2.

Asynchronous Communication. To overlap the computation with communication, MPI exchanges are frequently performed in an asynchronous mode. As such, all data exchange functions can operate in either blocking or non-blocking mode (via callbacks and helper functions). In practice, each high-level exchange operation consists of two to four asynchronous steps. For example, a `pack_send` operation on a non-GPU Direct back-end must 1) run the pack kernel asynchronously, 2) perform an asynchronous device-to-host copy of the packed buffer, whenever the pack kernel finishes, 3) send the host buffer using MPI_ISend , and 4) free the temporary host buffer.

Therefore, an infrastructure is provided to coordinate such asynchronous pipelines, which is built from three main components: an MPI request queue, helper functions that trigger after pending requests complete, and callback functions that are triggered by the back-ends after kernels/transfers finish. When an asynchronous MPI operation is invoked, its corresponding request is registered on a queue alongside the type of meta-operation being performed, and the necessary data and helper function required to finish any remaining work. At this time, the pending requests are only checked for completion, when a `meta_flush` is called. However, there is an opportunity to check for completion more frequently. Each time an asynchronous device kernel or transfer is invoked, the necessary data and callback function can be specified such that the back-end runtime triggers the remaining work upon completion of the accelerated device operations.

Companion Features

The final software component of the front-end API is a set of companion features that provide optional capability, which may not be needed by all users. Currently, MetaMorph provides the following features:

- Timing infrastructure that performs transparent timing of kernels and data transfers across the different software back-ends.
- Fortran compatibility that exposes both polymorphic Fortran 2003 and `ISO_C_BINDINGS`-compatible versions of the user-level API.

7.3.3 Accelerator Back-Ends

As a consequence of the dissimilarity of platform-specific programming models, the software back-ends are less uniformly constructed. However, ultimately each is responsible for providing a standard C interface to the accelerated kernels. The back-ends are segregated from one another in order to allow separate compilation and encapsulation of platform-specific nuances. Consequently, if a given back-end requires special-purpose libraries or tools to build that are not present on the target machine, it can be easily excluded from a given build of the library as a whole without loss of function in the remaining back-ends.

OpenMP Back-End. The OpenMP back-end provides standard C variants of all API functions and should be considered the default back-end, as it provides functionally correct results on any CPU, regardless of whether the compiler respects OpenMP pragmas. For some kernels, additional compile-time options are provided to further accelerate code, such as the option to use SIMD (vector) intrinsics.

CUDA Back-End. The CUDA back-end includes both the kernel functions and the host-side wrappers responsible for executing the kernel — and when necessary, auto-generating a CUDA grid/block configuration from the provided problem size. It uses the CUDA C execution configuration syntax and mixed device/host source files. When a kernel supports multiple data types, simple templates are used in order to minimize code duplication.

OpenCL Back-End. The OpenCL back-end is similar to the CUDA back-end, with a few exceptions. The kernels and host code are stored in separate files, as is common in OpenCL development. The OpenCL host code includes functions for automatically performing the OpenCL initialization boilerplate. This includes selection of OpenCL platform and device, construction of a command queue and context for executing on the device, just-in-time compilation of kernel code, and management of the resulting program and kernel objects.

7.4 Evaluation

This section presents a detailed case study of the proposed MetaMorph framework with the *structured grids* design pattern, which is widely used in computational fluid dynamics (CFD). The study demonstrates MetaMorph's capabilities with benchmarks and a representative CFD application. It shows that a program written once using MetaMorph's abstraction layer can seamlessly utilize a wide range of hardware accelerators across multiple compute nodes. The experiments evaluate MetaMorph on an experimental heterogeneous cluster with multi-core CPUs, Intel MICs, NVIDIA GPUs, and AMD GPUs, and perform scalability analysis on a large-scale CPU-GPU cluster. In addition, the study shows MetaMorph's interoperability with platform-specific libraries, such as clBLAS [9], Intel MKL [193], and MAGMA libraries [58].

7.4.1 Applications

3D Dot-Product Benchmark

We designed a benchmark that simulates the exchange of boundary or "ghost" regions in a structured grids computation, followed by a global dot-product on a 3D grid. This benchmark represent a heavily-used pattern in the iterative solvers for partial differential equations (PDEs), e.g., conjugate gradient (CG), biconjugate gradient stabilized (Bi-CGSTAB), and generalized minimum residual (GMRES).

Algorithm 10 describes our test benchmark that models a 3D structured grid computation on a domain of size Nx×Ny×Nz. After initialization, the global domain is decomposed into multiple regions (sub-domains) of size nx×ny×nz, which are logically connected in a torus along the X dimension, and each local sub-domain is assigned to a compute process. Each MPI process exchanges boundary elements with nearest neighbors, and performs a 3D dot-product on its local sub-domain. The final output is computed by combining the partial results from all processes.

Algorithm 10 3D Dot-Product Benchmark

```
Input: $Nx \times Ny \times Nz$                                    ▷ global domain size
Input: $N$                                                         ▷ number of sub-domains
Input: $Iters$                                                     ▷ number of iterations
Output: $result$                                                   ▷ global 3D dot-product result
 1: Allocate and initialize the global domain
 2: Perform domain-decomposition along the $X$ dimension
 3: Allocate and Initialize local domains of size $(nx + 1) \times ny \times nz$
 4: for $i \in$ 0:Iters-1 do
 5:     Pack ghost cells into $send\_buffer$
 6:     Send $send\_buffer$ to $proc + 1$ process
 7:     Receive ghost cells from $proc - 1$ process into $recv\_buffer$
 8:     Unpack ghost cells from $recv\_buffer$ into local domains
 9:     Compute 3D dot-product on the local domains
10:     Perform global reduction to compute $result$
11: end for
12: return $result$
```

MiniGhost

MiniGhost [19, 20] is a representative (proxy) application for CTH [85], a multi-material shock hydrodynamics code developed at Sandia Lab to model the hydrodynamic flow and dynamic deformation of solid materials. MiniGhost solves the partial differential equations of multiple variables, which represent a material state such as energy, mass, and momentum, using the finite difference method. It implements a difference stencil (e.g., 3D seven-point stencil) and explicit time-stepping scheme on a homogenous 3D domain (grid). MiniGhost supports two communication modes: a bulk synchronous parallel with message aggregation (BSPMA) mode and a single variable, aggregated face data (SVAF) mode. In BSPMA, the boundary data for all variables are exchanged in aggregated messages, while in SVAF the boundary data for each variable is sent/received in a dedicated message. When the number of variables is one, the two communication modes are equivalent.

Algorithm 11 illustrates the main computations and communication patterns in MiniGhost with the SVAF communication mode. After initialization, the global domain is decomposed along the X, Y, and Z dimensions into multiple sub-domains, and each sub-domain is mapped to a process. In each time step, the processes exchange boundary elements (2D faces) with neighbors that share a face in the X, Y, and Z dimensions. Next, the processes apply a finite difference stencil on their sub-domains. Finally, the value of the global domain is computed using a global summation.

7.4.2 Programmability and Productivity

Figure 7.7 shows the MetaMorph compute APIs that were used to accelerate the target workloads, in addition to the core APIs (see Figure 7.4). We implemented a MetaMorph version of the 3D dot-product and MiniGhost applications. To show MetaMorph's interoperability with external libraries, we created a variant of the 3D dot-product benchmark using MetaMorph and the platform-specific BLAS libraries, namely, clBLAS, Intel MKL, and MAGMA. Since the platform-specific BLAS libraries do not provide data marshaling primitives nor accelerator-aware communication interface, we use MetaMorph to do so, and only perform the dot-product operation with BLAS libraries.

To evaluate the programmability of MetaMorph, the study uses the number of effective code lines changed or added to accelerate the baseline MPI version as the target metric. We are

Algorithm 11 MiniGhost with SVAF communication mode

```
Input: $Nx \times Ny \times Nz$                                  ▷ global domain size
Input: $N$                                                       ▷ number of sub-domains
Input: $Nvar$                                                    ▷ number of variables
Input: $TimeSteps$                                               ▷ number of time steps
Input: $Stencil$                                                 ▷ stencil type (3D7P, 3D27P,...)
Output: $GridSum[Nvar]$                                          ▷ global domain value
 1: $b$                        ▷ number of neighbors per dimension (2 for 3D7P stencil)
 2: Allocate and initialize the global domain
 3: Perform domain-decomposition along the $X$, $Y$ and $Z$ dimensions
 4: Allocate and initialize local domains of size $(nx + b) \times (ny + b) \times (nz + b)$
 5: for $i \in$ 0:TimeSteps-1 do
 6:     for $j \in$ 0:Nvar-1 do
 7:         Pack boundary data into send_buffer
 8:         Send send_buffer to neighbors
 9:         Receive boundary data from neighbors into recv_buffer
10:         Unpack boundary data from recv_buffer into local domain
11:         Apply boundary conditions on the local domain
12:         Compute finite-difference stencils on the local domain
13:         Perform global reduction to compute $GridSum[j]$
14:     end for
15: end for
16: return $GridSum$
```

```
// Compute Kernels
meta_stencil(void *ind, void *outd, dim3 *size, dim3 *start, dim3 *end, meta_type type ...);

meta_dotProd(void *ind1, void *ind2, dim3 *size, dim3 *start, dim3 *end, void *result,
    meta_type type ...);

meta_reduce(void *ind, dim3 *size, dim3 *start, dim3 *end, void *result, meta_type type...);
```

Figure 7.7: MetaMorph's compute APIs that are used to accelerate the target applications.

interested in the source code lines that perform the core functionality. So, we do not consider any code lines used for profiling, timing, debugging or optional features. The applications use a single data type (double), although applications accelerated with MetaMorph can use five different data types without code modifications. Table 7.1 shows the lines of code changed or added to accelerate the sequential applications using MetaMorph, MetaMorph with platform-specific BLAS libraries, and directive-based programming models such as OpenMP. Due to its abstraction layer, MetaMorph has competitive programmability and productivity with directive-based programming models; however, unlike these programming models, it provides simultaneous access to several heterogeneous accelerators (with different characteristics and execution models) without code modifications. Moreover, through the interoperability layer, MetaMorph enables the user to utilize the existing platform-specific libraries.

Table 7.1: The number of effective code lines changed/added to accelerate the baseline MPI implementation.

Application	**OpenMP**	**MetaMorph**	**MetaMorph+BLAS**
3D dot-product	10	19	40
MiniGhost	23	38	NA

7.4.3 Experimental Results

Due to the lack of a large-scale heterogeneous cluster with different accelerators, covering all the currently supported back-ends in MetaMorph, the study considers an experimental heterogeneous cluster with multi-core CPUs, Intel MICs, NVIDIA GPUs, and AMD GPUs.

Table 7.2: The experimental cluster's configurations.

Machine Name	**CPU(s)**	**Accelerator(s)**	**OS**	**MPI**	**Compiler(s)**
ht20	Intel Xeon E5-2697v2 (2x)	NVIDIA Tesla K20Xm (2x)	Debian Jessie	MPICH 3.1.4	gcc 4.8.2 nvcc 6.0.1
dna1	Intel Core i5-2400	AMD Radeon 7970	Debian Wheezy	MPICH 3.1.4	gcc 4.7.2
mic	Intel Xeon E5-2697v2 (2x)	Intel MIC SC7120P (2x)	CentOS Linux 6	MPICH 3.1.4	icc 13.1.1

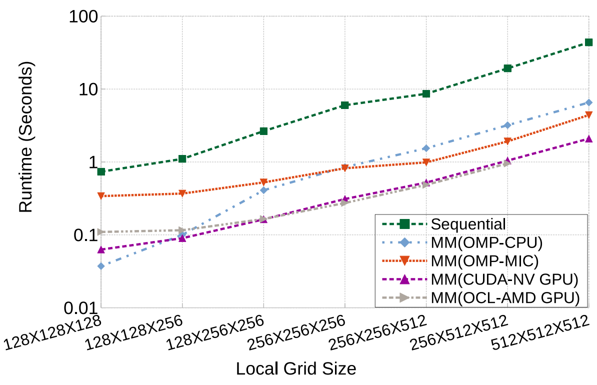
Table 7.3: The CPU-GPU cluster's configurations.

CPU(s)	**Accelerator(s)**	**OS**	**MPI (Interconnect)**	**Compiler(s)**
Intel Xeon E5645	NVIDIA Tesla C2050 (2x)	CentOS Linux 6	OpenMPI 1.6.4 (QDR InfiniBand)	gcc 4.5, icc 13.1, nvcc 5

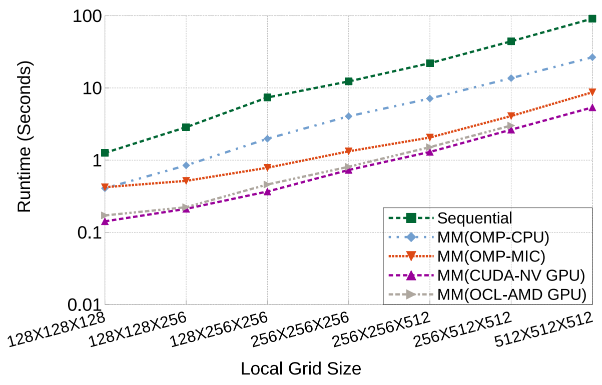
In addition, it perform scalability analysis on HokieSpeed, which is a large-scale CPU-GPU cluster at Virginia Tech. Details of the experimental nodes and the CPU-GPU cluster are provided in Table 7.2 and Table 7.3, respectively.

In the experiments, MiniGhost is configured to apply a 3D 7-point stencil on a global grid and to use an explicit time-stepping scheme with 100 time steps. The reported performance does not include sequential overheads, such as data initialization and boundary conditions. While MetaMorph supports several data types, the experiments use double-precision floating point only for brevity.

Figures 7.8, 7.9, and 7.10 show the performance of the 3D dot-product and MiniGhost applications on the experimental heterogeneous cluster. In the experiments, we launch four processes, each running on one of the supported back-end accelerators: multi-core CPU (Intel Xeon E5-2697), Intel MIC (Intel Xeon Phi SC7120P), NVIDIA GPU (K20Xm), and AMD GPU (AMD Radeon 7970). We use weak scaling with local grid sizes that are typically used in CFD applications. However, AMD Radeon 7970 could not execute the problem size of



(a) 3D dot-product benchmark



(b) MiniGhost

Figure 7.8: The performance of target applications with the different MetaMorph (MM) back-ends on the experimental cluster.

512x512x512, due to its limited global memory. Since the test compute resources are only interconnected with a high-traffic 1Gb ethernet, shared with approximately 30 other compute and memory nodes, resulting in too much network noise for meaningful intercommunication performance characterization, the reported performance does not include the inter-node data transmission time.

Figure 7.8 shows the execution time of each process of the 3D dot-product and MiniGhost applications running on one of the supported back-ends in comparison with the sequential reference implementation running on Intel Xeon E5-2697. The results show that MetaMorph achieves up to 21x and 17x speedup over the reference implementation in 3D dot-product and MiniGhost, respectively. Since structured grids applications are characterized by regular memory access pattern and low computational intensity, their performance is limited by the memory system; hence, they are suitable for many-core accelerators (GPUs and Intel MICs) with large memory bandwidth. However, the problem size must be large enough such that the kernel launch overhead and the additional data transfers are effectively amortized.

Figure 7.9 shows the runtime distribution of MiniGhost with 256x256x512 local problem size on the different MetaMorph back-ends. The main computation and communication kernels

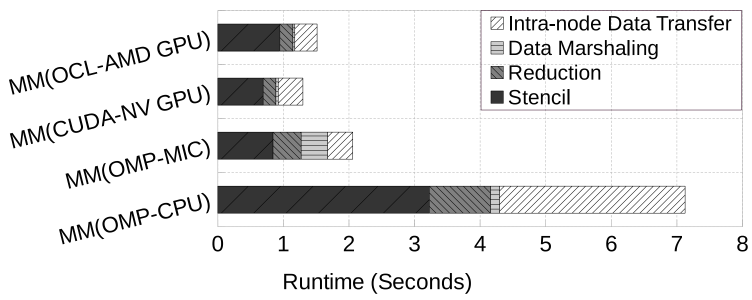


Figure 7.9: The runtime distribution of MiniGhost using a 256X256X512 local grid with the different MetaMorph (MM) back-ends on the experimental cluster.

of MiniGhost are stencil, reduction sum, data marshaling, and intra-node data transfer. On many-core accelerators, the intra-node data transfer includes host-to-device, device-to-host, and on-device data movement, while on CPUs it includes on-device data movement. MiniGhost has many on-device data transfers, as its finite-difference solver uses temporary work buffers to hold the intermediate results, while solving the partial differential equations of the material state variables. On multi-core CPUs, due to their limited memory bandwidth, the stencil kernel and the intra-node data transfers consume the majority of execution time. Many-core devices, with their large memory bandwidth, accelerate all kernels. However, data marshaling suffers from performance degradation on Intel MIC. The profiling data shows that data marshaling kernels do not utilize the vector units on Intel MIC efficiently, due to the non-unit stride memory access pattern and complex control flow. In addition, the hardware prefetcher in Intel MIC is less powerful than mainstream multi-core CPUs, when the memory access stream is non-contiguous (scattered) [64].

Figure 7.10 shows the performance of the 3D dot-product benchmark, when accelerated using MetaMorph with platform-specific BLAS libraries. The results demonstrate that the MetaMorph variant (Figure 7.8a) has comparable performance to the MetaMorph with BLAS libraries version on multi-core CPU, AMD GPU, and NVIDIA GPU, although MetaMorph's 3D dot product is more flexible than the simple contiguous dot product available in BLAS libraries, as it allows the user to perform dot product on an arbitrary sub-region of the 3D grid; this is very useful for CFD applications, where neighbor ghost cells data is often stored

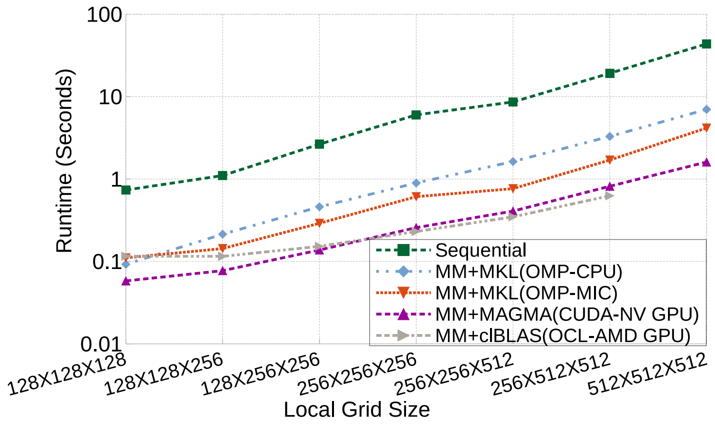
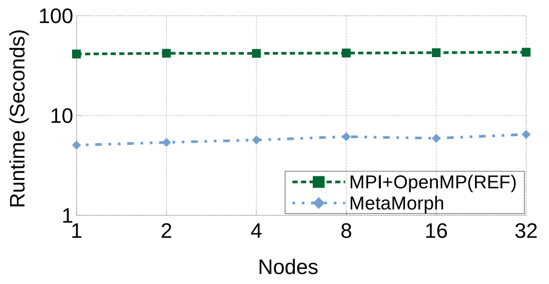


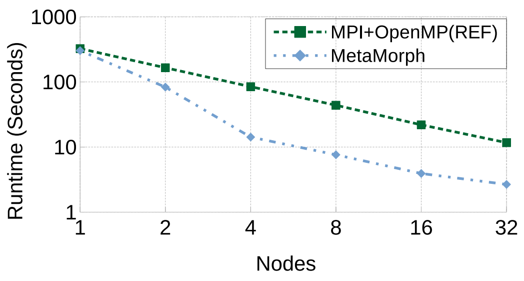
Figure 7.10: The performance of the 3D dot-product benchmark on the different MetaMorph (MM) backends with platform-specific BLAS libraries on the experimental cluster.

contiguously with local data to simplify stencil operations. However, MetaMorph with MKL outperforms MetaMorph only on Intel MIC for small problem sizes (less than 256x256x256). Our hypothesis is that Intel MKL adapts to the different inputs and problem sizes, which is a feature currently in development in MetaMorph.

Figure 7.11 shows the scalability analysis of MiniGhost on the large-scale CPU-GPU cluster. The weak scaling experiments use a problem size of 512x512x512 per node, and the strong scaling experiments use a global grid of size 1024x1024x1024. In comparison with the reference MPI+OpenMP implementation, MetaMorph achieves 7-8x speedup in the weak scaling problem. The main reason is that MetaMorph effectively utilizes all the available accelerators (CPUs and GPUs) within a node and across nodes. Unlike the reference MPI+OpenMP approach, the MetaMorph version transparently maps the workload to hardware accelerators from different vendors with diverse execution models and programming approaches. The workload is distributed based on the relative performance of the accelerators and the available on-device memory. In the strong scaling problem, the performance gap between MetaMorph and the reference parallel implementation decreases at lower node counts, as most of the workload is mapped to the host, due to the limited GPU memory. However, at larger node counts, MetaMorph achieves up to 6x speedup over the reference implementation.



(a) Weak scaling: 512X512X512 local grid per node



(b) Strong scaling: 1024X1024X1024 global grid

Figure 7.11: The scalability analysis of MiniGhost on the HokieSpeed cluster with MetaMorph vs. the reference MPI+OpenMP implementation.

7.5 Conclusion

This chapter presented MetaMorph, a middleware designed to enable interoperation and workload distribution across a range of current and future accelerator architectures, without a significant time investment in development and learning platform-specific nuances. We showed how the core principles of adaptivity, abstraction, and interoperability are instantiated in the MetaMorph prototype implementation. Through these principles, MetaMorph is able to transparently and efficiently map common computation/communication patterns across several compute nodes bearing hardware accelerators, from different vendors, with diverse execution models and programming approaches.

The experimental results demonstrated that while MetaMorph has comparable programmability and productivity to directive-based programming models, it provides performance and interoperability across an array of heterogeneous devices, including multi-core CPUs, Intel MICs, AMD GPUs, and NVIDIA GPUs. In addition, high performance similar to domain- and accelerator-specific approaches is achievable through the MetaMorph library. Further, by effectively utilizing all the available hardware accelerators, within a compute node and across compute nodes, MetaMorph achieves an order of magnitude scalable speedup over the reference parallel implementations.

Chapter 8

Conclusion and Future Work

With the increasing demand for performance and energy efficiency, driven by the exponential growth of data sets and machine learning (ML) applications, computer hardware is becoming more complicated and heterogeneous. This paradigm shift in high-performance computing has made it increasingly difficult for end users to extract the potential performance across the various hardware architectures and execution models in a large-scale system.

This dissertation explored the use of graph analytics to reason about the performance bottlenecks of user applications and to tackle the heterogeneity, scalability, and interoperability challenges of modern HPC systems, both at the node level and the cluster level. Specifically, the previous chapters described in detail our (1) graph-structured program representations to bridge the semantic gap between user applications and heterogeneous parallel architectures, and (2) automation frameworks for performance analysis and modeling, runtime adaptation, interoperation, and workload mapping and distribution. The extensive analysis and experiments conducted in this work demonstrated the efficacy of these innovative frameworks and showed that the proposed methods can achieve multiplicative performance gains for challenging applications (with limited parallelism and fine-grained data dependencies) in many important scientific domains, including computational fluid dynamics, computer-aided design, and electronic design automation.

8.1 Dissertation Summary

To summarize, this work made the following contributions:

- The AutoMatch framework projects the relative execution cost of a given sequential code across heterogeneous parallel architectures, which allows efficient workload distribution that simultaneously utilizes all the available compute resources in a system (§3).
- The CommAnalyzer framework estimates the minimum communication cost and maximum scalability of a given sequential code when ported to distributed-memory architectures. As a result, it enables the development of effective parallel implementations that realize the estimated communication and scalability bounds (§4).
- The ATA framework efficiently schedules application tasks and manages their fine-grained data dependencies on massively data-parallel architectures. By adapting to the dependency structure of input problems and the processing overhead on target architectures, ATA delivers up to an order-of-magnitude improvement in the end-to-end performance of sparse solvers, compared to the existing GPU task-execution approaches (§5).
- The SPICE-H framework provides a scalable execution of irregular computations with limited and data-dependent parallelism, which have been traditionally restricted to a single computing platform, on heterogeneous distributed-memory systems. For a representative and challenging application, namely, SPICE circuit simulation using the direct method, SPICE-H achieves up to an order-of-magnitude speedup over the optimized multi-threaded implementations (§6).
- The MetaMorph library framework is a middleware that resides between user applications and hardware platforms to support interoperation across an array of heterogeneous parallel architectures, including multi-core CPUs, many-core GPUs, and specialized accelerators. As a result, MetaMorph reduces the development effort, and it enables a simultaneous execution approach that effectively utilizes all the heterogeneous computing resources in an HPC system (§7).

8.2 Future Work

There are many research opportunities to expand on our program analysis and runtime adaptation methods to address the power efficiency and extreme heterogeneity challenges of future HPC systems.

While the shift to heterogeneous parallel computing has been crucial to attain scalable performance with power efficiency, HPC systems are becoming increasingly constrained by their power consumption. Therefore, the U.S. Department of Energy (DOE) has identified power consumption as a leading design constraint for future HPC systems and it is challenging the HPC community to achieve exascale computing (10^{18} operations per second) under 20-40 MW power budget [123]. This problem is even more complicated for the emerging machine learning and data analytics applications. Recently, Strubell et al. showed that training a single deep learning model can emit as much carbon as five cars in their lifetime [182]. Hence, there is an emerging need for automated runtime adaptations to sustain the performance improvement under strict power budget. To this end, the proposed automation frameworks for runtime analysis and adaptation can be extended to generate multi-objective cost models for relative performance and energy efficiency, laying the foundation for an intelligent parallel execution runtime.

Due to the end of transistor scaling, another paradigm shift in HPC systems is expected to happen over the next decade [71, 191, 192]. In this era, HPC systems will be built from an aggregation of custom accelerators specialized for different types of workloads. This extreme heterogeneity will significantly increase the complexity of developing HPC applications. To improve the end user productivity, the machine learning community is researching domain-specific languages (DSL), compilers, and runtime systems [38, 190] to adapt their rapidly changing workloads to the latest heterogeneous hardware. Such DSL systems separate the hardware-agnostic algorithms from the hardware-specific optimizations. Specifically, this separation of concerns is achieved by allowing end users to describe their workloads at a high level using domain-specific program constructs, while offloading the task of efficiently scheduling the computations and memory accesses on a specific architecture to the DSL compiler and runtime. Building on the proposed research for automated program analysis, transformation, and runtime adaptation, future work could explore the use of domain-aware program representations and emerging DSL compilers and runtime systems to support the automated optimization of challenging application kernels (such as sparse linear algebra and graph processing kernels) on heterogeneous parallel architectures.

Bibliography

[1] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] Amir Ali Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. Wireframe: Supporting Data-Dependent Parallelism through Dependency Graph Execution in GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 600–611. ACM, 2017.

[3] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[4] Alok Aggarwal and Jeffrey Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256. ACM, 1990.

[6] Ashwin M. Aji, Lokendra S. Panwar, Feng Ji, Karthik Murthy, Milind Chabbi, Pavan Balaji, Keith R Bisset, James Dinan, Wu-chun Feng, John Mellor-Crummey, et al. MPI-ACC: Accelerator-Aware MPI for Scientific Applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(5):1401–1414, 2015.

[7] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 95–105. ACM, 1995.

[8] José I. Aliaga, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. Accelerating the Task/Data-Parallel Version of ILUPACK's BiCG in Multi-CPU/GPU Configurations. *Parallel Computing*, 85:79–87, 2019.

[9] AMD. Accelerated Parallel Processing Math Libraries (APPML), October 2015. URL `http://developer.amd.com/tools/heterogeneous-computing`.

[10] AMD. Radeon's Next-Generation Vega Architecture. `https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf`, 2017.

[11] Edward Anderson and Yousef Saad. Solving Sparse Triangular Linear Systems on Parallel Computers. *International Journal of High Speed Computing*, 1(01):73–95, 1989.

[12] Hartwig Anzt, Edmond Chow, and Jack Dongarra. Iterative Sparse Triangular Solves for Preconditioning. In *Proceedings of the 21st European Conference on Parallel Processing (Euro-Par)*, pages 650–661. Springer, 2015.

[13] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-Architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 725–737. IEEE, 2015.

[14] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, University of California, Berkeley, 2006.

[15] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[16] Amir Bahmani and Frank Mueller. Scalable Performance Analysis of Exascale MPI Programs through Signature-Based Clustering Algorithms. In *Proceedings of the international conference on Supercomputing (ICS)*, pages 155–164. ACM, 2014.

[17] Ioana Baldini, Stephen J. Fink, and Erik Altman. Predicting GPU Performance from CPU Runs using Machine Learning. In *Proceedings of the IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 254–261. IEEE, 2014.

[18] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. A Regression-Based Approach to Scalability Prediction. In *Proceedings of the 22nd annual international conference on Supercomputing (ICS)*, pages 368–377. ACM, 2008.

[19] Richard F. Barrett, Courtenay T. Vaughan, and Michael A. Heroux. MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies using Stencil Computations in Scientific Parallel Computing. *Sandia National Laboratories, Tech. Rep. SAND*, 5294832, 2011.

[20] Richard F. Barrett, Simon D. Hammond, Courtenay T. Vaughan, Douglas W. Doerfler, Michael A. Heroux, et al. Navigating an Evolutionary Fast Path to Exascale. In *Proceedings of the SC Companion: High Performance Computing, Networking Storage and Analysis (SCC)*, pages 355–365. IEEE, 2012.

[21] 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 (SC)*, pages 1–11. IEEE, 2012.

[22] Amr Bayoumi, Michael Chu, Yasser Hanafy, Patricia Harrell, and Gamal Refai-Ahmed. Scientific and Engineering Computing using ATI Stream Technology. *Computing in Science and Engineering*, 11(6):92–97, 2009.

[23] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 54–67. ACM, 2018.

[24] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[25] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, et al. The gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[26] Guy E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39 (3):85–97, 1996.

[27] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low Depth Cache-Oblivious Algorithms. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2010.

[28] Uday Bondhugula. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 33. ACM, 2013.

[29] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, page 45. ACM, 2013.

[30] Alexandru Calotoiu, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. Fast Multi-Parameter Performance Modeling. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 172–181. IEEE, 2016.

[31] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 52. IEEE, 2011.

[32] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, LLNL, 1999.

[33] Marc Casas, Rosa Badia, and Jesús Labarta. Automatic Analysis of Speedup of MPI Applications. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 349–358. ACM, 2008.

[34] Bryan Catanzaro, Kurt Keutzer, and Bor-Yiing Su. Parallelizing CAD: A Timely Research Agenda for EDA. In *Proceedings of the 45th annual Design Automation Conference (DAC)*, pages 12–17. ACM, 2008.

[35] Mi-Chang Chang, Ibrahim N. Hajj, et al. iPRIDE: A Parallel Integrated Circuit

Simulator using Direct Method. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 304–307. IEEE, 1988.

[36] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE international symposium on workload characterization (IISWC)*, pages 44–54. IEEE, 2009.

[37] Guoyang Chen and Xipeng Shen. Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 407–419. ACM, 2015.

[38] 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 *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018.

[39] Xiaoming Chen, Yu Wang, and Huazhong Yang. NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32(2):261–274, 2013.

[40] Xiaoming Chen, Ling Ren, Yu Wang, and Huazhong Yang. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 26(3):786–795, 2015.

[41] Lung-Sheng Chien. How to Avoid Global Synchronization by Domino Scheme. In *GPU Technology Conference (GTC)*, 2014.

[42] Trishul Madhukar Chilimbi. *Cache-Conscious Data Structures: Design and Implementation.* PhD thesis, University of Wisconsin-Madison, 1999.

[43] Jee Choi, Marat Dukhan, Xing Liu, and Richard Vuduc. Algorithmic Time, Energy, and Power on Candidate HPC Compute Building Blocks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 447–457. IEEE, 2014.

[44] Edmond Chow and Aftab Patel. Fine-Grained Parallel Incomplete LU Factorization. *SIAM journal on Scientific Computing*, 37(2):C169–C193, 2015.

[45] Paul F. Cox, Richard G. Burch, Dale E. Hocevar, Ping Yang, and Berton D. Epler. Direct Circuit Simulation Algorithms for Parallel Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 10(6):714–725, 1991.

[46] Daivd E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture, A Hardware/Software Approach.* Morgan Kaufmann, 1996.

[47] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, et al. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–12. ACM, 1993.

[48] Kenneth Czechowski and Richard Vuduc. A Theoretical Framework for Algorithm-Architecture Co-design. In *Proceedings of the IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 791–802. IEEE, 2013.

[49] Kenneth Czechowski, Casey Battaglino, Chris McClanahan, Aparna Chandramowlishwaran, and Richard Vuduc. Balance Principles for Algorithm-architecture Co-design. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar)*, pages 9–9, 2011.

[50] Mayank Daga, Zachary S. Tschirhart, and Chip Freitag. Exploring Parallel Programming Models for Heterogeneous Computing Systems. In *Proceedings of the IEEE In-*

ternational Symposium on Workload Characterization (IISWC), pages 98–107. IEEE, 2015.

[51] Timothy A. Davis. *Direct Methods for Sparse Linear Systems.* SIAM, 2006.

[52] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[53] Timothy A. Davis and Ekanathan Palamadai Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Transactions on Mathematical Software (TOMS)*, 37(3):36, 2010.

[54] Yangdong Steve Deng, Bo David Wang, and Shuai Mu. Taming Irregular EDA Applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD)*, pages 539–546. ACM, 2009.

[55] Roxana E. Diaconescu and Hans P. Zima. An Approach to Data Distributions in Chapel. *International Journal of High Performance Computing Applications*, 21(3): 313–335, 2007.

[56] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on parallel and distributed systems (TPDS)*, 23(8):1369–1386, 2012.

[57] Douglas Doerfler and Ryan Grant. Sandia MPI Micro-Benchmark Suite. `http://www.cs.sandia.gov/smb/index.html`, 2019. Accessed: 2019.12.01.

[58] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. In *Numerical Computations with GPUs*, pages 3–28. Springer, 2014.

[59] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. High-Performance Conjugate-Gradient Benchmark: A New Metric for Ranking High-Performance Computing Systems. *The International Journal of High Performance Computing Applications*, 30(1): 3–10, 2016.

[60] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):1–17, 1988.

[61] Izzat El Hajj. *Techniques for Optimizing Dynamic Parallelism on Graphics Processing Units*. PhD thesis, University of Illinois at Urbana-Champaign, 2018.

[62] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[63] Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, and Ramón Doallo. Evaluation of Messaging Middleware for High-Performance Cloud Computing. *Personal and Ubiquitous Computing*, 17(8):1709–1719, 2013.

[64] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-Driving Intel Xeon Phi. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 137–148. ACM, 2014.

[65] Wu-chun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3801–3804. IEEE, 2010.

[66] Michael Garland and David B. Kirk. Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53(11):58–66, November 2010.

[67] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A Runtime System for Dataflow Task Programming on Heterogeneous Architectures. In *Proceedings of IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1299–1308. IEEE, 2013.

[68] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[69] Thomas George, Vaibhav Saxena, Anshul Gupta, Amik Singh, and Anamitra R. Choudhury. Multifrontal Factorization of Sparse SPD Matrices on GPUs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 372–383. IEEE, 2011.

[70] Pawel Gepner, Michal F. Kowalik, David L. Fraser, and Kazimierz Wackowski. Early Performance Evaluation of New Six-Core Intel Xeon 5600 Family Processors for HPC. In *Proceedings of the Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 117–124. IEEE, 2010.

[71] U.S. Government, White House, Executive Office of the President, President's Council of Advisors on Science, and Technology (PCAST). *Ensuring Long-Term U.S. Leadership in Semiconductors - 2017 Report, Influencing China, Improving U.S. Business Climate, Moonshots for Computing, Bioelectronics, Electric Grid, Weather Forecasting*. Independently published, 2017.

[72] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures. In *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*. IEEE, 1993.

[73] Thomas Grass, César Allande, Adrià Armejach, Alejandro Rico, Eduard Ayguadé, Jesus Labarta, Mateo Valero, et al. MUSA: A Multi-Level Simulation Approach for

Next-Generation HPC Machines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 45. ACM, 2016.

[74] Joseph L. Greathouse, Kent Knox, Jakub Poła, Kiran Varaganti, and Mayank Daga. clSPARSE: A Vendor-Optimized Open-Source Sparse BLAS Library. In *Proceedings of the 4th International Workshop on OpenCL (IWOCL)*, page 7. ACM, 2016.

[75] Kanupriya Gulati, John F. Croix, Sunil P. Khatr, and Rahm Shastry. Fast Circuit Simulation on Graphics Processing Units. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 403–408. IEEE, 2009.

[76] Anshul Gupta and Vipin Kumar. Performance Properties of Large Scale Parallel Systems. *Journal of Parallel and Distributed Computing (JPDC)*, 19(3):234–244, 1993.

[77] Manish Gupta and Prithviraj Banerjee. Compile-Time Estimation of Communication Costs on Multicomputers. In *Proceedings of the International Parallel Processing Symposium (IPPS)*, pages 470–475. IEEE, 1992.

[78] Ahmed E. Helal, Amr M. Bayoumi, and Yasser Y. Hanafy. Parallel Circuit Simulation using the Direct Method on a Heterogeneous Cloud. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 186:1–186:6. ACM, 2015.

[79] Ahmed E. Helal, Paul Sathre, and Wu-chun Feng. MetaMorph: A Library Framework for Interoperable Kernels on Multi- and Many-Core Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 11:1–11:11. IEEE, 2016.

[80] Ahmed E. Helal, Wu-chun Feng, Changhee Jung, and Yasser Y. Hanafy. AutoMatch: An Automated Framework for Relative Performance Estimation and Workload Distribution on Heterogeneous HPC Systems. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 32–42. IEEE, 2017.

[81] Ahmed E. Helal, Wu-chun Feng, Changhee Jung, and Yasser Y. Hanafy. CommAnalyzer: Automated Estimation of Communication Cost on HPC Clusters using Sequential Code. Technical report, Virginia Tech, 2017.

[82] Ahmed E Helal, Changhee Jung, Wu-chun Feng, and Yasser Y Hanafy. CommAnalyzer: Automated Estimation of Communication Cost and Scalability on HPC Clusters from Sequential Code. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 80–91. ACM, 2018.

[83] Ahmed E. Helal, Ashwin M. Aji, Michael L. Chu, Bradford M. Beckmann, and Wu-chun Feng. Adaptive Task Aggregation for High-Performance Sparse Solvers on GPUs. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 324–336. ACM, 2019.

[84] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, and et al. Improving Performance via Mini-Applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[85] E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. Mcglaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, pages 377–382. Springer, 1993.

[86] Jonathan Hines. Stepping up to Summit. *Computing in Science and Engineering*, 20 (2):78–82, 2018.

[87] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, 1992.

[88] Victoria Hodge and Jim Austin. A Survey of Outlier Detection Methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[89] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. Netgauge: A Network Performance Measurement Framework. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC)*, pages 659–671. ACM, 2007.

[90] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Multistage Switches are not Crossbars: Effects of Static Routing inHigh-Performance Networks. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 116–125. IEEE, 2008.

[91] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: Simulating Large-scale Applications in the LogGOPS Model. In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 597–604. ACM, 2010.

[92] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance Modeling for Systematic Performance Tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 6. ACM, 2011.

[93] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 152–163. ACM, 2009.

[94] RD Hornung, JA Keasler, and MB Gokhale. Hydrodynamics Challenge Problem. Technical report, Lawrence Livermore National Laboratory (LLNL), 2011.

[95] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs. OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 136–143. IEEE, 2013.

[96] Nikhil Jain, Abhinav Bhatele, Michael P. Robson, Todd Gamblin, and Laxmikant V. Kale. Predicting Application Performance using Supervised Learning on Communication Features. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 95. ACM, 2013.

[97] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 519–536. ACM, 2011.

[98] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, et al. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 919–932. IEEE, 2013.

[99] George Karypis. hMETIS 1.5.3: A Hypergraph Partitioning Package, 2007. URL `http://glaros.dtc.umn.edu/gkhome/metis/hmetis`.

[100] George Karypis and Vipin Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 1–13. ACM, 1998.

[101] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[102] Eric R. Keiter, Heidi K. Thornquist, Robert J. Hoekstra, Thomas V. Russo, Richard L. Schiek, and Eric L. Rankin. Parallel Transistor-Level Circuit Simulation. In *Simulation and Verification of Electronic and Biological Systems*, pages 1–21. Springer, 2011.

[103] Wei keng Liao. The Software Package of Parallel K-means. `http://users.eecs.northwestern.edu/wk-liao/Kmeans/index.html`, 2018. Accessed: 2019.12.01.

[104] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[105] Darren J. Kerbyson and Kevin J. Barker. Automatic Identification of Application Communication Patterns via Templates. *ISCA PDCS*, 5:114–121, 2005.

[106] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 535–546. IEEE, 2010.

[107] Minjang Kim, Pranith Kumar, Hyesoon Kim, and Bevin Brett. Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1318–1329. IEEE, 2012.

[108] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8):30–37, 2009.

[109] Scott J. Krieder, Justin M. Wozniak, Timothy Armstrong, Michael Wilde, Daniel S. Katz, Benjamin Grimmer, Ian T. Foster, and Ioan Raicu. Design and Evaluation of the GeMTC Framework for GPU-Enabled Many-Task Computing. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 153–164. ACM, 2014.

[110] Gabriel Kron. *Diakoptics: The Piecewise Solution of Large-Scale Systems*, volume 2. MacDonald, 1963.

[111] Viktor Kuncak and Martin Rinard. Existential Heap Abstraction Entailment is Undecidable. In *Proceedings of the 10th International Static Analysis Symposium (SAS)*, pages 418–438. Springer, 2003.

[112] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Run-

times. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops*, pages 29–38. IEEE, 2014.

[113] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO)*, page 75. IEEE, 2004.

[114] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. COMPASS: A Framework for Automated Performance Modeling and Prediction. In *Proceedings of the ACM on International Conference on Supercomputing (ICS)*, pages 405–414. ACM, 2015.

[115] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, ISCA '10, pages 451–460. ACM, 2010.

[116] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, pages 109–118. ACM, 2015.

[117] Peng Li et al. Parallel Circuit Simulation: A Historical Perspective and Recent Developments. *Foundations and Trends® in Electronic Design Automation*, 5(4):211–318, 2012.

[118] Ruipeng Li and Yousef Saad. GPU-Accelerated Preconditioned Iterative Linear Solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[119] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Proceedings*

of the 22nd European Conference on Parallel Processing (Euro-Par), pages 617–630. Springer, 2016.

[120] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides. *Concurrency and Computation: Practice and Experience*, 29(21): e4244, 2017.

[121] 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, 2014.

[122] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. Graphlab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041*, 2014.

[123] Robert Lucas, J Ang, K Bergman, S Borkar, W Carlson, L Carrington, G Chiu, R Colwell, W Dally, J Dongarra, et al. Top Ten Exascale Research Challenges. *DOE ASCAC subcommittee report*, pages 1–86, 2014.

[124] Dimitar Lukarski. PARALUTION Project, 2015. URL `http://www.paralution.com`.

[125] Joe Macri. AMD's Next-Generation GPU and High-Bandwidth Memory Architecture: FURY. In *IEEE Hot Chips 27 Symposium (HCS)*, pages 1–26. IEEE, 2015.

[126] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi-and Many-Core Architectures. In *In Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 300–307. IEEE, 2011.

[127] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[128] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 23–23, 1996.

[129] David S. Medina, Amik St.-Cyr, and Timothy Warburton. OCCA: A Unified Approach to Multi-Threading Languages. *arXiv preprint arXiv:1403.0968*, 2014.

[130] H Meuer, E Strohmaier, J Dongarra, and HD Simon. Top 500 supercomputers, 2019. URL `http://www.top500.org`.

[131] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic Points-to Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 66–72. ACM, 2001.

[132] S.S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[133] Aaftab Munshi. The OpenCL Specification. In *Proceedings of the IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[134] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. 2010.

[135] Laurence W. Nagel. SPICE2: A Computer Program to Simulate Semiconductor Circuits. *Ph. D. dissertation, University of California at Berkeley*, 1975.

[136] Maxim Naumov. Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. Technical Report NVR-2011-001, NVIDIA, 2011.

[137] Maxim Naumov. Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU. Technical Report NVR-2012-003, NVIDIA, 2012.

[138] Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU. *NVIDIA White Paper*, 2015.

[139] P Nenzi. Ngspice Circuit Simulator. URL `http://www.ngspice.org`.

[140] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471. ACM, 2013.

[141] NVIDIA. Compute Unified Device Architecture Programming Guide. 2007.

[142] NVIDIA. CuBLAS Library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008.

[143] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *Comput. Syst*, 26:63–72, 2009.

[144] DoD High Performance Computing Modernization Program Office. Department of Defense High Performance Computing Modernization Program FY 2010 Requirements Analysis Report. Technical report, DTIC Document, 2010.

[145] OpenACC Working Group and others. The OpenACC Application Programming Interface, 2011.

[146] OpenMP, ARB. OpenMP 4.0 Specification, May 2013.

[147] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. Fine-Grain Task Aggregation and Coordination on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 181–192. IEEE, 2014.

[148] Necati Ozisik. *Finite Difference Methods in Heat Transfer*. CRC press, 1994.

[149] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver. In *Proceedings of the 29th International Supercomputing Conference (ISC)*, page 124. Springer, 2014.

[150] David Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. 2009.

[151] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 55–. ACM, 2003.

[152] Lawrence T. Pillage, Ronald A. Rohrer, and Chandramouli Visweswariah. *Electronic Circuit and System Simulation Methods (SRE)*. McGraw-Hill, Inc., 1998.

[153] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *IEEE Computer Architecture Letters*, 14 (1):34–36, 2015.

[154] Sooraj Puthoor, Ashwin M. Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M. Beckmann, and Gregory Rodgers. Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU)*, pages 53–62. ACM, 2016.

[155] T. Quarles. *SPICE 3 Version 3F5 User's Manual*. Departement Electrical Engineering Computer Sciences University of California, 1994. URL `https://books.google.com/books?id=Nk2IMwEACAAJ`.

[156] Ganesan Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[157] Edwin D. Reilly. *Milestones in Computer Science and Information Technology*. Greenwood Publishing Group, 2003.

[158] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. Accelerating Sparse Cholesky Factorization on GPUs. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 9–16. IEEE, 2014.

[159] Justin Riley. Starcluster. URL `http://star.mit.edu/cluster/`.

[160] German Rodriguez, Rosa Badia, and Jesús Labarta. Generation of Simple Analytical Models for Message Passing Applications. In *European Conference on Parallel Processing (EuroPar)*, pages 183–188, 2004.

[161] Phil Rogers. Heterogeneous System Architecture Overview. In *IEEE Hot Chips 25 Symposium (HCS)*, pages 1–41. IEEE, 2013.

[162] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 72–83. IEEE, 2012.

[163] Philip C. Roth, Jeremy S. Meredith, and Jeffrey S. Vetter. Automated Characterization of Parallel Application Communication Patterns. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 73–84. ACM, 2015.

[164] Karl Rupp, Florian Rudolf, and Josef Weinbub. ViennaCL-A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. pages 51–56, 2010.

[165] Yousef Saad. Krylov Subspace Methods on Supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989.

[166] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.

[167] Resve A. Saleh, Kyle A. Gallivan, M-C Chang, Ibrahim N. Hajj, David Smart, and Timothy N. Trick. Parallel Circuit Simulation on Supercomputers. *Proceedings of the IEEE*, 77(12):1915–1931, 1989.

[168] Joel H. Saltz. Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors. *SIAM journal on scientific and statistical computing*, 11(1):123–144, 1990.

[169] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[170] Paul Sathre, Mark Gardner, and Wu-chun Feng. Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation. In *In Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 89–96. IEEE, 2012.

[171] Thomas Scogland, Wu-chun Feng, Barry Rountree, and Bronis de Supinski. CoreTSAR: Core Task-Size Adapting Runtime. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 26(11):2970–2983, 2015.

[172] Leonid Ivanovich Sedov. *Similarity and Dimensional Methods in Mechanics*. CRC press, 2018.

[173] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[174] Junaid Shuja, Kashif Bilal, Sajjad A Madani, Mazliza Othman, Rajiv Ranjan, Pavan Balaji, and Samee Khan. Survey of Techniques and Architectures for Designing Energy-Efficient Data Centers. *IEEE Systems Journal*, 10(2):507–519, 2016.

[175] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A Framework for Performance Modeling and Prediction. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, pages 21–21. ACM, 2002.

[176] Kyle L. Spafford and Jeffrey S. Vetter. Aspen: A Domain Specific Language for Performance Modeling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 84:1–84:11. ACM, 2012.

[177] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin Slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 112–122. ACM, 2007.

[178] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic Scheduling on GPUs. *ACM Transactions on Graphics (TOG)*, 31(6):161, 2012.

[179] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU. *ACM Transactions on Graphics (TOG)*, 33(6):228, 2014.

[180] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12(1-3):66–73, 2010.

[181] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[182] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. *arXiv preprint arXiv:1906.02243*, 2019.

[183] Brad Suchoski, Caleb Severn, Manu Shantharam, and Padma Raghavan. Adapting Sparse Triangular Solution to GPUs. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 140–148. IEEE, 2012.

[184] Rabin A. Sugumar and Santosh G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling Computer Systems (SIGMETRICS)*. ACM, 1993.

[185] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[186] Nathan R. Tallent and Adolfy Hoisie. Palm: Easing the Burden of Analytical Performance Modeling. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 221–230. ACM, 2014.

[187] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T. Kandemir, and Chita R. Das. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 649–660. IEEE, 2017.

[188] Heidi K. Thornquist, Eric R. Keiter, Robert J. Hoekstra, David M. Day, and Erik G. Boman. A Parallel Preconditioning Strategy for Efficient Transistor-Level Circuit Simulation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 410–417. ACM, 2009.

[189] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[190] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[191] Jeffrey S. Vetter, Erik P. DeBenedictis, and Thomas M. Conte. Architectures for the Post-Moore Era. *IEEE Micro*, (4):6–8, 2017.

[192] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. Extreme Heterogeneity 2018-Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity. Technical report, Lawrence Berkeley National Lab (LBNL), Berkeley, CA (United States), 2019.

[193] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel Xeon Phi*, pages 167–188. Springer, 2014.

[194] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 528–540. ACM, 2015.

[195] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 583–595. IEEE, 2016.

[196] Wei Wang, Lifan Xu, John Cavazos, Howie Huang, and Matthew Kay. Fast Acceleration of 2D Wave Propagation Simulations using Modern Computational Accelerators. *PloS one*, 9(1), 2014.

[197] Xinliang Wang, Wei Xue, Weifeng Liu, and Li Wu. swSpTRSV: A Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 338–353. ACM, 2018.

[198] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gun-

rock: GPU Graph Analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4 (1):3:1–3:49, 2017.

[199] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC First Experiences with Real-World Applications. In *Proceedings of the European Conference on Parallel Processing (Euro-Par)*, pages 859–870. Springer, 2012.

[200] 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.

[201] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve Tjiang, Shih-Wei Liao, et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM Sigplan Notices*, 29(12): 31–37, 1994.

[202] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 235–246. IEEE, 2010.

[203] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing Data Warehousing Applications for GPUs using Kernel Fusion/Fission. In *IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*, pages 2433–2442. IEEE, 2012.

[204] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, et al. Top 10 Algorithms in Data Mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[205] Xing Wu, Frank Mueller, and Scott Pakin. Automatic Generation of Executable Communication Specifications from Parallel Applications. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 12–21. ACM, 2011.

[206] Shucai Xiao and Wu-chun Feng. Inter-block GPU Communication via Fast Barrier Synchronization. In *Proceedings of the IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 1–12. IEEE, 2010.

[207] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 221–234. ACM, 2017.

[208] Sencer Nuri Yeralan, Timothy A. Davis, Wissam M. Sid-Lakhdar, and Sanjay Ranka. Algorithm 980: Sparse QR Factorization on the GPU. *ACM Transactions on Mathematical Software (TOMS)*, 44(2):17, 2017.

[209] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines using a Single Node. In *In Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP)*, pages 305–314. ACM, 2010.

[210] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: A Versatile Programming Framework for Pipelined Computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 587–599. ACM, 2017.