Program Transformation for Automatic GPU-Offloading using OpenMP

A Dissertation Presented

by

Alok Mishra

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2022
Stony Brook University

The Graduate School

Alok Mishra

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation

Dr. Barbara Chapman - Dissertation Advisor
Distinguished Professor, Department of Computer Science

Dr. Rezaul A. Chowdhury - Chairperson of Defense
Associate Professor, Department of Computer Science

Dr. Klaus Mueller
Professor, Department of Computer Science

Dr. Abid M. Malik
Senior Technology Engineer, Computational Science Initiative
Brookhaven National Laboratory

This thesis proposal is accepted by the Graduate School

Celia Marshik
Interim Dean of the Graduate School
Abstract of the Dissertation

Program Transformation for Automatic GPU-Offloading using OpenMP

by

Alok Mishra

Doctor of Philosophy

in

Computer Science

Stony Brook University

2022

Due to their extensive parallelism and energy efficiency, GPU-based HPC clusters are drawing an increasing number of users and developers. However, with the increasing variety of GPUs available, it is impractical to write separate code that is appropriate for each target system for a specific HPC application. Another challenging feature of a GPU, besides portability, is its programmability. Programming in native languages, like CUDA, is extremely difficult for an application developer because they depend heavily on in-depth knowledge of the model and the underlying architecture. Furthermore, because the physical memory for the CPU and GPU are different, they must ensure data synchronization between the two. Identifying suitable code areas in a legacy application to offload and explicitly handling data between hosts and devices are other issues with GPU offloading. The best option for an application developer is to use directive-based parallel programming models, like OpenMP. Nevertheless, even with OpenMP, the developer must choose from among a number of strategies for offloading the kernel to a GPU. Therefore, once it has been decided that the code should be offloaded to a GPU, the application developer will greatly benefit from a tool that will help them in making decisions about what data to offload and what approach to use when offloading code to the GPU.
In this thesis, I present a first of its kind compiler tool that takes a C code as input, recognizes an OpenMP parallel loop, and then offers several kernel variants for GPU offloading. Using a compile time cost model, it then statically identify the kernel that is best suited for GPU offloading. This tool is divided into three modules. We chose to design our tool using a modular approach so that we can replace a module as needed without affecting the other modules.

The first is the Kernel Analysis module, which detects and analyzes an OpenMP kernel and suggests several variants, by applying various potential code level transformations, for offloading that kernel to a GPU. The main objective behind this tool is to maintain the portability of OpenMP. As such to maintain the portability, I am performing all the analysis and data extraction on the AST itself and not going down the level in compiler. Moreover, this module performs an essential verification to ensure that the OpenMP transformation is accurate. The second module is a Compiler Cost Model, that estimates the cost of executing the original code and the different offloading code variants. Modern compilers typically use analytical cost models for this purpose. Regrettably, creating a cost model for a compiler optimization, particularly GPU offloading, is an extremely challenging task that requires a lot of effort from a compiler engineer. Additionally, it seems implausible to try to develop an analytical cost model that is portable across various GPU architectures. Recently, cost models have been successfully developed using Machine Learning techniques for a variety of compiler optimization issues. However, a cost model for GPU offloading is still lacking in this situation. So, in this module, I define COMPOFF, a cost model that statically calculates the Cost of OpenMP OFFlooding using neural networks. Our initial reports demonstrate that using COMPOFF, one can estimate the cost of offloading OpenMP kernels with an accuracy ranging from 95% to 99%. This accuracy is independent of different compilers and GPU architectures. In the third module we modify the original source code using the analysis and prediction from the other modules and return newly generated code that supports GPU offloading.

Although the tool is currently limited to GPUs, it can be extended to support other OpenMP-capable devices. The main objective behind this tool is to maintain the portability aspect of OpenMP. Preliminary findings indicate that this tool can assist compiler developers and HPC application researchers in porting their legacy HPC codes to the upcoming heterogeneous computing environment.
For my very inquisitive son, Akshat.

You made the pandemic bearable. Thank you for being a patient and consistent audience for all of my practice talks, and for asking me “insightful” questions throughout my PhD journey.

Never stop asking WHY.
# Table of Contents

List of Tables \hfill x

List of Figures \hfill xi

Listings \hfill xiv

Acknowledgements \hfill xv

Publications \hfill xvi

Awards and Recognition \hfill xviii

## 1 Introduction

1.1 Research Goals and Contributions \hfill 5

1.2 Dissertation Organization \hfill 6

## 2 Background and Related Work

2.1 The Exascale Computing Project \hfill 8

2.2 GPU \hfill 11

2.2.1 Graphics before GPUs - a brief history \hfill 11

2.3 Programming GPUs \hfill 12

2.3.1 CUDA \hfill 13

2.3.2 OpenCL \hfill 15

2.3.3 OpenACC \hfill 18

2.3.4 OpenMP \hfill 19

2.4 Compilers - LLVM \hfill 22

2.4.1 Clang \hfill 23

2.4.2 How does Clang work? \hfill 23

2.4.3 Abstract Syntax Tree (AST) \hfill 24
3 Motivation

3.1 Introduction ................................................. 29
3.2 Benchmark Development ................................. 32
  3.2.1 Benchmarking OpenMP GPU Offloading ............. 34
  3.2.2 Benchmarking Unified Memory ...................... 37
3.3 Evaluation ...................................................... 39
  3.3.1 Experimental Methodology ...................... 39
  3.3.2 GPU Offloading without Unified Memory ........... 41
  3.3.3 Unified Memory ................................... 43
3.4 Improving Unified Memory Performance .................. 46
3.5 Related Work .................................................. 47
3.6 Conclusion ..................................................... 48

4 State of the Art ................................................. 50

4.1 GPU Offloading ............................................. 50
  4.1.1 The Benefits of Using GPUs ..................... 51
  4.1.2 NVIDIA Thread Organization .................... 52
  4.1.3 Streaming Multiprocessors ..................... 53
  4.1.4 NVIDIA Compute Model ........................... 55
4.2 OpenMP Offloading ........................................ 55
4.3 GPU Cost Model ............................................. 56
4.4 What is missing? .............................................. 59

5 OpenMP Advisor ............................................... 60

5.1 Related Work ................................................ 63
5.2 OpenMP Advisor ........................................... 64
  5.2.1 Core Attributes ................................ 64
  5.2.2 Design ............................................. 66
5.3 Clang Plugins Implementation ............................ 69
  5.3.1 Kernel Analysis ................................ 70
  5.3.2 Cost Model ........................................ 73
  5.3.3 Kernel Transformation ............................ 76
## List of Tables

3.1 Rodinia Benchmark details ........................................ 33
3.2 Compute node configuration on SummitDev. ................. 40
5.1 Number of Variants generated for each number of for loops ... 73
5.2 Benchmark Applications. ....................................... 75
6.1 Data Management in Optimized Code vs Base Code for benchmark applications ............................. 93
6.2 Result comparing Base code with Optimized code for time taken by different APIs when run on NVIDIA Tesla K80 ... 96
6.3 Result comparing Base code with Optimized code for time taken by different APIs when run on NVIDIA Tesla P100 ... 96
6.4 Result comparing Base code with Optimized code for time taken by different APIs when run on NVIDIA Tesla V100 ... 97
6.5 % Improvement in performance on NVIDIA Tesla P100 and NVIDIA Tesla V100 when compared to NVIDIA Tesla K80 ... 102
7.1 Static Kernel features independent of architectures and compilers .................................................. 113
7.2 Evaluation results on the micro-benchmark data files for respective transform and cluster ..................... 119
7.3 Evaluation results on Wilson Dslash kernel data ................ 120
8.1 Cluster and Compilers used in experiment ................... 125
8.2 Number of variants generated for all applications .......... 128
9.1 Updated benchmarks from the Rodinia Benchmark Suite ... 152
9.2 Size of executables on Summit and Seawulf (in bytes) ... 154
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Oak Ridge National Laboratory’s Journey from Petascale to Exascale [110]</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Energy Efficiency for different ORNL HPC clusters [110]</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>OpenMP Parallel Threads</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Growing Relevance of OpenMP: NERSC</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>LLVM Model</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>LLVM Compiler Design</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Traditional OpenMP GPU offloading without unified memory</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Two level of parallelism</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>OpenMP GPU offloading with unified memory</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance of CPU, GPU without unified memory, and GPU with unified memory</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Execution time breakdown</td>
<td>44</td>
</tr>
<tr>
<td>3.6</td>
<td>Data transfer breakdown</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Transistors Dedicated to Data Processing on CPU vs GPU</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>CUDA Thread Hierarchy</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Technical Specifications per Compute Capability [45]</td>
<td>54</td>
</tr>
<tr>
<td>5.1</td>
<td>LLVM Architecture</td>
<td>66</td>
</tr>
<tr>
<td>5.2</td>
<td>High level flow of the Training mode</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>High level flow of the Prediction mode</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>Workflow for identifying data transfer opportunities in an application using OpenMP for GPU offloading</td>
<td>86</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparison of data transferred between CPU and GPU for different benchmarks</td>
<td>98</td>
</tr>
<tr>
<td>6.3</td>
<td>Number of calls to CUDA APIs for Data Management</td>
<td>99</td>
</tr>
</tbody>
</table>
6.4 Comparison of time taken for different data management APIs vs kernel computation time on V100 GPU .................................. 100
6.5 % Improvement in Compute time and Data Management time on NVIDIA Tesla P100 and NVIDIA Tesla V100, keeping NVIDIA Tesla K80 as baseline. ........................................ 101

7.1 Teams distribute parallel for ........................................ 109
7.2 Overview of the ML Model ........................................... 117
7.3 Validation of Combined Offload on Summit and Seawulf .... 121
7.4 Validation of Collapse Offload on Summit and Seawulf .... 121
7.5 Wilson Dslash operator prediction on Summit and Seawulf ... 122

8.1 Total Data transfer for all Benchmarks Before and After Data Analysis ................................................................. 126
8.2 Data Transfer time (ms) for Wilson Dslash Operator on different Clusters for Before (1.5GB) and After (0.75GB) transformation ................................................................. 127
8.3 Validation of Cost Model on different clusters ................. 129
8.4 Wilson Dslash operator’s Actual and Predicted Runtimes (in sec) on Summit for all variants sorted by runtime, where $L_X, L_Y, L_Z$ and $L_T$ are set to 32, 32, 32 and 16. ......................... 131
8.5 RMSE and Normalized RMSE for predicting the runtime of all variants of Wilson Dslash operator on different clusters for $L_X, L_Y, L_Z, L_T$ set at 32, 32, 32, 16 each. The range of runtimes for each cluster is mentioned below their name. .......... 131

9.1 C Pre-processing vs OpenMP Metadirective usage comparison. ................................................................. 135
9.2 Execution time comparison for the x solve region of BT using different OpenMP runtime configurations at different power levels. Smaller value is better. The function was run on Intel Sandy Bridge [10]. ................................................................. 138
9.3 Compile time selection of hardware support. .................. 139
9.4 Metadirective Syntax .................................................. 140
9.5 Context Selector Syntax ............................................. 141
9.6 Common 3-Phase Compiler Design ............................... 143
9.7 OpenMP Metadirective Dynamic User Selection ............. 145
9.8 Matrix Multiplication implementation without metadirective 146
9.9 Matrix Multiplication implementation with static metadirective
9.10 Matrix Multiplication implementation with dynamic metadirective
9.11 Execution time (in sec) of matrix multiplication for 3 different implementations
9.12 Percentage increase in size of the executable built with our dynamic metadirective implementation in LLVM 12.0.0
9.13 Percentage change in running time of the executable built with our dynamic metadirective implementation
10.1 Dataset Splitting Approach
10.2 Modification to AST to create Augmented AST for ParaGraph
## Listings

2.1 Serial Matrix Multiplication ........................................ 13
2.2 Matrix Multiplication using CUDA ................................. 14
2.3 Matrix Multiplication using OpenCL ............................... 16
2.4 OpenACC Matrix Multiplication ................................. 19
2.5 OpenMP Matrix Multiplication ................................. 21
3.1 OpenMP GPU offloading code example ......................... 35
3.2 The BPNN structure in Back propagation benchmark ........ 37
3.3 Unified memory code example ........................................ 39
5.1 Loops of Wilson Dslash Operator .................................. 61
5.2 Four nested “for” loops, with “teams distribute” in loop position 0 and “parallel for” in loop position 1 .......... 72
5.3 Location of the seven strings needed for generating the code ........................................ 78
6.1 Code snippet for Loop Check ........................................ 88
7.1 Sequential Matrix Multiplication program ....................... 108
7.2 Matrix Multiplication on GPU using OpenMP ................... 109
7.3 Combined parallelism .................................................. 110
7.4 Split parallelism ....................................................... 110
7.5 Swap Combined parallelism .......................................... 110
7.6 Swap Split parallelism .............................................. 111
7.7 Collapse Loop ....................................................... 111
7.8 Swap Collapse Loop .................................................. 111
Acknowledgements

First and foremost, I’d like to express my gratitude to Dr. Barbara Chapman, my research advisor, and Dr. Abid Malik, my mentor, for taking me under their wings, and providing me with ample guidance within their research group Exasca||ab (Exascale Programming Models Laboratory). Thanks also to Dr. Rezaul A. Chowdhury and Dr. Klaus Mueller, members of my thesis committee, for their time and valuable feedback. Special thanks to Tony Curtis for all the guidance provided throughout my PhD journey.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

I would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the SeaWulf computing system, which was made possible by a $1.4M National Science Foundation grant (#1531492).

Last but not least, I’d like to express my gratitude to my wife, Neha Ojha, and our entire family for their unwavering patience and support throughout this journey.
Publications

Conferences/Workshops

1. **COMPOFF: A Compiler Cost model using Machine Learning to predict the Cost of OpenMP Offloading.**
   Alok Mishra, Smeet Chheda, Carlos Soto, Abid Malik, Meifeng Lin and Barbara Chapman. In 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

2. **Extending the LLVM/Clang Framework for OpenMP Metadirective Support.**
   Alok Mishra, Abid Malik and Barbara Chapman. In LLVM-HPC 2020: Sixth Workshop on the LLVM Compiler Infrastructure in HPC, November 2020 - Atlanta, Georgia, USA.

3. **Data Transfer and Reuse Analysis Tool for GPU-offloading using OpenMP.**
   Alok Mishra, Abid Malik and Barbara Chapman. In the International Workshop on OpenMP (IWOMP2020), September 2020, Texas, USA.

4. **Kernel fusion/decomposition for automatic GPU-offloading.**
   Alok Mishra, Martin Kong, and Barbara Chapman.

5. **Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading.**
Journals

1. FreeCompilerCamp.org: Training for OpenMP Compiler Development from Cloud.

Paper under Review

1. OpenMP Advisor: A Compiler Tool for Heterogeneous Architectures.
   Alok Mishra, Abid M. Malik, Meifeng Lin and Barbara Chapman.

Poster Presentations

1. Improved Machine Learning Strategy for OpenMP Target Offloading in the 19th Workshop on Compiler-Driven Performance held in conjunction with CASCON 2022.


Awards and Recognition

1. The work on “Program Transformation for Automatic GPU-Offloading with OpenMP” was selected in the top five entries in ACM Student Research Competition of Parallel Architectures and Compilation Techniques (PACT) 2021.

2. Won the Silver medal in ACM Student Research Competition of SuperComputing (SC) 2019 for work on “Data Reuse Analysis for GPU Offloading using OpenMP”.

3. The poster on “FreeCompilerCamp: Online Training for Extending Compilers” was selected in the top 5 best posters in Research Poster category in SuperComputing (SC) 2019.

4. Won the Bronze medal in ACM Student Research Competition of International Symposium on Code Generation and Optimization (CGO) 2019, for work on “Kernel fusion/decomposition for automatic GPU-offloading”.

xviii
Chapter 1

Introduction

“For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution.”


The development of high-performance computing (HPC) has been significantly influenced for over half a century by Amdahl’s law [5] and Moore’s law [101]. When Amdahl stated in his law that the inherently serial portion of a computation imposes a limit on the potential speedup of parallel processing, he was pessimistic about the success of parallel processing. Although the validity of Amdahl’s law has been the subject of considerable debate, the mathematical relationship it expresses has an unquestionably logical conclusion. On the other hand, Moore’s law articulated an empirical finding about patterns in technological capability and manufacturing effectiveness. Moore observed that the number of transistors in a dense integrated circuit (IC) doubles about every two years. The specifics of Moore’s law were modified over time to more accurately reflect the growth in transistor density, but the exponential nature of Moore’s law persisted, providing the semiconductor industry with decades of significant opportunity. While Moore’s law inspired great optimism for the future of computing in general and eventually led to the current pervasiveness of computers in daily life, Amdahl’s law sparked widespread skepticism about the ultimate potential of parallel computing.

Thankfully, as problem size increases, the effective fraction of a serial
code typically reduces, making Amdahl’s speedup and efficiency constraints less stringent. Gustafson Law [46], published in 1988, suggests that programmers can increase the size of problems in order to fully utilize more powerful computing systems, correcting Amdahl’s Law’s flaws (which assume fixed problem size).

Over two decades have passed since what has been referred to as “the death of Moore’s law” [140] and the transition to multi-core processors from ever-faster single chips. However, it could be argued that given the rising number of transistors on a chip, Moore’s law has not completely broken (yet), rather it is no longer possible to run these transistors at ever-increasing speeds. This is due to Dennard scaling [30], a related effect, which coincided with the reduction in transistor size. The main outcome of Dennard scaling was that power density remained constant as transistor size decreased. Dennard had found that because voltage and current scale proportionally with feature size and power scales proportionally with area, performance per watt doubles roughly every two years. Consequently, what went wrong was not the ability to etch smaller transistors, but rather the capacity to reduce the voltage and current they require to function properly.

Since the end of Dennard scaling [126], Amdahl’s skepticism about the ultimate potential of parallel computing have grown stronger. Developers in the HPC industry have started looking beyond Moore’s law and hardware developers have improved chip performance by configuring a growing number of compute cores. This rapid change in architecture hardware necessitates programming language updates and modification of application code to exploit the cores, e.g. by inserting pthreads or OpenMP [27] constructs into the source code. The HPC community was quick to embraced this multi-core processor technology.

In the last decade, General Purpose Graphics Processing Units (GPGPUs) have been attached to the multi-core processors on many HPC platforms in order to benefit from their ability to handle large amounts of data parallelism with low power consumption. In order to demonstrate that a system can be built around GPU computing and still be power-efficient, Showerman et al. [127] constructed the EcoG GPU-based cluster. Initially intended for graphics operations, they are now being used extensively by HPC clusters and for general-purpose computing on graphics processing units. Although there are some notable exceptions, the most recent TOP500 list [139] clearly demonstrates the trend toward heterogeneous HPC platforms, particularly those using GPUs. A significant portion of the systems are heterogeneous,
with Intel Xeon Phis or NVIDIA GPUs typically providing high performance per watt. The current fourth-ranked Summit supercomputing cluster [149] is composed of two IBM PowerPC9 multicore CPUs with six NVIDIA V100 GPUs per node and three kinds of memory. A successor to Summit, the top-ranked Frontier supercomputing cluster [124] integrates AMD Epyc 7A53s 64 core CPUs with 4 Radeon Instinct MI250X GPUs per node.

Moving forward, we expect HPC platforms to be more diverse and potentially include domain-specific accelerators designed for specialized paradigms like machine learning, neuromorphic computing and ultimately quantum computing, leading to an era of extreme heterogeneity. However, can such a diverse range of heterogeneous architectures be handled by the software stack of these HPC clusters? Current programming environments are already challenged by the presence of a single type of accelerator on a node; we are not yet ready for a time when multiple types of heterogeneous accelerators may be configured. Many application developers are adapting their codes to exploit GPUs. This is a time-consuming effort that may involve re-engineering data structures as well as large regions of code in order to make the most of the GPU’s computational power and minimize the overheads incurred. It will be far more difficult to develop code for systems with extreme heterogeneity containing different devices. It is therefore imperative to develop techniques that will relieve the application developers of the burden of such development.

One of the ways of handling portability across diverse architectures is to use a directive based programming model like OpenMP, which is the de-facto programming standard for parallel programming in C/C++ and Fortran. Since specification 4.x [107], OpenMP has supported GPU offloading and has plans to transition to extremely heterogeneous architectures [50]. This makes the underlying hardware transparent and highly portable. Nevertheless, even when using directive-based programming models like OpenMP, it is still quite an arduous task to optimize large scale applications with tens-to-hundreds of thousands of lines of code.

To tackle this challenge, compiler engineers are working on a number of frameworks [90, 116] that will automatically help application developers deal with extreme heterogeneity. These frameworks require analytical cost models which will help them to make a better decision in selecting the choice of optimization or transformation needed by the application. However, the efforts to develop a cost function are quite expensive, and almost all modern compilers, including LLVM/Clang, use a simple “one-size-fits-all” cost
function [85], which does not provide the best performance in case of diverse architecture. Currently, hand-tuned cost functions are widely used; but such cost models require a deeper understanding of the underlying hardware in calculating the costs and benefits of a compiler optimization. Despite its effectiveness, manually constructing a cost model for a single architecture can take months. Since cost functions are critical and manual tuning is rather laborious, Machine Learning (ML) techniques are being investigated by compiler engineers as a means of automating this process.

Compiler developers have historically been reluctant to incorporate machine learning methods into compilation. And with good cause. For instance, no one will use a compiler, if the developer claims that, while it produces accurate results 9 out of 10 times, occasionally it may not provide the proper transformation. On the other hand, if a machine learning (ML) developer claims that their model is 90% accurate, it will be viewed as a huge success. It makes sense why we were hesitant to include machine learning in compilation. But recently, a lot of research has been done on how to use ML techniques in compilation as well. Early work exploiting ML in compilers primarily explored its use to help optimize sequential programs [100, 70]. However, its application to the task of optimizing parallel programs has attracted significant attention in the past decade due to the prevalence of multi-core platforms and, more recently, heterogeneous systems. [72, 121]. A decision-tree-based approach has been developed to predict the scheduling policy for an OpenMP parallel region [93]. Also [31] uses machine learning techniques to optimize OpenMP programs for scheduling policies and number of threads. ML has been used to determine the optimum degree of parallelism for transactional memory [153] and hardware resource allocation [64]. The work presented in [9] and [67] applies ML to complex parallel programs and divide them among the available multi-core resources. The Petabricks project [4] employs a genetic search to tune algorithmic choices at compile time to reduce searching overheads. Tree and graph-based features have also been used by Malik et.al. [86] who present a unique graph-based approach for feature representation. ML techniques were used to build classifiers to determine whether to offload OpenCL code [33], and to select a clock frequency at which the processor should operate [61]. A high level of accuracy was reported, however the benefits could not be quantified as the work did not attempt to generate modified code. They also explored regression techniques to build curve fitting models to search for the sweet spot for work partitioning among processors [44] or a trade-off of energy and performance [123].
The results of prior efforts from applying ML on compiler optimizations are encouraging. However, new feature engineering practices need to be explored that can help ML learn more about a code and its computational needs.

1.1 Research Goals and Contributions

The goal of this work is to design and develop a compiler tool which transforms an OpenMP code to effectively offload to a GPU. The driving principle of our work resides in generating numerous possible kernel variants and using a cost model to statically find the best suited kernel which can be offloaded to the GPU.

We chose to design our tool using a modular approach so that we could replace a module whenever necessary without affecting the other modules. We divide the tool into three distinct modules.

1. The first is the Kernel Analysis module, which detects and analyzes an OpenMP kernel and offers several alternatives for offloading that kernel to a GPU by using various potential code level transformations.

2. In the second module, we define COMPOFF, a tool that predicts the Cost of OpenMP Offloading at compile time. We need some kind of analytical cost model to make better decisions about which transformation a kernel requires. However, developing a cost function is quite costly, and almost all modern compilers, use a simple "one-size-fits-all" cost function that does not provide the best estimate. Currently, hand-tuned cost functions are widely used. However manually constructing such cost model for a single architecture can take months. To automate this process, we looked into Machine Learning techniques.

3. Using the analysis and forecast from the other modules, we modify the original source code in the third module, which then outputs newly generated code that supports GPU offloading.

To the best of our knowledge this work is a first of its kind in OpenMP GPU offloading.
1.2 Dissertation Organization

The rest of the dissertation is organized as follows:

– Chapter 2 provides a background on the concepts of HPC, ECP, Supercomputers, GPUs, OpenMP, Compilers, Cost Model and Machine Learning.

– Chapter 3 introduces the problem which motivated the development of this tool.

– Chapter 4 provides information on the state of the art of OpenMP GPU offloading, OpenMP Cost Model and Machine Learning in Compilers.

– Chapter 5 gives a high level design of the proposed tool.

– Chapter 6 discusses an optimization to automatically recognize data which do not need to be transferred between CPU and GPU, and capitalize on any data reuse opportunities in an application.

– Chapter 7 presents COMPOFF, a cost model that statically estimates the Cost of OpenMP OFFloading using a neural network model.

– Chapter 8 covers the final experiments carried out in this dissertation and the analysis of the results.

– Chapter 9 discusses our implementation of OpenMP metadirective, which can be used for future extension of this tool.

– Chapter 10 discusses what the future holds.

– Chapter 11 summarizes our work and provides concluding remarks.
Chapter 2

Background and Related Work

“Today, your cell phone has more computer power than all of NASA back in 1969, when it placed two astronauts on the moon.”

– Michio Kaku in Physics of the Future [68]

Following Moore’s law’s success over the past five decades, Michio Kaku correctly observes the trend in computer architecture that, as time passes, the size of the computers continues to shrink whilst the compute power tends to increase. In his book Physics of the Future [68] he correctly states that a simple smart phone today has more compute power than all of NASA in 1969. But does that undermine the actual computing power of the Apollo mission computers in any way? Definitely not. A system is defined more by its capability than by its computational power. Of course, any modern device has far more computational power than any early machine, but a smart phone cannot actually be used to guide a spacecraft to the moon.

By definition, High Performance Computing (HPC) is the practice of aggregating computing power in a way that produces performance that is significantly higher than what would be possible from a typical desktop computer or workstation. This computing power can be used to solve complicated research problems in science, engineering, or business. Scientists need HPC once they have reached that critical juncture in their research where it becomes necessary to broaden the current study area, or incorporate new data, or increase model resolution, and their desktop or single server no longer suffices. One of the most well-known kinds of HPC solutions is the supercomputer. It is made up of thousands of computing nodes that work together to accomplish one or more tasks in parallel.
The computing power of an HPC system is usually measured in terms of FLOPS or Floating Points Operations Per Second. But how much processing power must a machine have to qualify as HPC? The answer to this query is ambiguous. For instance, the Cray-2 supercomputer, built by Cray Research in 1985, had four vector processors and a peak performance of 1.9 GFLOPS, making it the fastest machine in the world at the time. Today, Google’s Pixel 6 smartphone boasts a peak performance of over 2 TFLOPS, but that is not an HPC system. The ECP project’s Frontier supercomputer now offers a peak capability of 1.102 ExaFLOPS. Such a performance would not have been achievable without the ECP project’s strong coordination between hardware and software stack development.

2.1 The Exascale Computing Project

An HPC system’s superior computing power is ineffective if its software stack is unable to make the most of it. In 2016, the US Department of Energy (DoE) started the Exascale Computing Project (ECP [92]), as a part of a coordinated effort to develop the next generation of HPC and to expedite scientific advancement and economic competitiveness. Exascale computing requires at least a billion billion operations to be performed every second, and getting there wasn’t an easy task. To be useful to a wide spectrum of applications, in addition to peak speed, supercomputers need to have large memories and the ability to store and read vast quantities of data at high speed. The supercomputer must, above all, have a software environment that makes efficient and effective use of it possible. In addition to enabling conventional supercomputing applications at exa-scale, the ECP aims to contribute to the growing convergence of supercomputing, big data analytics, and machine learning across a wide range of science and engineering domains and disciplines to address issues of national importance and to aid in meeting our national security needs.

The strong synergy between the hardware development and the application/software stack development has been a major element of the ECP strategy. As a result, HPC has become a tool with increasing capabilities for both science and business. HPC serves as the cornerstone for developments in science, business, and society. Compute servers, or nodes, are networked collectively to form a cluster in order to develop a high performance computing architecture. Each node in the majority of HPC systems nowadays
is connected to a number of accelerators, like GPUs, to execute high speed computations. Algorithms and software programs run concurrently on the cluster’s nodes. The cluster is connected to the data storage in order to capture the output. These parts work in tandem to perform a wide range of tasks, and for each part to function at its best, it must keep up with the others. For example, the storage component must be able to feed and ingest data to and from compute nodes as quickly as it is processed. Similarly, networking components must be capable of supporting high-speed data transit between compute servers, data storage and accelerators.

The Department of Energy (DOE) is in charge of leading the development of HPC systems in the US, and their national laboratories are home to the fastest and most powerful supercomputers in the country. According to the TOP500 [139] ranking of the top 500 most powerful supercomputers as of June 2022, the US owns five of the top 10 systems in the world, with four of those belonging to the DOE. **Frontier** (rank 1) and **Summit** (rank 4) at the Orange River National Laboratory (Figure 2.1), **Sierra** (rank 5) at the Lawrence Livermore National Laboratory and **Perlmutter** (rank 7) at the Lawrence Berkeley National Laboratory are all projects of the DOE. And hardware technology is not the only thing that characterizes DOE leadership. The success of its research projects in crucial areas of study, such as applied mathematics and computer science, advanced lithography, and nanoscale materials science, is essential to the organization’s leadership.
Figure 2.1 shows how the HPC system at ORNL used Jaguar to reach a peak performance of 2.3 PF. However, it also shows that Jaguar utilized roughly 7MW of energy annually to deliver that performance. If they had attempted to construct an Exascale computer using the same architecture, they would have required roughly 3 GW of energy annually to achieve one ExaFLOPS performance, which is equivalent to about 3 billion USD. The ECP project had set a target of below 20 MW/EF. Figure 2.2 shows a plot of energy efficiency different ORNL HPC clusters. The HPC system’s energy efficiency increased by nearly ten times between Jaguar and Titan. The usage of GPUs was the main distinction between these two designs. However, Titan continued to consume far more energy than was intended for Exascale. By more effectively utilizing the GPU technology in Frontier, they were able to accomplish their goal. Starting in 2012 with Titan and continuing today with Frontier, ORNL was a pioneer in the use of GPUs for supercomputing. When it comes to high performance with low energy usage, GPUs have proven their mettle as accelerators.
2.2 GPU

Modern mainstream computing systems now come standard with a Graphics Processing Unit (GPU). GPUs’ performance and functionalities have significantly improved over time. In addition to being a powerful graphics engine, modern GPUs are also highly parallel programmable processors with peak computation and memory bandwidth that significantly outperform their CPU counterparts. The GPU’s rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU. This effort in general-purpose computing on the GPU has established the GPU as a compelling alternative to traditional microprocessors in future high-performance computer systems.

2.2.1 Graphics before GPUs - a brief history

Long before the first graphic card was invented, graphics were used with computer hardware. The first computer graphic was used on a computer in the 1940s, when the Massachusetts Institute of Technology developed the Whirlwind-I, a flight simulator that could locate objects for the United States Navy. In 1965, IBM released their first graphic display machine, the IBM 2250, which displayed vector graphics drawn on a screen with a light pen. Around the same time, MIT graduate Dr. Ivan Sutherland developed Sketchpad, a software program that drew images on a computer using a TX-2 computer with a monitor and a light pen.

Raster and vector graphics are used on a computer to display images. Raster graphics uses pixels, small dots that attach to a bitmap grid. Each pixel is placed on a specific location based on the drawing. If the graphic is a line, it will appear smooth. But once the graphic is enlarged, it appears distorted or fuzzy. Vector graphics uses a point-to-point math scale. Using a math scale, vector graphics have a defined path for points, lines and curves. The images are much clearer, even when enlarged.

In the past, cathode ray tubes were used to display graphics and looked like a TV set. In the 1980s, IBM was the first company to develop the graphics card. The monochrome display adapter and the color graphics adapter were plug-in devices that attached to the CRT. The MDA card had 4 kilobytes of memory that could handle 720 by 350 pixels and could display 25
rows of 80 characters. The CGA had 16 kilobytes of memory and 160 by 200 pixels. The CGS had two types of resolution for text and three types for graphics. IBM would later develop the Enhanced Graphics Adapter, the Video Graphics Array, Extended Graphics Array, Ultra Extended Graphics Array and the Super Video Graphics Array.

In the 1990s, IBM developed the Extended Video Graphics Array, still in use today with monitors and projectors. The EVGA has 1024 by 768 pixels and displays 256 colors. Intel has developed the Accelerated Graphics Port Digital Display card. When the adapter card is attached to the Intel 865G chip, it can be used as an output device for televisions, digital displays and monitors. The ARG adapter has a resolution of 2048 by 1536 pixels and 16.7 million colors.

Today, graphic cards are not being made only for computer use. Companies like Nvidia and ATI are making 2-dimensional and 3-dimensional graphic cards for gaming. Video consoles like PlayStation and GameBoy use these graphic cards for video display.

2.3 Programming GPUs

Although GPU programming has only been practically feasible for the past two decades, it is now used in almost every industry. While a CPU only has a few cores that are optimized for sequential serial processing, a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed to handle numerous tasks at once. While some tasks, like finding a word in a document, benefit greatly from parallel processing, others, like computing the Fibonacci sequence, do not. GPUs are highly suitable for some tasks because of their capacity to handle multiple tasks at once. However, deep learning, one of the most in-demand skills in technology today, is one of the tasks that does significantly benefit from parallel processing. Machines are now able to learn how to comprehend language [141, 87], recognize patterns [129, 21, 1], and even create music [54, 14] thanks to deep learning algorithms. The demand for developers who comprehend general-purpose computing on a GPU has been growing as a result of the increasing significance of artificial intelligence.

Early attempts to use GPUs as general-purpose processors required reformulating computational problems in the language of graphics cards because GPUs understand computational problems in terms of graphics primi-
tives. Fortunately, GPU-accelerated computing is now much simpler thanks to parallel computing frameworks like Nvidia’s CUDA, OpenCL, OpenACC or OpenMP. These platforms enable programmers to put less emphasis on the language barriers between the CPU and the GPU and more emphasis on higher-level computing concepts. Let’s quickly review each of these programming languages with an example Code 2.1 shows a C-code for naive serial matrix multiplication. We will compare other programming languages with this code as we discuss how they can assist in executing it on a GPU.

### 2.3.1 CUDA

Initially released by Nvidia in 2007, CUDA (Compute Unified Device Architecture) is the dominant proprietary framework today. One way to develop extremely parallel applications with CUDA is through CUDA C++. It enables the user to create high performance algorithms using the robust C++ programming language, which are then supported by thousands of parallel GPU-powered threads. Many programmers have used this technique to speed up their computationally and bandwidth-intensive applications, such as the libraries and frameworks that support the current artificial intelligence revolution.

Code 2.2 demonstrates how a programmer can use C++ to create the basic CUDA matrix multiplication code. When compared to Code 2.1, the
differences are very obvious. Before even starting the kernel, one would need to set up the runtime configurations, such as how many grid blocks and threads per block are required, allocate memory on the device, and transfer data from the host to the device. When the kernel execution is finished, they will need to return the data to the CPU and release the GPU’s memory.

```c
void multiply_cuda(float *A, float *B, float *C, int N)
{
    float *d_A;
    float *d_B;
    float *d_C;

    int size = N * N * sizeof(float);

    // Setup runtime configuration
    dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 blocksPerGrid(N / BLOCK_SIZE, N / BLOCK_SIZE);

    // Allocate Device memory
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy memory from host to device
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    // Launch kernel
    matMult<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);

    // Copy memory from device to host
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free memory on device
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
/* CUDA kernel */
__global__ void matMult(float *A, float *B, float *C, int N) {
    int ROW = blockIdx.y * blockDim.y + threadIdx.y;
    int COL = blockIdx.x * blockDim.x + threadIdx.x;

    if (ROW < N && COL < N) {
        float sum = 0.0f;
        for (int i = 0; i < N; i++) {
            sum += A[ROW * N + i] * B[i * N + COL];
        }
        C[ROW * N + COL] = sum;
    }
}

Code 2.2: Matrix Multiplication using CUDA

Quite challenging, but manageable, right? However, an application might have hundreds of different kernels, and we must follow these procedures for each kernel. Furthermore, CUDA only functions with NVIDIA GPUs. It is not compatible with any other GPUs. This renders CUDA-written code on other platforms useless.

2.3.2 OpenCL

The most popular open, royalty-free standard for cross-platform, parallel programming is OpenCL, which was first introduced by the Khronos Group in 2009. As of now, OpenCL has been adopted by Altera, AMD, Apple, ARM, Creative, IBM, Imagination, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS. It is incredibly versatile and supports all widely used operating systems on all major platforms. Although OpenCL specifies a C-like language for program creation, other programming languages and platforms, such as Python or Java, have third-party APIs.

But writing a program in OpenCL is much more convoluted. Code 2.3 shows how a programmer can use OpenCL to adapt the same simple matrix multiplication code to run on a GPU. Although it is a matter of preference, we realize that programming in OpenCL is far too difficult when compared to Code 2.1 or Code 2.2. Before the kernel can be launched, the same naive implementation of matrix multiplication requires a significant amount of pre-processing. This is definitely a deterrent for most application scientists who
want to offload their code to a GPU.

```c

void multiply_opencl(float *A, float *B, float *C, int N)
{
    int err;            // Error code from API calls
    cl_device_id devID; // Compute device id
    cl_context context; // Compute context
    cl_command_queue commands; // Compute command queue
    cl_program program; // Compute program
    cl_kernel kernel; // Compute kernel
    cl_event event = NULL; // Compute Event

    // OpenCL device memory for matrices
    cl_mem d_A;
    cl_mem d_B;
    cl_mem d_C;

    int size = N * N * sizeof(float);
    cl_mem_flags copy_to = CL_MEM_READ_WRITE |
                             CL_MEM_COPY_HOST_PTR;
    cl_mem_flags copy_from = CL_MEM_READ_WRITE;
    const int TS = 32;
    const size_t local[2] = { TS, TS };  
    const size_t global[2] = { N, N };  

    // Initialize OpenCL device
    cl_uint dev_cnt = 0;
    clGetPlatformIDs(0, 0, &dev_cnt);
    cl_platform_id platform_ids[dev_cnt];
    clGetPlatformIDs(dev_cnt, platform_ids, NULL);

    // Connect to a compute device
    err = clGetDeviceIDs(platform_ids[0], CL_DEVICE_TYPE_GPU, 1, &devID, NULL);

    // Create a compute context
    context = clCreateContext(0, 1, &devID, NULL, NULL, &err);
```
// Create a command commands
commands = clCreateCommandQueue(context, devID, 0, &err);

// Create a program from the kernel source
program = clCreateProgramWithSource(context, 1,
    (const char **)&matMult, NULL, &err);

// Build the program executable
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create device memory for our calculation
d_A = clCreateBuffer(context, copy_to, size, A, &err);
d_B = clCreateBuffer(context, copy_to, size, B, &err);
d_C = clCreateBuffer(context, copy_from, size, NULL, &err);

// Create the compute kernel in the program we wish to run
kernel = clCreateKernel(program, "matMult", &err)
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_A);
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&d_B);
err = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&d_C);
err = clSetKernelArg(kernel, 3, sizeof(int), (void*)&N);

// Launch Kernel
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL,
    global, local, 0, NULL, &event);
err = clWaitForEvents(1, &event);
err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, mem_size_C,
    h_C, 0, NULL, NULL);

// Cleanup
c1ReleaseMemObject(d_A);
c1ReleaseMemObject(d_C);
c1ReleaseMemObject(d_B);

c1ReleaseProgram(program);
c1ReleaseKernel(kernel);
c1ReleaseCommandQueue(commands);
c1ReleaseContext(context);
/* OpenCL kernel */
__kernel void matMult(__global float* A, __global float* B,
    __global float* C, int N)
{
    int ROW = get_global_id(0);
    int COL = get_global_id(1);

    if(ROW < N && COL < N) {
        float sum = 0.0f;
        for (int i = 0; i < N; i++) {
            sum += A[ROW * N + i] * B[i * N + COL];
        }
        C[ROW * N + COL] = sum;
    }
}

Code 2.3: Matrix Multiplication using OpenCL

The decision between these two parallel computing platforms depends on the user’s objectives and working environment. For instance, CUDA is frequently used in academia and is regarded as the most straightforward to learn. Although OpenCL is by far the most portable parallel computing platform, each target platform still requires a customized optimization of OpenCL programs. This is where directive programming languages like OpenACC or OpenMP comes to the rescue.

2.3.3 OpenACC

To simplify parallel programming of heterogeneous CPU/GPU systems, a group of companies led by Cray, CAPS, NVIDIA, and PGI (the Portland Group) released OpenACC in 2011 [150]. According to OpenACC’s official website [106] — “OpenACC is a user-driven directive-based performance-portable parallel programming model. It is designed for scientists and engineers interested in porting their codes to a wide-variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model.” The OpenACC API is largely convincing in terms of productivity and programability. OpenACC enthusiasts can mark up C, C++, and Fortran source code to indicate to the
void multiply_openacc(float *A, float *B, float *C, int N)
{
    #pragma acc kernels copyin(A[0:N*N], B[0:N*N]) copy(C[0:N*N])
    {
        #pragma acc loop independent
        for(int i = 0; i < N; i++) {
            #pragma acc loop independent
            for(int j = 0; j < N; j++) {
                float sum = 0.0f;
                #pragma acc loop independent reduction(+: sum)
                for(int k = 0; k < N; k++) {
                    sum += A[i * N + k] * B[k * N + j];
                }
                C[i * N + j] = sum;
            }
        }
    }
}

Code 2.4: OpenACC Matrix Multiplication

GPU which areas should be accelerated. The objective is to offer a programming model for accelerators that is adaptable to different operating systems, host CPUs, and accelerators.

The parallel directives in OpenACC tell the compiler that the loop is one that runs in parallel. How the loop is parallelized is up to the compiler. The compiler, for instance, can produce code to run iterations across threads or across SIMD lanes. The method of parallelization will be decided by the compiler based on the underlying hardware architecture, or a combination of methods may be used. The ease with which the matrix multiplication code can be offloaded to the GPU using OpenACC is evident in Code 2.4. When compared to CUDA (Code 2.2) and OpenCL (Code 2.3) codes, it is clear why an application developer would be interested in OpenACC.

2.3.4 OpenMP

As the de facto directive based parallel programming model, OpenMP has been adopted extensively in the supercomputing world and gains more and
more attention in general purpose computing [27]. Using OpenMP directives and its APIs, several parallel patterns such as loop level and task based parallelism can be expressed. Thus, OpenMP facilitates CPU parallel programming in shared memory systems significantly.

![Figure 2.3: OpenMP Parallel Threads](image)

All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until it encounters the first parallel region construct. The master thread then forms a group of parallel threads (Figure 2.3). Thread management is implicit. Special directives are used to specify that a section of code should be run in parallel. The number of threads to be used is specified using an out-of-band mechanism that is an environment variable. Thus, developers are not required to manage thread lifetimes.

Workload partitioning and task-to-worker mapping require a relatively low programming effort. Developers only need to specify compiler directives to denote a parallel region. OpenMP also abstracts away how the workload (e.g., an array) is divided into tasks (e.g., sub-arrays) and how tasks are assigned to threads. It supports several constructs to support implicit synchronization, where developers specify only where synchronization occurs. The actual synchronization mechanism is thus not the responsibility of the developers.

Instead of CPUs, today’s systems rely more on hardware accelerators like GPUs to achieve performance goals. As the current most popular accelerator, the massive multi-threading ability of GPUs can especially benefit applications with large amount of parallelism, such as scientific computations and machine learning. Therefore, GPUs will remain a crucial component in supercomputing systems in the foreseeable future. For instance, the Summit supercomputer was equipped with 6 NVIDIA Volta GPUs on each node.
/* Multiply Matrices $C[N \times N] = A[N \times N] \times B[N \times N]$ */

void multiply_openmp(float *A, float *B, float *C, int N)
{
    #pragma omp target data map(to:A[0:N*N],B[0:N*N]) map(C[0:N*N])
    {
        #pragma omp target teams distribute collapse(2)
        for(int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++) {
                float sum = 0.0f;
                #pragma omp parallel for reduction(+: sum)
                for(int k = 0; k < N; k++) {
                    sum += A[i * N + k] * B[k * N + j];
                }
                C[i * N + j] = sum;
            }
        }
    }
}

Code 2.5: OpenMP Matrix Multiplication

[109].

In order to leverage accelerators like GPUs, OpenMP 4.0 introduced the ability to offload computations to accelerators [107]. It is called device of-

Figure 2.4: Growing Relevance of OpenMP: NERSC

21
Offloading. The offloading devices can include GPUs, FPGAs, DSPs, etc. GPU offloading is arguably the most important one among them currently.

Programming with OpenMP (Code 2.5) is just as simple as programming with OpenACC (Code 2.4). Compared to other GPU programming models such as CUDA (Code 2.2) and OpenCL (Code 2.3), using OpenMP for GPU programming has a shorter learning curve for users and is more performance portable. Compared to other directive based methods like OpenACC, OpenMP has a broader user community and is more widely available. According to the NERSC report from 2015 [49], OpenMP would serve as the primary on-node programming interface for their upcoming platforms. Figure 2.4 demonstrates that almost 50% of all NERSC apps used OpenMP. By 2016, this number had increased to 75%. Therefore, we expect the number of OpenMP+GPU users to continue to grow.

2.4 Compilers - LLVM

The LLVM Project is a collection of reusable and modular toolchain and compiler technologies. Originally named as an acronym “Low Level Virtual Machine”, has developed over the last two decades from an academic project to the standard back-end of compilers for C, C++, Objective C, etc. The name LLVM itself has stopped being an acronym; it is now the full name of the project, and it no longer has much to do with conventional virtual machines. LLVM is an umbrella project that houses and creates a number of low-level toolchain components (like assemblers, compilers, debuggers, etc). The primary characteristic that distinguishes LLVM from other compilers is its internal architecture, despite the fact that it offers some special capabilities and is well-known for some of its popular tools (such as the Clang compiler).

The most advantageous aspect of the LLVM architecture is when a compiler chooses to support various source languages or target architectures. As shown in Figure 2.5, if the compiler’s optimizer uses a common code representation, any language that can compile to it can have a front end, and any target that can compile from it can have a back end. With this design, a new front end must be implemented in order to port the compiler to support a new source language, but the back end and optimizer can be reused. The same holds true for creating a new architecture’s back end. The fact that different skills are needed to implement a front end versus an optimizer and
back end is another significant advantage of this design. It is simpler for a front-end worker to improve and maintain their portion of the compiler when these are separated.

### 2.4.1 Clang

For languages in the C language family, such as C, C++, etc., the Clang project offers a language front-end and tooling infrastructure for LLVM. Clang has a fantastic modular design that makes it simple to use and can parse and analyze any source code in the C language family. It has a decent documentation and is very helpful for performing static analysis.

### 2.4.2 How does Clang work?

Clang serves as both a tool development platform and a compiler for the C language family. Because of its library-based architecture, reusing and integrating new features is more flexible and simple to integrate into other projects. Clang compiler, like many other compilers, has three phases (Figure 2.6):

- **The Frontend** – that parses source code, checks for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code.

- **The Optimizer** – its goal is to do some optimization on the AST generated by the front end.
• **The Backend** — that generate the final code to be executed by the machine.

![LLVM Compiler Design](image)

LLVM handles Clang’s optimizer and back end. Clang will typically execute the *preprocessor* (expanding all macros) and convert the source code into an *Abstract Syntax Tree* (AST). Although working with the preprocessed AST is much simpler than working with source-level C code, we can always easily refer to the original code. In fact, every Clang data structure (AST, CFG, etc.) that represents the code can always be linked back to the original source, which is very helpful for a variety of analysis tasks (refactoring, etc). Clang outperforms LLVM when it comes to analyzing and modifying code at the source level. When performing analysis with LLVM, we must use the internal representation of the code provided by LLVM, which is similar to assembly language.

### 2.4.3 Abstract Syntax Tree (AST)

An Abstract Syntax Tree [114], or AST, is used to represent the source code by almost all compilers and static analysis tools. In contrast to ASTs created by some other compilers, Clang’s AST closely resembles both the written C++ code and the C++ standard. For instance, the AST provides unreduced forms of parenthesis expressions and compile time constants. As a result, refactoring tools work well with Clang’s AST.

The simplest way to understand it is to simply dump the AST for a straightforward source file to see how it is organized. A Clang AST typically consists of two extremely flexible Classes: `Decl` and `Stmt`. Each has a large number of subclasses; here are a few examples:

1. **FunctionDecl** — Represents a function declaration or definition.
2. **BinaryOperator** — A builtin binary operation expression such as “x + y” or “x <= y”.

24
3. **CallExpr** – Represents a function call, such as \( \text{foo}(x, 2) \).

Majority of the AST classes (such as `ForStmt`, `IfStmt`, `ReturnStmt`) are self-explanatory and simple enough that one can quickly pick them up after experimenting with them for a short while. When in doubt, we could refer to the Clang’s documentation for a thorough explanation of its internal code.

### 2.4.4 How to use Clang?

Clang has some cool built-in static analysis tools and can be used as a drop-in replacement for compilers like gcc. Depending on how we want to program, we can use Clang as a library in one of three ways, giving us as programmers access to all of its capabilities:

1. **LibClang** – A reliable high level C interface to clang is LibClang. If in doubt, one should probably use the LibClang interface. Only when there is a compelling reason not to use LibClang do we consider the other interfaces. Periodically, Clang changes, and if we use a Plugin or Libtooling, we may need to update our code to reflect those changes. We must use LibClang to access Clang’s API from a language other than C++ (such as Python). But Plugins and LibTooling are better options if we want total control over the AST.

2. **Clang Plugin** – The use of Clang Plugins enables the execution of additional user-defined operations during compilation. We cannot maintain any global information or other contextual information across different source files because our code is the plugin itself and is run as a completely new instance for each source file (but we can still run it on multiple files sequentially). By providing command-line arguments to our build system (Clang, Make, etc.), a plugin can be launched. It resembles turning on an optimization in GCC (e.g., "-O1"). No custom task can be executed either before or after a source file is analyzed. LibTooling is a better option if we want complete control over Clang’s configuration.

3. **LibTooling (Clang Tool)** – In LibTooling, our code is a typical C++ program, and the main() function serves as the program’s entry point. LibTooling is typically used to perform analysis on a single file of source code (or multiple files, if we chose so) outside of our standard
build process. Similar to a Clang Plugin, a new instance of our analysis code (and a new AST) will be created for every new source file, but we are able to preserve contextual information across each source file because persistent data elements like global variables. We can also execute tasks before or after Clang has finished analyzing all of our source files because we have a main() function.

2.5 ML in Compilers

A number of frameworks [90, 95, 116] are being developed by compiler engineers to help application developers deal with extreme heterogeneity more effectively. These frameworks require analytical cost models to assist them in making better decisions when selecting the appropriate optimization or transformation for the application. However, developing a cost function is time-consuming, and almost all modern compilers, including LLVM/Clang, use a simple “one-size-fits-all” cost function that does not provide the best performance in the case of diverse architecture. Hand-tuned cost functions are currently popular, but calculating the costs and benefits of a compiler optimization requires a deeper understanding of the underlying hardware. Despite its effectiveness, manually constructing a cost model for a single architecture can take months. Compiler engineers are looking into Machine Learning (ML) techniques as a way to automate this process because cost functions are important and manual tuning is quite time-consuming.

The idea of machine learning in compilers is not new; there are papers on the topic from over two decades ago. But historically, and for good reasons, compiler developers have been hesitant to incorporate machine learning techniques into compilation. Compiler have two jobs – translation and optimisation. They must first translate programs into binary correctly. Secondly they have to find the most efficient translation possible. We are aware that no one will use a compiler that produces efficient code if the developer claims that, while it will produce accurate results 9 out of 10 times, there may be a single instance in which it does not provide the proper transformation. On the other hand, if a machine learning developer claims that their model is 90% accurate, it will typically be viewed as a huge success. The most important component of a compiler is correctness, and a compiler developer cannot compromise on it. Therefore, it makes sense that we were hesitant to apply machine learning to compilation.
But recently, a lot of research has been done on how to use ML techniques in compilation as well. Early work exploiting ML in compilers, like [100], primarily explored its use to help optimize sequential programs. However, with the proliferation of multi-core platforms and, more recently, heterogeneous systems, its application to the task of optimizing parallel programs has received significant attention in the last decade. A decision-tree-based approach has been developed to predict the scheduling policy for an OpenMP parallel region [93]. Furthermore, [31] optimizes OpenMP programs for scheduling policies and thread count using ML techniques. ML has been used to determine the optimum degree of parallelism for transactional memory [153] and hardware resource allocation [64]. The work presented in [9] and [67] applies ML to complex parallel programs and divide them among the available multi-core resources. The Petabricks project [4] employs a genetic search to tune algorithmic choices at compile time to reduce searching overheads. Malik et al. [86], who present a novel graph-based approach for feature representation, have also used tree and graph-based features. ML techniques were used to build classifiers to determine whether to offload OpenCL code [33], and to select a clock frequency at which the processor should operate [61]. Although the work was said to be extremely accurate, no modified code was produced, so the benefits could not be calculated. The outcomes of earlier work using ML on compiler optimizations are promising. New feature engineering techniques that can assist ML in understanding a code and its computational requirements must be looked into.

So, what do all of these studies have in common? They are using ML to select a transformation for their code that has already been shown to be accurate, ensuring that accuracy is not compromised. In a similar vein, we create COMPOFF [94], a first-of-its-kind compiler cost model that uses a neural network model to predict how long an OpenMP kernel will take to execute on a GPU. More details on COMPOFF can be found in Chapter 7.
Chapter 3

Motivation

Motivation can come from anywhere. Sometimes a clearly defined problem is unable to inspire us, and other times a minor challenge shows up and motivates us to make every effort to solve it. This work was motivated by the first project I worked on at the start of my research. The goal of the project was to investigate the unified memory aspect of GPU offloading and look into ways to incorporate it into OpenMP. It turns out that GPU offloading is much more complicated than it seems, even when using OpenMP.

Despite the fact that the latest OpenMP standard includes device offloading capabilities to help with GPU programming, there are still many obstacles to overcome. One of these is the unified virtual memory\(^1\) feature introduced in recent GPUs. GPUs in current and future HPC systems have enhanced support for unified memory space. In such systems, CPU and GPU can access each other’s memory transparently, that is, the data movement is managed automatically by the underlying system software and hardware. Memory over subscription is also possible in these systems. However, there is a significant lack of knowledge about how this mechanism will perform, and how programmers should use it.

We modified a number of benchmarking codes in the Rodinia benchmark suite to address this gap and investigate the behavior of OpenMP accelerator extensions, for example, by using them to investigate the effects of unified memory in an OpenMP context. Additionally, we changed the free and open-source LLVM compiler to support OpenMP programs using unified memory. Our evaluation’s results indicate that while unified memory per-

\(^{1}\text{We use terms unified virtual memory, UVM, and unified memory interchangeably in this research.}\)
forms similarly to standard GPU offloading for benchmarks with little data reuse, it incurs a significant overhead when GPU memory is oversubscribed for benchmarks with a lot of data reuse.

Based on these findings, we give programmers a number of recommendations for using unified memory to improve performance. But even so, we also became aware of how challenging GPU programming can be, even when using the OpenMP programming language.

3.1 Introduction

As the de facto directive based parallel programming model, OpenMP has been adopted extensively in the supercomputing world and gains more and more attention in general purpose computing. Using OpenMP directives and its APIs, several parallel patterns such as loop level and task based parallelism can be expressed. Thus, OpenMP facilitates CPU parallel programming in shared memory systems significantly.

Instead of CPUs, today’s systems rely more on hardware accelerators like GPUs to achieve performance goals. As the current most popular accelerator, the massive multi-threading ability of GPUs can especially benefit applications with large amount of parallelism, such as scientific computations and machine learning. Therefore, GPUs will remain a crucial component in supercomputing systems in the foreseeable future. For instance, the next generation supercomputer in ORNL, Summit, will be equipped with 6 NVIDIA Volta GPUs on each node [109].

In order to leverage accelerators like GPUs, OpenMP 4.0 introduced the ability to offload computations to accelerators [107]. It is called device offloading. The offloading devices can include GPUs, FPGAs, DSPs, etc. GPU offloading is arguably the most important one among them currently. Compared to native GPU programming models such as CUDA [103] and OpenCL [130], using OpenMP for GPU programming has a shorter learning curve for users and is more performance portable. Compared to other directive based methods, like OpenACC [105], OpenMP has a broader user community and is more widely available. Therefore, we expect the number of OpenMP+GPU users to continue to grow.

Despite this, many challenges remain to be addressed to make OpenMP GPU offloading performance competent. One of the most important ones is the lack of support for unified virtual memory (UVM). Normally, CPU and
GPU have separate memory. Before the introduction of unified memory, host and devices would use separate memory space. As a result, CPU and GPU could not access each other’s memory, and communication between CPU and its devices had to be managed by programmers explicitly.

Now, with unified memory, a single memory space shared by both CPU and GPU is used to unify the previously separated memory spaces. This permits the host to refer to memory locations on the attached devices, and the devices to access addresses on their host. In addition, the data movement is managed by the underlying system software automatically. Unified memory on pre-Pascal GPUs behaves identical to cases when not using it: all data needs to be transferred to GPU before execution. The only difference is that the programmer is not required to handle it explicitly, rather this is managed automatically by the software. On the other hand, Pascal and later GPUs support on-demand page migration, which is a big step forward, and brings two major advantages.

First, the copying process of complex data structures is greatly simplified; there is no need for address translation, and CPU and GPU can use the same pointer. Second, leveraging the unified memory feature makes it feasible to run kernels with memory footprints larger than the GPU memory capacity. Without on demand page migration, GPU offloading is possible only when dataset fits into GPU memory. While with it, part of data can reside in CPU memory, and the GPU will fetch them to its own memory when they are actually required in the computation. These advantages, promote the use of unified memory by using OpenMP as a vehicle to ease the process of GPU programming.

However, there is a significant lack of knowledge regarding how unified memory performs and how programming models should use it, especially with the on-demand page migration support in Pascal and later GPUs\(^2\). To the best of our knowledge, there is no comprehensive performance study for it yet. Programmers tend to think unified memory can deliver similar performance compared with cases when it is not used, and no extra work needs to be done. However, as we will show in Section 3.3, straightforward unified memory usage can result in significant performance degradation without careful coding and tuning. Therefore, we believe a comprehensive study of unified memory is in great need and critical.

\(^2\)In the rest of this paper, unified memory means unified memory with on demand page migration support unless otherwise stated.
In this paper, we develop a set of benchmarks and use them to study the performance of unified memory. Our benchmarks have been ported from the Rodinia benchmark suite [19], which is widely used in GPU performance evaluation. Since there are very few public available benchmarks for OpenMP GPU offloading, we first develop a set of OpenMP GPU offloading benchmarks. We also plan to make these benchmarks publicly available later. Upon them, we further add unified memory support by modifying the LLVM compiler [75] and using OpenMP directives/ clauses in various ways. Finally, we conduct a performance study on these benchmarks.

The contributions of this work are the following:

• We develop a set of benchmarks for OpenMP GPU offloading both using and not using unified memory. Using these benchmarks, we analyze and identify the inefficiency existing in the default unified memory management policy. These benchmarks also demonstrate that using unified memory can simplify OpenMP GPU programming significantly, especially when transferring complex data structures is needed.

• To the best of our knowledge, we conduct the first comprehensive performance study for unified memory. We show cases when unified memory performs well and when it does not. We believe the findings in this paper can help programmers make better use of unified memory capabilities.

• We introduce a light-weight method to use unified memory within the current OpenMP framework, with negligible modifications to the OpenMP runtime.

• Based on the analysis results, we provide several programming guidelines for unified memory. Users can improve the performance of their applications following these guidelines when using unified memory.

The rest of this paper is organized as follows: Section 3.2 presents the developed benchmarks for OpenMP GPU offloading and unified memory. Section 3.3 evaluates the performance using the ported benchmarks. In Section 3.4, we describe how to improve the performance of unified memory. Section 3.5 discusses some related work. Finally, Section 3.6 consists of the conclusion of this paper.
3.2 Benchmark Development

Although there are plenty of OpenMP benchmark suites publicly available, little effort has been put into OpenMP GPU offloading benchmarking since it is relatively new in OpenMP. To the best of our knowledge, there are very few public available benchmarks for OpenMP GPU offloading.

As the first step, we develop a set of benchmarks that use OpenMP GPU offloading. We choose the OpenMP benchmark subset of the Rodinia benchmark suite [19] as our baseline, and extend it to make use of GPU offloading. Once we complete the porting the complete suite we will release our code and try to make it an official part of Rodinia.

Rodinia was originally developed by Che et al. at University of Virginia to benchmark GPUs. Since its first release, many people have made contributions to Rodinia, and it is still evolving. Rodinia is one of the most widely used benchmark suites for GPU benchmarking nowadays.

Rodinia has several advantages over other GPU benchmark suites such as Parboil [131], SHOC [28], and PolyBench/ACC [43]. First, benchmarks exhibit more irregular behavior, and thus are more challenging to implement and optimize for GPU execution. Next, Rodinia consists of a wider class of benchmarks compared to others. It covers the spectrum of scientific simulation, machine learning, graph processing, data processing, linear algebra, etc. Moreover, Rodinia benchmarks are written in multiple programming models, including CUDA, OpenCL, and OpenMP, which makes it relatively easy to extend. Note that the current OpenMP version in Rodinia is for CPU and does not include GPU offloading.

By default, the current OpenMP implementation does not allocate data in unified memory. Users need to explicitly specify how to map data to GPU, and OpenMP runtime will use the provided information to manage data transfers between host and device. Therefore extra effort needs to be made to investigate the unified memory performance for OpenMP GPU offloading. In Section 3.2.2, we discuss the modifications done to the OpenMP implementation in the LLVM framework in order to provide mechanisms that allow using unified memory allocation. We also exploit existing OpenMP offloading features to prevent the OpenMP runtime from moving data explicitly. Table 3.1 summarizes the subset of benchmarks used in the paper.

---

3At the time this paper was written, OpenMP 4.5 benchmarks like SPEC were still being developed.
<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back Propagation (Backprop)</td>
<td>Pattern Recognition</td>
<td>Back Propagation is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network.</td>
</tr>
<tr>
<td>Breadth First Search (BFS)</td>
<td>Graph Algorithms</td>
<td>Graph algorithms are fundamental and widely used in many disciplines and application areas. Large graphs involving millions of vertices are common in scientific and engineering applications. Breadth First Search traverses all the connected components in a graph.</td>
</tr>
<tr>
<td>CFD Solver (CFD)</td>
<td>Fluid Dynamics</td>
<td>The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow.</td>
</tr>
<tr>
<td>K-means</td>
<td>Data Mining</td>
<td>K-means is a clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity. Many data-mining algorithms show a high degree of data parallelism.</td>
</tr>
<tr>
<td>k-Nearest Neighbors (NN)</td>
<td>Data Mining</td>
<td>Nearest Neighbor finds the k-nearest neighbors from an unstructured data set. The sequential NN algorithm reads in one record at a time, calculates the Euclidean distance from the target latitude and longitude, and evaluates the k nearest neighbors. The parallel versions read in many records at a time, execute the distance calculation on multiple threads, and the master thread updates the list of nearest neighbors.</td>
</tr>
<tr>
<td>Speckle Reducing Anisotropic Diffusion (SRAD)</td>
<td>Image Processing</td>
<td>SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features.</td>
</tr>
</tbody>
</table>

Table 3.1: Rodinia Benchmark details

---

33
3.2.1 Benchmarking OpenMP GPU Offloading

OpenMP device offloading includes two essential parts:

- mapping data between host (i.e., CPU) and device (GPU in this paper) since they cannot address each other’s memory by default, and
- offloading computations from host to device.

In this subsection, we discuss how these two parts work.

![Diagram of traditional OpenMP GPU offloading without unified memory.](image)

**Figure 3.1:** Traditional OpenMP GPU offloading without unified memory.

**Data Mapping.** Figure 3.1 shows how GPU offloading generally works in OpenMP. Suppose we have two 2D arrays A and B and perform matrix addition on them to create array C. Initially all the data is located in the CPU memory. Recall that the GPU cannot access these data without the support of unified memory. In order for the GPU to access host data, OpenMP needs some mechanisms to facilitate GPU data allocation and transfer between the CPU and the GPU. This is achieved by using the `target data` directives and `map` clauses which instruct the OpenMP runtime to allocate the corresponding data copy in the device, followed by a synchronization between the copies of the host and device. The C code for the example in Figure 3.1 is given in Code 3.1.

In this code segment, the `map` clauses instruct OpenMP to copy the values of array A and B from the CPU to GPU at the start of the target data region with the keyword `to`. It also instructs to copy array C from GPU to CPU.
```c
#pragma omp target data map(to: A[0:N][0:M], B[0:N][0:M]) \  
           map(from: C[0:N][0:M])
{
    #pragma omp target teams distribute
    for (int i = 0; i < N; i++) {
        #pragma omp parallel for
        for (int j = 0; j < M; j++)
            C[i][j] = A[i][j] + B[i][j];
    }
}
```

Code 3.1: OpenMP GPU offloading code example.

at the end of the target data region with the keyword from. Note that the keyword to/from can be used to enforce data synchronizations both at the beginning and at the end of the data region (i.e., when copying-in and copying-out).

**Computation Offloading.** Once the data mapping has completed, control is transferred to the GPU. This is done by using the target directive as shown in Code 3.1. Then, the computation is offloaded to the device when the data synchronization is finished. After this, the compiler packs the target region into a function, which is transferred to the target device, i.e., to the GPU. Upon termination of the offloaded kernel, the control returns to the CPU.

Unlike OpenMP’s flat threading model on CPUs, GPUs have a hierarchical thread organization: a GPU kernel is composed by multiple thread blocks, and each thread block consists of multiple threads\(^4\). Different thread blocks can execute in parallel, as well as different threads within the same thread block. To fully exploit this two level of parallelism, OpenMP introduces a new concept called team besides traditional thread. A team corresponds to a thread block in the GPU context, and it can include multiple threads, which correspond to threads within a GPU thread block.

At the beginning of a target region, only one team and one member thread is active. To have multiple teams, we use the directive teams distribute at first, which distributes the entire loop iteration space among all teams.

\(^4\)NVIDIA terminology is used in this paper. Thread block is called workgroup in AMD terminology.
Further, if there are additional nested parallel loops, we use the directive `parallel for` upon them to distribute the nested loop iterations among threads within a team as shown in Code 3.1. In scenarios when there is only one level of parallel loops, or when there is enough parallelism in the outer loop, we make use of the combined directive `teams distribute parallel for` to distribute the iteration space of one loop both among teams and threads within a team. Figure 3.2 illustrates a typical workload partition among teams and threads for the example of Code 3.1.

**Deep Copy Challenge.** During the process of implementing GPU offloading on our benchmarks, we found that the most challenging and error-prone part is to map complex data structures between the host and the device, many of which can make use of pointers to access different memory regions. This constitutes a strong case for the need of deep copy. Code 3.2 shows the main data structure `BPNN` in the Backprop benchmark. Besides scalar variables, it also includes many pointers that refer to the actual data storage of a neural network. It is not sufficient to just map an instance of `BPNN` to the GPU memory in this scenario, since the pointers stored in it are still pointing to the CPU memory, where the GPU cannot access.

To solve this problem, we need to explicitly map all the memory regions that the pointers within `BPNN` point to. We also need to create another in-
typedef struct {
    int input_n;
    int hidden_n;
    int output_n;
    float *input_units;
    float *hidden_units;
    float *output_units;
    float *hidden_delta;
    float *output_delta;
    float *target;
    float **input_weights;
    float **hidden_weights;
    float **input_prev_weights;
    float **hidden_prev_weights;
} BPNN;

Code 3.2: The BPNN structure in Back propagation benchmark.

3.2.2 Benchmarking Unified Memory

As shown in Section 3.2.1, in current OpenMP GPU offloading, map clauses are used to perform the GPU data allocation as well as the data movement between CPU/GPU. In such scenarios, it is the responsibility of the OpenMP runtime to allocate data in the GPU memory and to generate explicit data transfers to make sure the data is in a coherent state. However, when using unified memory, we would like the underlying system to manage these things automatically. Figure 3.3 illustrates how the function and data transfers work in the presence of unified memory. Therefore, in order for OpenMP to use unified memory, first, we need its runtime to relinquish the data movement
management.

To do this, `map` clauses should not be used in the presence of unified memory. To make things more complicated, the current implementation of the OpenMP runtime keeps records of all the GPU and CPU data and their relationship. If the `map` clause is not used for a pointer in the target region, the OpenMP runtime will assume that this pointer is the CPU copy and try to find the corresponding GPU copy in its records. This raises another problem since there is no such copy in the presence of unified memory, and CPU and GPU just refer to the same data. The approach we follow to address this issue, is to use `is_device_ptr` clauses to inform the OpenMP runtime that a particular pointer can in fact be used by the GPU.

The last problem encountered is that the current OpenMP implementation never allocates GPU data using unified memory allocation APIs. Therefore, we modify the OpenMP runtime in the LLVM framework to make it allocate data in unified memory. To be more specific, we modify the OpenMP runtime API `omp_target_alloc` so that it will use the unified memory allocator by default. We also try to pass the data allocated by the CUDA unified memory allocator `cudaMallocManaged` to the OpenMP target region directly. However, the latter method does not work because the data allocated by `cudaMallocManaged` is invisible to the OpenMP runtime.

Overall, to enable unified memory in OpenMP GPU offloading we use the modified `omp_target_alloc` for data allocation in unified memory and...
is_device_ptr clauses to pass them to target regions. The rest of the implementation remains unchanged, and works as the default GPU offloading introduced in Section 3.2.1. The C example code that shows how we use unified memory, where we assume A, B, and C are integer arrays is given in Code 3.3.

```c
A = omp_target_alloc(N*M*sizeof(int));
B = omp_target_alloc(N*M*sizeof(int));
C = omp_target_alloc(N*M*sizeof(int));
#pragma omp target data is_device_ptr(A, B, C)
{
    #pragma omp target teams distribute
    for (int i = 0; i < N; i++) {
        #pragma omp parallel for
        for (int j = 0; j < M; j++)
            C[i][j] = A[i][j] + B[i][j];
    }
}
```

Code 3.3: Unified memory code example.

Address Deep Copy with Unified Memory. As we discussed earlier, programming complex/deep data structure copy is time consuming and error-prone in cases where the data communication is managed explicitly by programmers. On the other hand, this job is much easier in the presence of unified memory. The reason behind this is that, with unified memory, both the GPU and the CPU can use the same address space to access memory regions, i.e., there is no need for pointer translation. For the Backprop example shown in Code 3.2. With this new mechanism, we can conveniently pass the structure pointer of BPNN to the GPU without further concerning about the mapping of pointers inside it.

3.3 Evaluation

3.3.1 Experimental Methodology

To evaluate the performance of our benchmarks, we use the next generation supercomputer prototype at ORNL, SummitDev. SummitDev has a total of 54 nodes. Each node is equipped with 2 POWER8 CPUs and 4 Tesla
P100 GPUs, connected through NVLink 1.0. Table 3.2 provides the detailed configuration of CPUs and GPUs in the compute node of SummitDev.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2 POWER8</td>
</tr>
<tr>
<td>Cores / Socket</td>
<td>10</td>
</tr>
<tr>
<td>Threads / Core</td>
<td>8</td>
</tr>
<tr>
<td>CPU Clock</td>
<td>2.0 GHz</td>
</tr>
<tr>
<td>Main Memory</td>
<td>256GB DDR4</td>
</tr>
<tr>
<td>GPU</td>
<td>4 Tesla P100</td>
</tr>
<tr>
<td>SMs / GPU</td>
<td>56</td>
</tr>
<tr>
<td>SM Clock</td>
<td>1328 MHz</td>
</tr>
<tr>
<td>Register File Size / SM</td>
<td>256 KB</td>
</tr>
<tr>
<td>FP32 CUDA Cores / SM</td>
<td>64</td>
</tr>
<tr>
<td>FP64 CUDA Cores / SM</td>
<td>32</td>
</tr>
<tr>
<td>GPU L2 Cache Size</td>
<td>4096 KB</td>
</tr>
<tr>
<td>GPU Memory</td>
<td>16GB HBM2</td>
</tr>
<tr>
<td>CPU &amp; GPU Interconnect</td>
<td>NVLink 1.0</td>
</tr>
</tbody>
</table>

Table 3.2: Compute node configuration on SummitDev.

We use Clang 4.0 to compile benchmarks for both CPU parallel computing and GPU offloading. To enable offloading for NVIDIA GPUs, along with -fopenmp we need to pass -fopenmp-targets=nvptx64-nvidia-cuda to Clang. All benchmarks are compiled under the O2 optimization level. Note that the Clang version used in our experiments includes enhancement to support unified memory. The Linux kernel version is 3.10.0 and the CUDA version is 8.0.61 on SummitDev.

We evaluate the performance of GPU offloading without unified memory (GPU w/o UM), GPU offloading with unified memory (GPU with UM), and the CPU version where all computation is performed by the CPU. We were not able to port K-means and Backprop to GPU without unified memory due to the complexity of their data structures. These two benchmarks require deep copy, which significantly increases the difficulty of normal GPU offloading as described in Section 3.2.1. Thus, K-means and Backprop only have the CPU version and the GPU with unified memory version. Nevertheless, these benchmarks will also be included in the final public suite once their porting is completed.

For each benchmark, we generate inputs of various sizes to study their
performance impact. For instance, the input sizes of BFS range from 0.2GB to 30GB, whereas for CFD it grows from 0.5GB to 22GB. We make sure the largest data size for each benchmark is always larger than the GPU memory capacity, so that we are able to study the performance impact of GPU memory over subscription. To evaluate the performance, we measure the execution time from the start till the end of the target data region. By doing this, the measured time captures both the GPU computation time and other overheads associated with OpenMP GPU offloading, such as data transfer time between CPU and GPU. We run each experiment 5 times and use the average execution time as the final result.

3.3.2 GPU Offloading without Unified Memory

Figure 3.4 shows the results of our evaluation. In this subsection, we focus on comparing the execution time of benchmarks when running on CPU vs GPU without unified memory. The results show that GPU offloading benefits 2 out of 4 benchmarks when the input size is smaller than that of the GPU memory. While BFS enjoys the most performance benefit from GPU offloading, which is on average 68.8% across different inputs, CFD shows a moderate gain from GPU offloading (21.7% improvement on average). The reason why BFS and CFD benefit from GPU offloading is that they are more computation bound and show a moderate amount of data reuse. Therefore, their performance is moderately improved by the GPU’s massive threading ability.

On the other hand, NN and SRAD do not benefit from OpenMP GPU offloading compared to their CPU counterparts. NN and SRAD suffer from an average performance degradation of 46.2% and 60.3% respectively. For these two benchmarks, the data transfer overhead outweighs the computing throughput benefit of GPU, because there is little data reuse and limited computation volume on each data item.

We also observe that the normal GPU offloading fails when the input size is comparable or larger than the GPU memory size. This is because in traditional GPU offloading, all the data must first be copied to the device before starting the kernel execution. When the required data exceeds the GPU’s memory capacity the data allocation/copy will fail. We observe the failure threshold is roughly 14GB. It is not possible to utilize the full memory capacity 16GB, GPU memory, since part of the memory space is reserved for thread private data and system usage. NN does not fail because its GPU memory requirement is constant and does not change with inputs. On the
Figure 3.4: Performance of CPU, GPU without unified memory, and GPU with unified memory.

other hand, the CPU version is able to deal with large inputs since
- the CPU memory is much larger (256GB per node on SummitDev)
- virtual memory system could exchange data between memory and disk dynamically if necessary.
3.3.3 Unified Memory

Figure 3.4 also shows the GPU offloading performance of different benchmarks with unified memory. These benchmark variants show significantly different performance depending on the program behavior. We distinguish two categories of benchmarks here, those with large amounts of data reuse (BFS, CFD, K-means and SRAD), and those with little to none data reuse (Backprop and NN). We discuss their performance separately in this subsection.

Benchmarks with Significant Data Reuse. For the first category of benchmarks, we observe that unified memory brings two major advantages. First, it helps improve performance compared with GPU offloading without unified memory when the working set fits into the GPU memory. For instance, unified memory outperforms traditional offloading by an average of 18% for BFS when the dataset fits into GPU memory. The reason why BFS enjoys the most performance improvement is that, in each iteration, BFS accesses a fraction of nodes. Then, the next iteration will access the neighbors of current accessed nodes until all nodes have been traversed. In such a scenario, unified memory does not bring all the data at the beginning. Instead, data transfers happen on demand during the entire traversal process and thus the data movement overhead is amortized. Without unified memory, all the data would have to be moved to the GPU before any GPU computation can start. Therefore unified memory achieves better performance in such cases.

The second advantage of unified memory is that it helps solve the problem of GPU memory over subscription, although the performance may be poor. BFS, CFD, and SRAD suffer from significant performance degradation when the data size exceeds the GPU memory capacity. For instance, when using 14GB inputs (i.e., GPU memory is not yet over subscribed), the performance gain of GPU w/ UM is 86% for BFS compared with CPU, while this is 16% for CFD. Then, as the input size increases, the GPU offloading crashes without unified memory, but the benchmarks are able to run correctly using unified memory, though with significantly degraded performance. Depending on the benchmark, the performance degradation varies. CFD suffers from the most severe degradation, whose execution time is 4.4x of that of the CPU version at the 22GB input.

To further analyze the performance results, especially the reasons behind the significant performance degradation when GPU memory is over subscribed, we breakdown the execution time and analyze its different com-
ponents. The potential overhead of unified memory includes data transfer from host to device, that from device to host, and page fault processing. The NVIDIA GPU profiler nvprof is used to collect this information.

Figure 3.5 shows the execution time breakdown of 4 benchmarks. The “Rest of Computation” is computed using the following formula.

\[
\text{Rest of Computation} = \text{total execution time} - \text{data transfer time} - \text{page fault processing time}
\]

Therefore it does not necessarily reflect the actual GPU computing time since some computation overlaps with data transfer and page fault processing. For instance, we observe that the “Rest of Computation” time decreases when input size exceeds that of the GPU memory for SRAD because of the overlapping. We were not able to collect these data for CFD because of the failure of nvprof. For benchmarks with significant data reuse, including BFS, K-means and SRAD, the results show that the data transfer time and page fault processing increases significantly when the GPU memory is over-subscribed. This is caused by the inefficiency of the default unified memory.
management policy, which is presumably an Least Recently Used (LRU) like policy.

![Graphs of data transfer breakdown for different benchmarks](image)

(a) BFS.  
(b) K-means.  
(c) SRAD.  
(d) NN.

Figure 3.6: Data transfer breakdown.

To show more details, Figure 3.6 illustrates the data transfer volume with unified memory for different benchmarks. Ideally, data transfers under unified memory should be near optimal because it only happens on demand which results in little redundant data movement. Unfortunately this is not true when the working set size exceeds that of the GPU memory. Because of data thrashing, both BFS and SRAD show a significant increment of data traffic from host to device and from device to host.

We use BFS as an example to explain the reasons of data thrashing for benchmarks with large amount of data reuse. There are two types of data reuse in BFS:

- a node may be traversed multiple times based on the structure of the input graph
- there is a lot spatial locality since neighbor nodes may get accessed in different iterations, resulting in page level reuse.

These reuse happens in a relative long distance, and when the total data
size is larger than that of GPU memory, it is likely that a data item will be evicted by the default LRU-like policy before it gets reused. This results in data thrashing: data are evicted before getting reused, and placing current accessed data in the GPU memory will further evict future reused data. As a result, little data locality is captured by the GPU memory under the LRU like management policy and the performance thus suffers \[118, 63, 79\].

Therefore, we conclude that reducing/eliminating data thrashing is critical to achieving better performance on benchmarks with significant data reuse. Improving unified memory management is an important means to achieve this goal.

**Benchmarks with Little Data Reuse.** For benchmarks such as Backprop and NN, the performance of unified memory is roughly proportional to the input size. It is also slightly outperformed by GPU offloading without unified memory. The reason behind this is that the data is effectively streamed (accessed without reuse). In such scenarios, the overhead associated with on demand data migration in unified memory becomes more dominant. As we observe in Figure 3.5 and 3.6, unified memory suffers from significant page fault processing overhead and on demand page movement overhead. The results for Backprop are similar. For these benchmarks, there is not enough data reuse to amortize the overhead compared to benchmarks with data reuse.

We also do not observe a large performance gap between inputs that fit in GPU memory and those that do not fit like those in BFS and CFD, since no data thrashing incurs without data reuse. Hence, we conclude that it is critical to reduce the data movement and page fault overhead to achieve better unified memory performance for these benchmarks.

### 3.4 Improving Unified Memory Performance

As shown in Section 3.3.3, unified memory does not always perform well. Users need to put in extra efforts in order to achieve optimized performance. In this Section, we will introduce several programming guidelines we get from the performance results.

**Applications with Significant Data Reuse.** For such cases, we observe that unified memory is preferable when dataset fits into GPU memory, but it suffers from significant data thrashing when dataset does not fit. In such scenarios, we suggest programmers to pin data with good locality into
GPU memory and allow unified memory runtime to manage the data with low locality. We leverage the existing OpenMP APIs to achieve data pinning. For instance, in order for an array A to be pinned, its memory must be allocated in the regular fashion (without unified memory). Then, in an OpenMP target region, map clauses are required to allow the OpenMP runtime to explicitly move pinned data between CPU and GPU.

In doing this, the pinned data cannot be replaced by the unified memory management policy and thus will get consistent reuse. Data thrashing will be reduced since it will only happen to those unpinned data with poor locality. Note that it is not possible to pin data whose size is larger than the GPU memory size.

Applications with Little Data Reuse. In these cases, our evaluation shows that the achieved performance is slightly better without unified memory since all the data can be preloaded into the GPU memory, thereby avoiding the page fault overhead. This does not mean we should give up unified memory in these applications since the ease of programming with unified memory is still desirable. Instead, we provide suggestions to help unified memory achieve similar performance without significant modifications.

To reduce overhead associated with on-demand data migration, we suggest programmers to insert prefetching instructions before the data is accessed. By doing this, the data will be ready in GPU memory when needed and both data migration and page fault overhead will be minimal. OpenMP programs can use the CUDA runtime API cudaMemPrefetchAsync on the desired data to achieve this goal, thanks to the interoperability of OpenMP and CUDA.

3.5 Related Work

A large amount of benchmarks have been developed to study GPU performance. Rodinia [19] is one of the first suites developed for GPU benchmarking. Currently, it provides implementations for programming models such as CUDA, OpenCL and OpenMP. It is also arguably the most widely used one. Parboil benchmarks [131] were developed for throughput processors including GPUs, and they are written in CUDA, OpenCL, and OpenMP. SHOC [28] is a GPU benchmark suite designed for both OpenCL and CUDA with a focus on scalability. PolyBench/ACC [43] ports the PolyBench suite to run on GPUs. The NUPAR benchmark suite [143] supports the exploration of
relatively new GPU features, including dynamic parallelism and concurrent kernel execution in CUDA and OpenCL. Valar [99] aims to study the interaction between CPU and GPU in heterogeneous systems. NAS parallel benchmarks [65] are widely used to benchmark supercomputers. Although OpenMP variants are provided, GPU support is not yet available. Furthermore, none of these benchmarks suites explore the unified memory features of recent GPUs, especially on the presence of on-demand page migration. Thus, we believe this work contributes to fill part of the knowledge gap.

Since the introduction of device offloading in OpenMP 4.0, several compilers have adopted this feature. For instance, Antao et al. [6] describes how to implement this extension in the LLVM compiler. Bercea et al. [12] evaluated the performance of OpenMP 4.0 GPU offloading using a CORAL proxy application and found that its performance is comparable to that of CUDA after some tuning. Martineau et al. [88] evaluated the performance of LLVM OpenMP 4.5 GPU support using several simple kernels and mini-apps. Based on the results, they propose to perform optimizations by combining directives, caching read-only data, etc. Cray’s compiler also supports OpenMP GPU offloading currently. In another research Martineau et al. [89] also studied the OpenMP GPU offloading performance of the Cray compiler on NVIDIA K20X and found that its performance is reasonably similar to that of CUDA. None of these compilers have unified memory support currently, and thus existing studies do not pay attention to unified memory.

There are several proposals to simplify and optimize GPU memory management. CGCM [62] provides compiler and runtime support to automatize GPU memory management for CUDA programs. Pai et al. [112] propose a software coherence mechanism to reduce redundant data transfer between CPU and GPU. These works aim at traditional GPU programs where data movement is managed explicitly by the programmer, and does not consider unified memory.

3.6 Conclusion

In this chapter, we describe how we develop benchmarks for OpenMP GPU offloading, with a focus on unified memory. Using these benchmarks, we evaluate the performance of unified memory and discover some inefficiencies existing in current unified memory system. We also provide several programming guidelines for users to achieve better performance with unified memory.
More importantly, we discovered how difficult GPU programming can be, even with OpenMP as a programming language. We came to the conclusion that using just an OpenMP directive is insufficient to get the GPU to operate at its peak performance. In the grand scheme of things, if a tool or framework exists that can identify kernels that can be offloaded to GPU and automatically modify the source code to make it run effectively on an HPC cluster, it would be extremely beneficial to any application scientist. This motivated us to create a compiler tool that automatically transforms an OpenMP code to effectively offload to a GPU.
Chapter 4

State of the Art

“People think that computer science is the art of geniuses, but the actual reality is the opposite; just many people doing things that build on each other, like a wall of mini stones.”

– Donald Knuth

We learned how crucial GPUs are to the HPC sector in the earlier chapters, as well as how challenging it is to program for GPUs. Using a directive-based language like OpenMP is one of the easier ways to write GPU-based programs. However, as we discussed in the previous chapter on our motivation for this work, we realized that simply using an OpenMP directive is insufficient to extract the maximum performance from the GPU. We must fully understand the underlying architecture as well as the interfacing programming model. For application scientists, GPU programming can be quite labor intensive and time-consuming, which is a major deterrent in practice and obviously has an impact on productivity.

4.1 GPU Offloading

Graphics Processing Units (GPUs) are a promising technology in the field of high-performance computing (HPC). GPUs are highly parallel co-processors, providing FLOPS at low power consumption. They were developed as co-processors for the graphics and gaming industry and were co-designed to fit the needs of these real-time applications, using the fast SIMD parallel pipeline that addressed graphics needs. This SIMD parallel architecture is
also suited to many scientific problems, so research and development on the use of GPUs for scientific simulation and HPC intensified in the 2000s.

GPUs are a fairly new technology in HPC, so many interactions of GPUs with HPC systems are not yet fully characterized. These include such questions as the reliability of the GPU, the patterns of errors that do occur and the performance of GPUs as an ensemble within an HPC system. These questions are especially important in a highly coupled high-performance computing system, in which performance or reliability problems on a single GPU might affect the entire calculation.

4.1.1 The Benefits of Using GPUs

Within a comparable price and power range, a GPU offers much higher instruction throughput and memory bandwidth than the CPU. Numerous applications take advantage of these improved capabilities to run on the GPU more quickly than on the CPU. Other computing devices, such as FPGAs, are also very energy efficient, but they provide significantly less programming flexibility than GPUs. The GPU is made specifically for highly parallel computations, so more transistors are allocated to data processing than to flow control and data caching. The schematic Figure 4.1 shows an example distribution of chip resources for a CPU versus a GPU.

The GPU can hide memory access latencies with computation rather than relying on large data caches and complex flow control to avoid long

Figure 4.1: Transistors Dedicated to Data Processing on CPU vs GPU
memory access latencies, both of which are expensive in terms of transistors. Dedicating more transistors to data processing, for example, floating-point computations, is advantageous for highly parallel computations. Applications with a high degree of parallelism can take advantage of the GPU’s massively parallel nature to achieve higher performance than on the CPU because applications typically have a mix of parallel and sequential components. As a result, systems are designed with a mix of GPUs and CPUs to maximize overall performance.

Companies that manufacture graphics hardware, like NVIDIA and AMD ATI, have created graphics processors with massively parallel processing capabilities. Let us take NVIDIA GPUs for example. These processors typically require high throughput and memory bandwidth to display high resolution graphics. These hardware tools could, however, be repurposed and used for tasks unrelated to graphics. Direct programming of the NVIDIA hardware is possible thanks to NVIDIA’s CUDA [103] (Compute Unified Device Architecture) programming interface. Massively parallel algorithms can be executed on NVIDIA devices at speeds that are many times faster than sequential implementations on traditional CPUs. It is important to comprehend what is going on at the hardware level of an NVIDIA GPU if we want to use a programming language like OpenMP to get the most performance out of it.

4.1.2 NVIDIA Thread Organization

There is a concept of a kernel in the CUDA processing paradigm. In essence, a kernel is a small program or subroutine. Kernels are the parallel applications that will run on the accelerator. A kernel program will run simultaneously on a number of primitive threads. These simple threads are grouped together into thread blocks. The number of primitive threads in a thread block is determined by the amount of shared memory that is available as well as the desired memory access latency hiding properties.

The architecture also places a cap on the number of threads in a thread block at 512 in total. The shared memory scoped to each thread block enables effective communication between each thread within that block. All threads within a thread block can also sync using this shared memory. Each thread has its own thread ID within a thread block. For convenience, thread blocks are conceptually arranged into 1D, 2D, or 3D arrays of threads.

A group of thread blocks with the same thread dimensionality that all
run the same kernel is referred to as a grid. Since there are a physical limit of 512 threads per block for thread blocks, grids are useful for computing many threads simultaneously. However, thread blocks in a grid might not be able to communicate with one another via shared memory, making it difficult for them to synchronize. The diagram in Figure 4.2 demonstrates the thread hierarchy described. Here, a given kernel contains a 3x2 grid of thread blocks. Each thread block is a 4x3 thread block, for a total of 72 threads running the kernel.

4.1.3 Streaming Multiprocessors

The Tesla architecture is used to build GPUs that can support CUDA. Any graphics card that supports this architecture can run a CUDA applications, but each GPU device may have different specifications, which means a slightly different set of supported features and a different amount of computational resources. Each thread block runs on a multiprocessor when a kernel is called. This multiprocessor has the capacity to support a certain number of threads.

When a kernel is running, one or more thread blocks are assigned to
Figure 4.3: Technical Specifications per Compute Capability [45]

A multiprocessor. On a collection of multiprocessors, the CUDA runtime manages the dynamic scheduling of thread blocks. Only when sufficient

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>3.5</th>
<th>3.7</th>
<th>5.0</th>
<th>5.2</th>
<th>5.3</th>
<th>6.0</th>
<th>6.1</th>
<th>6.2</th>
<th>7.0</th>
<th>7.2</th>
<th>7.5</th>
<th>8.0</th>
<th>8.6</th>
<th>8.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of resident grids per device</td>
<td>32</td>
<td>16</td>
<td>128</td>
<td>32</td>
<td>16</td>
<td>128</td>
<td>16</td>
<td>128</td>
<td>16</td>
<td>128</td>
<td>16</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x-dimension of a grid of thread blocks</td>
<td>$2^{31} - 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum y- or z-dimension of a grid of thread blocks</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum dimensionality of a thread block</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per SM</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per SM</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per SM</td>
<td>2048</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per SM</td>
<td>64 K</td>
<td>128 K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>64 K</td>
<td></td>
<td></td>
<td>32 K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>64 K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per SM</td>
<td>48 KB</td>
<td>112 KB</td>
<td>64 KB</td>
<td>96 KB</td>
<td>64 KB</td>
<td>96 KB</td>
<td>96 KB</td>
<td>64 KB</td>
<td>96 KB</td>
<td>64 KB</td>
<td>164 KB</td>
<td>100 KB</td>
<td>164 KB</td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per thread block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of local memory per thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>512 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of local memory per thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
resources are available to support the thread block will the scheduler assign a multiprocessor a thread block. Each block is divided into warps, or SIMD (Single-Instruction Multiple-Data) thread groups. To create a warp, the SIMD unit simultaneously creates, manages, schedules, and runs 32 threads. Since each warp executes as quickly as its slowest thread, care must be taken to prevent any warps from taking abnormally long to complete for some threads relative to other threads in the same warp.

4.1.4 NVIDIA Compute Model

The feature set of the device, including both its hardware and software features, is known as its compute capability. A compute compatibility number is assigned to every CUDA-capable device. This number represents the typical number of registers, memory size, etc. for all compatible devices. Backward compatibility exists for compute compatibility numbers. Table 4.3 shows some of the technical specifications associated with each compute capability that is currently supported.

4.2 OpenMP Offloading

Accelerator offloading is supported by the OpenMP standard as of version 4.0. By using these directives, programmers can transfer data and computation to hardware like GPUs. This makes it easier to write portable, heterogeneous parallel code. The directive’s name comes from the fact that OpenMP uses the target construct to offload execution from the host to the target device(s). Additionally, the relevant data must be transferred to the device(s). Once transferred, the data becomes the property of the target device, and access by the host while the target region is being executed is prohibited.

OpenMP typically uses a host - device model for offloading. Typically, there is only one host, such as a CPU, and one or more target devices of the same type, such as a GPU, FPGA, etc. The host and device each have their own memory address space unless using unified shared memory. The device’s execution is host-centric. Data is first mapped to the device(s) data environment by the host, which also creates the device(s) data environments. The target device is then given access to the OpenMP target regions from the host for execution. Upon completion, the host transfers the data from
the device to the host and destroys the device data environment.

Because OpenMP separates offload and parallelism, the target construct transfers the control flow to the device in a sequential and synchronous manner. To effectively use the target device, parallel regions must be explicitly created on it.

A league of one-thread teams is produced by the teams construct, with each team’s thread running concurrently and in its own contention group. The number of teams created is implementation defined, but if specified in the clause, it cannot be greater than num_teams. Unless thread limit is specified in the clause, the implementation defines the maximum number of threads that each team may have in the contention group. The maximum number of teams or threads, for NVIDIA GPUs, for example, depends on the device’s compute capability as described in Table 4.3. There is no inter-team synchronization but threads within a team can synchronize. There must be no other instructions, statements, or declarations between the teams construct and the target construct.

The distribute construct is a coarsely worksharing construct that distributes loop iterations among the master threads in the teams, but does not allow worksharing between threads within the same team. There is no implicit barrier at the conclusion of the construct, and there is no assurance about the order the teams will execute. The programmer must use the parallel for constructs to further divide loop iterations across threads and create additional threads within each team.

4.3 GPU Cost Model

Several works in the literature have proposed cost models to represent and predict application and kernel performance. Many studies have been conducted in an effort to suggest a method for precisely forecasting GPU performance and multithreaded program performance at compile-time. Even though there is a large body of research on various cost models for GPUs, we need predictions to be more accurate than any of these models can provide.

One of the first analytical models for GPUs was put forth by Hong and Kim [52] using Memory Warp Parallelism (MWP) and Computation Warp Parallelism (CWP). While CWP represents how many computations can be carried out by other warps while one warp is waiting for memory values, MWP represents the number of memory requests that can be executed con-
Currently, they calculate the effective costs of memory requests using MWP and CWP, which also calculates the total amount of time a program will take to run. Despite the fact that this model captures a rough estimate of the cost in terms of memory operations, it does not account for memory bank conflicts.

An analytical model to forecast the performance of general-purpose applications on a GPU is discussed by Baghshorki et al. [7] (validated on NVIDIA GeForce 8800 GPU using CUDA). This model’s purpose is to support an auto-tuning compiler by giving performance data. They examine GPU kernels to determine the key GPU microarchitecture features that each kernel employs. They use FFT kernels to assess memory bandwidth and latency and dense matrix multiplication to test the effectiveness of their model. They then apply it to prefix sum scan kernels, which do not distribute computation equally across threads. They then applied their performance model to kernels for sparse matrix-vector multiplication, which are renowned for their erratic and indirect memory accesses. They created a model that enables a compiler to evaluate different parallel kernel configurations without having to test every possible combination. More importantly, the model locates bottlenecks and can direct the compiler during optimization. The model is tested against various benchmarks for matrix multiply, prefix sum scan, FFT, and sparse matrix-vector multiplication. The performance of various tuned versions of the algorithms could be correlated between predictions and actual results, they were able to demonstrate. However, because data caching on GPU is not taken into account, this model does not adequately handle applications that require lots of data.

In order to identify performance bottlenecks and quantitatively analyze performance, Zhang and Owens [156] proposed a performance model that measures the execution time spent on the instruction pipeline, shared memory, and global memory access. This model enables programmers and architects to forecast the advantages of potential program optimizations and architectural improvements. Although their method aims to identify bottlenecks, it does not deliver the expected performance benefits.

For a large number of threads sharing a cache, Chen and Aamodt [20] proposed analytical models to accurately predict the number of extra cache misses caused by cache contention. A novel cache contention model is used to estimate the throughput by applying a Markov chain model, where states represent the number of stalled threads, along with transition statistics. This model accurately predicts cache contention among many threads while ac-
Sim et al. [128] propose a framework, GPUPerf, which quantitatively estimates the potential performance benefits along four dimensions: (a) inter-thread instruction-level parallelism; (b) memory-level parallelism; (c) computing efficiency; and (d) serialization effects. The authors suggest what types of optimization programmers (or even compilers) should try first using these four metrics. Their analytical model is an improved version of Hong and Kim’s [52] MWP-CWP model and applies to similar problems.

The most recent model that we evaluate in this study is the performance model developed by Huang et al. [55]. They proposed a performance modeling technique for GPU architectures based on interval analysis. If previous models were inaccurate in the latency-bound case, this model is inaccurate in the throughput-bound case. The model calculates the reciprocal execution rate as the sum of two terms: memory system contention delays and all other delays. When memory accesses are combined, contention delays are assumed to be zero.

NVIDIA [147] recently investigated the cause of the unreliability of these cost models. They discovered the following factors that contributed to these models’ failure:

1. Ignoring arithmetic latencies, which are small but can add up to large numbers;
2. Ignoring memory bandwidth, which can still limit throughput even when all accesses are fully coalesced;
3. Assuming that NVIDIA GPUs can switch to a different warp only on long latency events, not on each cycle;
4. Using wrong concurrency in Little’s law: warp concurrency (occupancy), instruction concurrency, arithmetic instruction concurrency, and memory instruction concurrency are all different quantities.

In addition to the NVIDIA conclusion stated above, our experience has shown that these cost models are unreliable due to the absence of two key components. The first is that none of the models previously mentioned take into account the cost of data transmission between the CPU and the GPU. They either assume that data is available in the GPU memory or they think the cost is negligible. We have already shown in chapter 3 on motivation that data transmission has a significant impact on performance and cannot be disregarded in applications that require a lot of data. Second, all of the cost models previously mentioned are overly reliant on the system architecture,
and given how rapidly GPU architecture is evolving, these cost models are quickly rendered obsolete.

In Chapter 7, we’ll introduce COMPOFF, a cost model that uses a machine learning model to statically estimate the Cost of OpenMP OFFloading. This cost model not only accounts for all the aforementioned failure modes, but it is also easily adaptable to various GPUs and independent of any specific architecture. If a completely new GPU architecture emerges in the future, we may very well be able to extend this model to that architecture as long as we collect enough data based on that architecture. This model is also compiler-independent and can be easily integrated with any compiler of the user’s choice.

4.4 What is missing?

We recognize that getting the most out of the GPUs is not an easy job. Although directive-based programming models, such as OpenMP, make it easier for application developers to offload their code to a GPU, extracting maximum performance from them is not always easy. GPU programming can be quite time-consuming and labor-intensive for application scientists, which in practice is a big barrier and unavoidably affects productivity. A good solution would be for compilers to effectively offload the kernel to the available GPU by optimizing the code. Looking at the whole picture, any application scientist would breathe a sigh of relief if there were a tool or framework that could recognize suitable kernels and automatically modify the source code to make it work effectively on an HPC cluster like Summit or Frontier. Such a framework is urgently needed by the HPC community. However, a compiler needs a good cost model to select the best optimization from all available transformations. Unfortunately, given how rapidly GPU design is evolving, it is virtually impossible for cost models to keep up with all of the architecture. Consequently, there is a need for a cost model that is not only accurate but also adaptable to different GPU architectures.

Developing such a framework for a single node on an HPC cluster is the goal of the research I conducted for my PhD. This framework’s task is to automatically modify OpenMP code so that it can be effectively offloaded to a GPU. To that end, we propose an OpenMP Static Offload Advisor design in the upcoming chapters.
Chapter 5
OpenMP Advisor

Developers in the HPC industry have started looking beyond Moore’s law and hardware developers have improved chip performance by configuring a growing number of compute cores. This rapid change in architecture hardware necessitates programming language updates and modification of application code to exploit the cores, e.g. by inserting pthreads or OpenMP [27] constructs into the source code. The HPC community was quick to embraced this multi-core processor technology.

In the last decade, General Purpose Graphics Processing Units (GPGPUs) have been attached to the multi-core processors on many HPC platforms in order to benefit from their ability to handle large amounts of data parallelism with low power consumption. Initially intended for graphics operations, they are now being used extensively by HPC clusters and for general-purpose computing on graphics processing units. Although there are some notable exceptions, the most recent TOP500 list [139] clearly demonstrates the trend toward heterogeneous HPC platforms, particularly those using GPUs. A significant portion of the systems are heterogeneous, with AMD or NVIDIA GPUs typically providing high performance per watt.

5.0.1 The Challenge

Many programmers are adapting their code to take advantage of GPUs. Unfortunately, optimizing the computational power of a GPU while minimizing overheads is a laborious process that may necessitate re-engineering large sections of code and data structures. The development of code for systems with extreme heterogeneity and numerous devices will be much more challeng-
\[ D_{\alpha\beta}^{ij}(x, y) = \sum_{\mu=1}^{4} \left[ (1 - \gamma_{\mu}) \alpha\beta U_{\mu}^{ij}(x) \delta_{x+\hat{\mu}, y} + (1 + \gamma_{\mu}) \alpha\beta U_{\mu}^{ij}(x + \hat{\mu}) \delta_{x-\hat{\mu}, y} \right] \] 

(5.1)

ing. Therefore, it is essential to create tools that will relieve the application scientists of the burden of such development.

Despite the variety of programming models available, it is still quite challenging to optimize large scale applications consisting of tens-to-hundreds of thousands lines of code. Even when using a directive based programming model such as OpenMP, pragmatizing each kernel is a repetitive and complex task. OpenMP offers a variety of options for offloading a kernel to GPUs. However, the application scientist must still figure out all the intricate GPU configurations. To demonstrate the complexity of porting a kernel to emerging exascale hardwares, we use a kernel from the Lattice Quantum Chromodynamics (LQCD) [16] application, which is a computer-friendly numerical framework for QCD. One of LQCD’s key computational kernels is the Wilson Dslash operator [80], which is essentially a finite difference operator, to describe the interaction of quarks with gluons.

The Wilson Dslash operator, \( D \), in four space-time dimensions is defined by Equation 5.1. Here \( x \) and \( y \) are the coordinates of the lattice sites, \( \alpha, \beta \) are spin indices, and \( i, j \) are color indices. \( U_{\mu}(x) \) is the gluon field variable and is an \( SU(3) \) matrix. \( \gamma_{\mu} \)'s are \( 4 \times 4 \) Dirac matrices that are fixed. The complex fermion fields are represented as one-dimensional arrays of size \( L_X \times L_Y \times L_Z \times L_T \times SPINS \times COLORS \times 2 \) for the unpreconditioned Dirac operator.

```c
#pragma omp target
for(int t=1; t<LT+1; t++) {
  for(int z=1; z<LZ+1; z++) {
    for(int y=1; y<LY+1; y++) {
      for(int x=2; x<LX+2; x++) {
        /* ... COMPUTE ... */
      }
    }
  }
}
```

Code 5.1: Loops of Wilson Dslash Operator
where $L_X, L_Y, L_Z$ and $L_T$ are the numbers of lattice sites in the $x, y, z$ and $t$ directions, respectively. The spin and color degrees of freedom, which are commonly 4 and 3, are denoted by the variables $SPINS$ and $COLORS$.

When we express Equation 5.1 in C++, it has four nested for loops iterating over $L_T, L_Z, L_Y$, and $L_X$, as shown in Code 5.1. When we keep the values of $L_T, L_Z, L_Y$, and $L_X$ at 16 each, the COMPUTE section of the code has over 5 million variable definitions, 1.2 billion variable references, over 150 million addition/subtraction, 163 million multiplication, and so on. Additionally, this function is called several times throughout the LQCD application.

It is a herculean task for an application scientist to understand the physics, transform it into computer program, analyze the offloadable kernel, and then consider how to parallelize it to execute efficiently on an HPC cluster. To get the best performance out of a GPU, an application scientist needs a thorough understanding of the underlying architecture, algorithm, and interface programming model. Alternately, they could test out various GPU transformations until they find the most effective one. However, none of these strategies is very efficient.

5.0.2 Our Contribution

This dissertation presents OpenMP Advisor, a first-of-its-kind compiler tool that advises application scientists on various OpenMP code offloading strategies. This tool performs the following tasks to successfully address the challenges of effectively transforming an OpenMP code:
1. detect potentially offloadable kernel;
2. identify data access and modification in kernel;
3. recommend several potential OpenMP variants for offloading that kernel to the GPU;
4. evaluate the profitability of each kernel variant via an adaptive cost model;
5. insert pertinent OpenMP directives to perform offloading.

Although the tool is currently limited to GPUs, it can be extended to support other OpenMP-capable devices.
5.1 Related Work

Many studies have looked into how to best manage GPU memory when data movement must be explicitly managed. A fully automatic system for managing and optimizing CPU-GPU communication for CUDA programs is provided by Jablin et.al. [62]. Gelado et.al. [39] present a heterogeneous computing programming model to simplify and optimize GPU data management. OMPSan [11] performs static analysis on explicit data transfers that have already been inserted into an OpenMP code. However, these studies do not provide insight into how data must be transferred and how data reusability on GPU can be used for implicitly managed data between multiple kernels. In Chapter 6 we propose a technique for statically identifying data used by each kernel and automatically recognizing data reuse opportunities across kernels. In our Advisor, we use this technique to manage data between the CPU and the GPU.

Recently, Roy et.al. [120] developed BLISS, a novel approach for auto-tuning parallel applications without the need for instrumentation, domain-specific knowledge, or prior knowledge of the applications. With the help of Bayesian optimization, this auto-tuner creates a collection of various lightweight models that can rival the output quality of a complex, heavyweight Machine Learning (ML) model. Other autotuners like BOHB[35], HiPerBot [91], GPTune [82] and ytopt [151] frequently use Bayesian optimization in this context. However, due to their expensive evaluation functions, tuning large-scale applications remains a challenge. Besides, the need to optimize HPC programs on heterogeneous hardware (such as GPUs) and software configurations prevents the widespread use of these auto-tuning techniques in the Exascale era. HPC applications are getting extremely heterogeneous, complicated, and increasingly expensive to analyze.

There is a need for a tool like ours that assists application scientists to offload their code to GPUs. Other related research on automatic GPU offloading by Mendonça et.al. [90] and Poesia et.al. [116] can benefit from our tool by including our technique of data optimization and cost model in their framework, further reducing the challenges of using GPUs for scientific computing. However, developing a cost model is time-consuming, and practically all contemporary compilers, such as LLVM/Clang [85], adopt a straightforward “one-size-fits-all” cost function that does not perform optimally in the situation of extreme heterogeneous architecture. With the use of a neu-
ral network model, in Chapter 7 we define COMPOFF which offers a new portable cost model that statically estimates the cost of OpenMP offloading on different architectures. We extend the techniques defined in COMPOFF to develop a portable static cost model to be used in our OpenMP Advisor tool.

It is well known that source-to-source transformation of C/C++ code is a challenging problem. Until recently, Clang was not a popular compiler option for source-to-source transformation. Developers used tools such as the Rose compiler [117] to accomplish source-to-source transformation. However, the recent development of Clang as a potent and library-friendly C++ compiler demonstrates that there may finally be some relief in sight. We use LLVM/Clang as the main source-to-source transformer.

5.2 OpenMP Advisor

We design and develop the OpenMP Advisor, a compiler tool which transforms an OpenMP code to effectively offload to a GPU. This tool identifies OpenMP kernels, suggest a number of possible OpenMP variants for offloading that kernel to the GPU, and predict the cost of running each of these kernels separately using a static neural network-based compile time cost model. The OpenMP Advisor’s first version aids the application scientists in writing code for accelerators like GPUs. This Adviser, however, can be extended to support all OpenMP-enabled devices.

The tool has two modes: Training mode and Prediction mode. The training mode must be executed on the target hardware. It takes all benchmark codes as input and generates all possible variants of the code to run on the target device. Then it collects data from all generated codes to train an ML-based cost model for use in prediction mode. In prediction mode the tool does not need any interaction with the target device. It accepts C/C++ code as input and returns the best code variant that can be used to offload the code to the specified device. The tool can determine the kernels that are best suited for offloading by predicting their runtime using a machine learning-based cost model as defined in Section 5.3.2.

5.2.1 Core Attributes

The following are the key attributes of the OpenMP Advisor.
1. **Static** – Access to GPUs is a significant challenge when analyzing an application using GPU offloading. During development, many application scientists may not have easy access to GPUs. As a result, it is critical that in the prediction mode our Advisor performs all its analysis at compile time and does not require runtime profiling. Even in the training mode, sometimes it is impractical to predict certain values during code generation. In order to help the advisor in such circumstances, we provide a json file for input where users can provide these values.

2. **Minimalistic** – The Advisor makes no changes to the kernel’s body. It simply identifies the application’s `omp target` regions and adds various OpenMP directives and clauses to generate multiple kernel variants.

3. **Correctness** – The Advisor verifies the correctness of the generated code to keep in line with OpenMP programming model. Currently, the advisor does not modify the kernel body and does not guarantee the correctness of its content. Consequently, despite its ability to predict the optimal scenario for GPU offloading, it generates the top N (N provided by the user) code variants and lets the application scientist select which code to utilize.

4. **Portability** – The Advisor is portable to multiple compiler and HPC clusters. In our work we worked on four different compilers: LLVM/clang (nvptx), LLVM/clang (rocm), GNU/gcc and NVIDIA HPC/nvc. The advisor also works for a variety of HPC clusters with different GPUs: such as Summit [109], Ookami [17], Wombat [111] and Sea-wulf [144] clusters using NVIDIA GPUs and Corona [74] using AMD GPUs. Limited work was also done on the Intel Devclouds [60] with Intel GPU and icpx compiler.

5. **Adaptability** – The advisor is adaptable enough to accept new applications. When attempting to predict for a new application for which real-time data collection is difficult or impractical, the application scientist could create a proxy application similar to the target application and train the model using data collected on the proxy application.
5.2.2 Design

Before proceeding with the implementation, there were a few design challenges that needed to be addressed. The first concern was which toolkit or compiler to employ in the development of our framework. This was a straightforward question since we used the most popular library at the time – the LLVM compiler project. LLVM is a library for building, optimizing, and producing intermediate and/or binary machine code. Like most modern compilers, LLVM divides the compilation process into three levels, as shown in Figure 5.1:

1. **Frontend**: The frontend translates high-level language (like C/C++) to LLVM-IR.

2. **Middle-end**: Performs several optimization passes on the LLVM-IR.

3. **Backend**: Converts the optimized LLVM-IR into the assembly language of the corresponding platform.

We chose the Clang compiler (ver 14.0.0) because our target applications are written in C++. Clang is a tool development platform as well as a compiler for the C language family in the LLVM project. Because of its library-based architecture, reusing and integrating new features into other projects is more flexible and simple.

We then had to decide at what level we should implement our framework. Despite the fact that LLVM’s strength lies in the LLVM-IRs, our requirement is to generate and return C/C++ code to the scientist. We require the accurate source location from a C/C++ file in order to insert OpenMP directives.

![Figure 5.1: LLVM Architecture](image-url)

Figure 5.1: LLVM Architecture
Figure 5.2: High level flow of the Training mode

into that file. Although the LLVM-IR can contain source code information in source level debugging, the accuracy of the source location cannot be guaranteed. On the other hand Clang’s AST closely resembles both the written C++ code and the C++ standard. Clang has a one-to-one mapping of each token to the AST node and an excellent source location retention for all AST nodes. The AST is the best place to get accurate source code information. Hence we implemented our Advisor in the frontend (Clang). The Advisor
follows the design depicted in Figure 5.2 in the training mode, and Figure 5.3 in the prediction mode. Both modes have three major modules, which are explained in the following subsections - *Kernel Analysis* (Section 5.3.1), *Cost Model* (Section 5.3.2) and *Code Transformation* (Section 5.3.3).
5.3 Clang Plugins Implementation

Clang invokes the preprocessor, which expands all macros and parses the source code into an Abstract Syntax Tree (AST). The preprocessed AST is much easier to work with than the source-level C code, and we can always easily reference the original code using Clang’s Plugins [23] interface. Clang Plugins enable the execution of additional user-defined actions during compilation. The compiler loads a plugin from a dynamic library at runtime. The FrontendAction interface, which enables user-specific actions to be executed as part of the compilation, is the typical entry point when developing a clang-based tool, such as a Clang Plugin. Clang provides the ASTFrontendAction, an interface that handles action execution, to run tools over the AST.

We define and register a plugin called VariantPluginAction, that implements the CreateASTConsumer method, which returns an ASTConsumer per translation unit. ASTConsumer is an interface that allows developers to write generic actions on an AST regardless of how it was generated. The Clang’s AST is consumed (or read) by the ASTConsumer. By overriding as many functions as we want, we are able to have our code executed each time a particular type of AST node is parsed.

There are a number of possible entry points for the ASTConsumer, one of which is HandleTopLevelDecl, which is called each time Clang parses a new set of top-level declarations. However, by overriding HandleTopLevelDecl, we can force a function to execute our code whenever a new Decl is parsed in the source code rather than waiting until the entire source file has been parsed. This creates a problem because, from the parser’s perspective, we cannot access or reason regarding code written after the function we’re currently analyzing. Hence for our application, HandleTranslationUnit is required, which is called only after the entire source file is parsed. In this case, a translation unit effectively represents an entire source file, and the AST for that source file is represented by an ASTContext class.

Clang provides a RecursiveASTVisitor [24] class that traverses the entire Clang AST in preorder or postorder depth-first order and visits each node. This class has three distinct responsibilities:

1. Recursively traverse the AST (i.e. visit each AST node), using the TraverseDecl method.
2. At a given node, use the WalkUpFrom<X>(<X>*) method to walk up the class hierarchy, starting from the node’s dynamic type <x>, until the
top-most class is reached.

3. Given a \((\text{node}, \text{class})\) combination, where ‘\text{class}’ is some base class of the dynamic type of ‘\text{node}’, call a user-overridable function to actually visit the node. This is done using the \text{Visit}<X>(<X>*) methods, where <X> is the base class.

Using the \text{RecursiveASTVisitor} we may visit any type of AST node, such as \text{FunctionDecl} and \text{Stmt}, simply by overriding a function with that name, e.g., \text{VisitFunctionDecl} and \text{VisitStmt}. Instead of calling any of the \text{Visit*} functions directly, we should use \text{TraverseDecl}, which will call the appropriate \text{Visit*} function in the background. From such \text{Visit*} functions, we must return \text{true} to continue traversing the AST (to examine other nodes) and \text{false} to halt the traversal and essentially exit Clang.

In our plugin, we define our own \text{TargetASTConsumer}, an \text{ASTConsumer}'s instance that uses the \text{HandleTranslationUnit} as an entry point. In the following sections, we will go over the three modules depicted in Figure 5.2 and 5.3 – \text{Kernel Analysis}, \text{Cost Model} and \text{Kernel Transformation}. Each of the three modules are called from the \text{HandleTranslationUnit} method of our \text{TargetASTConsumer} to perform their respective tasks.

### 5.3.1 Kernel Analysis

This is the first module which the \text{HandleTranslationUnit} method calls in both the Prediction and Training mode. As the name implies, this module analyzes an OpenMP kernel. Overall, this module takes as input a C/C++ source file, analyzes it, and outputs all possible GPU offloading variants. This module is responsible for three tasks: \text{Identifying Kernels}, \text{Data Analysis} and \text{Variant Generation}.

#### Identifying Kernels.

User’s input is utilized to help us identify a target region since the scope of this project is not to automatically parallelize the code. An application scientist only needs to use the “\text{omp target} directive” to mark the region. We parse the clang AST to search for the \text{OMPTargetDirective} node, which is a subclass of the \text{OMPExecutableDirective} class and an instance of the \text{Stmt} class. We override the \text{OMPAdvisorVisitor} class’s \text{VisitStmt} method and check each visited \text{Stmt} to see if they are the \text{OMPTargetDirective} node.
Once a kernel is identified, we assign it a unique id and create an instance of the class “KernelInformation”. In that object we store information like, unique_id, start and end locations of the kernel, function from which the kernel is called, whether the kernel is called from within a loop and the number of nested for loops. All of this information will further be used to identify variables accessed within kernels.

**Data Analysis**

The next step is to determine what data the kernel uses. Since data transfers to and from the GPU are extremely expensive, careful data management and mapping between host and device are essential for the use of accelerators in HPC. The complexities of achieving efficient data management are a deterrent to GPU utilization. Identifying and utilizing GPU data reuse in an application automatically can reduce the amount of data transferred and thus the overall execution time. In our work on Data Transfer and Reuse Analysis [97], we discuss how to gather information on which variables are being used by a kernel. We use the Clang AST to implement *live variable analysis* for each kernel, concentrating only on variables used within a kernel. Before the variables are stored in the `KernelInformation` object, we classify them into five groups, based on how they are accessed before, during, or after the kernel:

- **alloc**: These are the variables that are being *assigned within* the kernel for the first time and no data transfer to the GPU is required.
- **to**: These are variables that were *assigned before* but were only *accessed* and not modified within the kernel. This data must be transferred to the device but does not have to be transferred back to the host.
- **from**: These are variables that are *updated within* the kernel and *accessed after* the kernel call. This category does not require data transfer to the device, but it does necessitate data transfer from the device to the host.
- **tofrom**: These are variables that are *assigned before*, *updated within* and *accessed after* the kernel definition. This type of data must be transferred in both directions between the host and the device.
- **private**: Finally, these are variables that are defined and used *only within* the kernel, and does not need to be transferred between the host and the device.

Data labeled alloc, to, and tofrom are mapped in “omp target enter data” directives before the kernel, while data labeled from and tofrom are
mapped in “omp target exit data” directives after the kernel. Chapter 6 provides a more thorough explanation of the data reuse analysis work.

**Variant Generation**

Finally, we generate a number of different kernel variants that can be used to offload the kernel to the GPU. We’ll start by counting how many nested collapsible for loops are there. In the current implementation, we can check up to four levels of collapsing the for loops. We chose four nested loops because, as shown in Code 5.1, the Wilson Dslash kernel has four for loops. Each of these for loops is given a unique position number ranging from 0 to 3. The for loop at position 0 is always expected to be distributed across all teams on the GPU. The variants are generated based on the following parameter:

1. Value of collapse used in **teams distribute** directive
2. Value of collapse used in **parallel for** directive
3. Position of the **parallel for** directive
4. Scheduling type of the loop iterations
5. Data transfer between the host and the device

The total number of for loops and the position of the **parallel for** directive determine the maximum value of **collapse** that can be used in **teams distribute** and **parallel for** directive. For instance, suppose there are

```c
#pragma omp target teams distribute collapse(1) 
for (int i = 0; i < N_i; i++) { <= Loop Position 0
#pragma omp parallel for collapse(3) schedule(static)
    for (int j = 0; j < N_j; j++) { <= Loop Position 1
        for (int k = 0; k < N_k; k++) { <= Loop Position 2
            for (int l = 0; l < N_l; l++) { <= Loop Position 3
                ...
            }
        }
    }
}
```

Code 5.2: Four nested “for” loops, with “teams distribute” in loop position 0 and “parallel for” in loop position 1
<table>
<thead>
<tr>
<th>Number of for loops</th>
<th>Total Variants Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>84</td>
</tr>
</tbody>
</table>

Table 5.1: Number of Variants generated for each number of for loops

four for loops, as shown in Code 5.2. The position of the teams distribute directive is always at loop position 0. If the parallel for directive is also at loop position 0, the teams distribute directive is combined with the parallel for directive, and thus the collapse for teams distribute directive does not exist. If the position of the parallel for directive is on loop position \(x\) (where \(1 \leq x \leq 3\)), then the maximum possible value of collapse for the teams distribute directive is \(x\). Whereas for the parallel for directive, the maximum possible value of collapse will be \((N - x)\), where \(N\) is the total number of for loops (in this example \(N = 4\)).

The scheduling type of the loop iteration could be one of static, dynamic or guided. Given these parameters, we calculated that for a given number of for loops, we could generate a variety of GPU offloading code variants, as shown in Table 5.1. Once all of the variants have been generated, we use our static cost model to predict the runtime of each of these generated kernels.

### 5.3.2 Cost Model

A compile-time cost model is required to select the best option from all the variants generated by the Kernel Analysis module. Most modern compilers employ analytical models to calculate the cost of executing the original code as well as the various offloading code variants. Building such an analytical model for compilers is a difficult task that necessitates a lot of effort on the part of a compiler engineer. Recently, machine learning techniques have been successfully applied to build cost models for a variety of compiler optimization problems. In our paper [94], we provided a proof of concept for the idea that compiler developers could train an ML model and use it to make better decisions in offloading a kernel to a GPU using OpenMP.
We presented COMPOFF, a cost model that statically estimates the Cost of OpenMP OFFloading using a neural network model. Our results show that this model can predict offloading costs with a root mean squared error in prediction of less than 0.5 seconds. COMPOFF is discussed in greater detail in Chapter 7. As soon as we know the prediction for the generated variant, we store it in the instance of the KernelInformation class so that the Kernel Transformation module can use it.

Apps marked † are adopted from the Rodinia Benchmark Suite.

<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Breadth First Search</strong> (bfs) [133]†</td>
<td>Graph Algorithms</td>
</tr>
<tr>
<td>Graph algorithms are basic and frequently employed in a variety of academic and professional fields. In scientific and technical applications, huge networks with millions of vertices are typical. This algorithm offers GPU implementations of the breadth-first search (BFS) technique, which iterates through all of a graph’s connected elements.</td>
<td></td>
</tr>
<tr>
<td><strong>Correlation Coefficient</strong> (correlation) [125]</td>
<td>Statistics</td>
</tr>
<tr>
<td>The Pearson’s Correlation Coefficient is a measurement of the linear correlation between two sets of data. It is essentially a normalized measurement of the covariance, with the result always falling between −1 and 1. It is the ratio between the covariance of two variables and the product of their standard deviations. The Pearson correlation coefficient is commonly used for data that is jointly normally distributed (follows a bivariate normal distribution).</td>
<td></td>
</tr>
<tr>
<td><strong>Covariance</strong> (covariance)</td>
<td>Probability Theory</td>
</tr>
<tr>
<td>The sample covariance are statistics calculated from a sample of data on one or more random variables. The sample covariance can be used to estimate the population covariance matrix and to assess the validity of sample means as estimators.</td>
<td></td>
</tr>
<tr>
<td><strong>Gaussian Elimination</strong> (gauss) [132]†</td>
<td>Linear Algebra</td>
</tr>
<tr>
<td>The Gauss Seidel method for solving linear equations is an iterative method, in which the values for the given variables keep changing until a certain threshold of variance is reached. Although the algorithm must synchronize between iterations, the values produced in each iteration can be computed in parallel.</td>
<td></td>
</tr>
<tr>
<td><strong>K-nearest neighbors (knn)</strong> [134]†</td>
<td><strong>Data Mining</strong></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>The k-nearest neighbors technique utilizes proximity to classify or predict how a single data point will be grouped. It is a non-parametric, supervised learning classifier.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Laplace (laplace)</strong> [71]</th>
<th><strong>Numerical Analysis</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>In physics, Laplace’s equation is a second-order partial differential equation. The solution of Laplace’s equation in two dimensions with finite differences can be approximated using a straightforward Jacobi iteration, according to numerical analysis texts.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Matrix-Matrix Multiplication (mm)</strong></th>
<th><strong>Linear Algebra</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A naive matrix-matrix multiplication. The most significant matrix operation is arguably matrix multiplication. It is extensively utilized in a variety of fields, including population modeling, the solution of linear systems of equations, translation of coordinate systems, and network theory.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Matrix-Vector Multiplication (mv)</strong></th>
<th><strong>Linear Algebra</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A naive matrix-vector multiplication. It should be noted that matrix-vector multiplication only applies when the length of the vector is equal to the number of columns in the matrix.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Matrix Transpose (mt)</strong></th>
<th><strong>Linear Algebra</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The transpose of a matrix is simply a flipped version of the original matrix. We can transpose a matrix by switching its rows with its columns.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Particle Filter (particle)</strong> [135]†</th>
<th><strong>Medical Imaging</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This algorithm belongs in the Medical Imaging domain. Given noisy measurements of the target’s location and an idea of the object’s movement in a Bayesian framework, the particle filter is a statistical estimator of the location of a target item. It can be used for a wide variety of applications, including video compression and video surveillance that tracks faces, cells, and automobiles.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Proxy App (proxy.app)</strong></th>
<th>-NA-</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a proxy app for our target kernel, the Wilson Dslash operator, and it uses a similar number of loops and computations. Whenever it is difficult to collect data on real applications, proxy apps help us collect more data.</td>
<td></td>
</tr>
</tbody>
</table>

---

Table 5.2: Benchmark Applications.
As soon as we know the prediction for the generated variant, we store it in the instance of the \texttt{KernelInformation} class so that the Kernel Transformation module can use it. But the biggest challenge in implementing an ML based cost model is the lack of available training data. To overcome this problem, we wrote additional benchmark applications and adopted some benchmarks from the Rodinia benchmark suite [19]. The goal is to include a broader class of benchmarks that would cover the spectrum of statistical simulation, linear algebra, data mining, etc. For this we developed applications tabulated in Table 5.2. More applications from various other domains will be added to this repository in the future.

5.3.3 Kernel Transformation

In the Kernel Transformation module we need to actually transform the original source code based on the analysis and predictions from the previous modules. For the given kernel, we generate every possible code variation in the Training mode. However, before we can generate code in Prediction mode, we must first address another crucial question. Which code should we generate? Should we only generated code for the fastest kernel? Regrettably, once the directives are in place, neither the Advisor nor OpenMP validate the kernel’s correctness. Its a different issue, and falls outside the purview of the current research. This is inline with the OpenMP philosophy as well. As a result, we can only guarantee the correctness of the generated OpenMP directive in our framework.

So how can we overcome this problem? We could generate code for every possible variation, as we do during training, and let the user choose which one to use. But this means that users will be overwhelmed with information. Alternatively, we could ask the user to provide a number for the maximum number of codes to generate. The predicted runtime can be put as a comment before the kernel in every piece of code. The application scientist will then have more power to accept or reject the generated code. While we acknowledge that this is not the most desirable solution, it does a great job of supporting the proof of our concept. We will be able to produce a single code and provide it to the user once the issue of validating an OpenMP code for correctness is resolved. Until then, our Advisor will be able to generate the top best variants as specified by the application scientist.

Regardless, we need to write a module to modify the existing source code and generate a new code. It is well known that source-to-source transfor-
mation of C/C++ code is a challenging problem. Clang was not a popular compiler option for source-to-source transformation until recently. To accomplish this, developers once used tools like the Rose compiler [117]. However, the recent development of Clang as a potent and library-friendly C++ compiler demonstrates that there may finally be some relief in sight.

Clang provides the Rewriter interface, whose primary function is to route high-level requests to the involved low-level RewriteBuffers. A Rewriter assists us in managing the code rewriting task. In the Rewriter interface we can set the SourceManager object which handles loading and caching of source files into memory. This object assigns distinct FileIDs to each distinct #include chain and is the owner of the MemoryBuffer objects for all of the loaded files. The SourceManager can be queried to obtain information about SourceLocation objects, which can then be converted into spelling or expansion locations. Spelling locations represent the bytes that correspond to a token, while expansion locations represent the location in the source of the code. For example, in the case of a macro expansion, the spelling location indicates where the expanded token originated, while the expansion location specifies where it was expanded.

The Rewriter is a critical component of our plugin’s kernel transformation module. The strategy used here is to meticulously alter the original code at crucial locations to carry out the transformation rather than handling every possible AST node to spit back code from the AST. For this, the Rewriter interface is essential. It is an advanced buffer manager that effectively slices and dices the source using a rope data structure. For each SourceManager, the Rewriter also stores the low-level RewriteBuffer. Together with Clang’s excellent retention of source location for all AST nodes, Rewriter makes it possible to remove and insert code very precisely in the RewriteBuffer. When the update is finished, we can dump the RewriteBuffer into a file to obtain the updated source code.

The instance of KernelInformation contains information about the kernel, such as its start and end location, the start and end location of each nested for loop, the data it uses, the various kernel variants that can be made for it, and the predicted runtime for each of these variants. In the Kernel Transformation module, we create a vector of seven strings. The location of these seven strings are shown in Code 5.3. The first string (at #0) is always the comment that maintains the text “Predicted Runtime: ## s”. As the text suggests ## is the predicted runtime for this particular kernel variant. This string is always placed before the kernel’s start location.
Then comes the target enter data construct (at #1). This directive handles what memory on the GPU needs to be created for the kernel and what data needs to be sent to the GPU before execution. This string is always placed right after the comments string. The next string (at #2) contains the OpenMP directive which specifies that this is the kernel to offload to the target. To gain maximum performance out of the GPU, we should always distribute the kernel across all the teams available in the GPU. Hence this string always contain the directive — “#pragma omp target teams distribute”. The variant determines whether this directive contains any other clauses such as collapse or map, and what will be the values of the clause. This string is always placed immediately before the kernel’s start location, but after the target enter data string. The remaining strings (#3, #4 and #5 if required by the variant) are placed just before the start location of their nested for loop. If these strings are not needed by a variant, they are left empty and no code is inserted in their location. The last string (at #6) is the target exit data construct, which identifies the data that must be released from the GPU or returned to the CPU. If not empty, each of these strings is always terminated by a new line. Once these seven strings are placed in their respective positions, the code is dumped into a new C++ file, which is
returned to the application developer. The kernel runtime provided in the comment can help the application developer select the best code. They could choose to select or reject the transformations and based on their selection, they could select the file to build and link with their code build.
Chapter 6
Data Reuse

In the high performance computing industry, researchers and developers expend considerable effort to port their applications to GPU-based clusters in order to take advantage of the massively parallel architecture and energy efficiency of a GPU. Unfortunately porting or writing an application for accelerators, such as GPUs, requires extensive knowledge of the underlying architectures, the application/algorithm and the interfacing programming model, such as CUDA or OpenMP. Compared to native GPU programming models, OpenMP has a shorter learning curve, is portable and potentially also performance portable. OpenMP also provides implicit data transfer between CPU and GPU, taking the burden off the developers. OpenMP users may control the duration of a data object’s allocation on the GPU via the use of target data regions, but they do not need to. Unfortunately, unless this is explicitly provided by the user, compilers like Clang move all data without considering its availability on the device. As a result, applications may spend a significant portion of their execution time on data transfer. Exploiting data reuse opportunities in an application can reduce data transfer time and ultimately its overall execution time. In this paper we present an optimization to automatically recognize data which do not need to be transferred between CPU and GPU, and capitalize on any data reuse opportunities in an application. We analyze applications which use OpenMP for exploiting GPU parallelism, and insert the pertinent code to take advantage of data reuse on GPU. Our optimization enabled us to transfer data when required and retain reused data on the GPU, thereby reducing the overall execution time of our micro-benchmarks and some benchmark applications from the Rodinia benchmark suite.
6.1 Introduction

GPUs are well known for their massively parallel architectures, as well as exceptional performance and energy efficiency for suitable codes. Supercomputing clusters like Summit \([145]\) derive the lion’s share of their compute power from GPUs. Each node of Summit is configured with 22 CPU cores and 6 NVIDIA GPUs. Just one NVIDIA Tesla V100 GPU consists of over 5000 CUDA cores and can easily outperform the most powerful CPUs for many tasks in parallel processing. For the last two decades, GPUs are the preferred accelerator in the high performance computing (HPC) industry, where they serve as a co-processor to accelerate general-purpose scientific and engineering application codes. Today, they expedite computational workloads in cutting edge scientific research in diverse areas such as Physics, Bioinformatics, Chemistry, Climate Modeling, Machine Learning, and much more. GPUs are used in a variety of area, such as embedded systems, mobile devices, workstations, and gaming consoles.

Due to their massive parallelism, they can dwarf the calculation rate of even the most powerful CPU for many parallel processing tasks. The same shader cores that allow multiple pixels to be rendered simultaneously can similarly process multiple streams of data at the same time. Hence, researchers and developers increasingly desire to port their applications to a GPU-based clusters. Unfortunately porting or writing applications for a GPU requires extensive knowledge of the underlying architecture, the application/algorithm and the interfacing programming model. One approach to reduce this effort of the developers is to use a Directive-based programming models, such as OpenMP, may reduce this effort. Compared to native GPU programming models such as CUDA \([104]\) and HiP \([18]\), using OpenMP for GPU programming implies a shorter learning curve, program portability and potentially performance portability. In order to leverage accelerators, OpenMP 4.0 introduced the ability to offload computations to attached accelerator devices \([107]\) (GPUs, FPGAs, DSPs, etc.). Features for offloading to devices are under active improvement, as are the implementations \([108]\) in numerous compilers, including Clang/LLVM \([84]\), GCC \([38]\), Intel \([59]\), Cray \([26]\), and IBM XL \([57]\).

OpenMP is an attractive approach due to its productivity benefits. An advantage that OpenMP provides over most native GPU programming mod-
els is that it enables implicit data transfer for GPU kernels\textsuperscript{1}. Users can choose whether or not to handle data transfer explicitly; in the latter case, the OpenMP compiler will manage the data motion. Unfortunately, the compiler support is in need of improvement. For example, the Clang compiler currently transfers all data which is used in the kernel both to and from the device, even if some data is already available on the device. Given the relatively high cost of transfers, applications may thus spend a significant portion of their execution time on data movement as a result. In this research we present an automatic approach to optimize data transfer between CPU and GPU by capitalizing on any data reuse opportunities in an application. It is worth noting that this research only considers traditional data management and not data management through Unified Memory [77, 96]. More details about our approach can be found in Section 6.3.

The rest of this paper is organized as follows: Section 6.2 presents the GPU programming model and introduces the problem of data transfer in OpenMP. Section 6.4 describes the experimental setup for our research. Section 6.5 and 6.6 provide and analyze the results that we obtained out of this research. We conclude the work with discussion about probable usage and our future work in Section 6.7.

6.2 GPU Programming Model

In the HPC context, GPUs are generally considered to be auxiliary processors that are attached to a CPU. Following this convention, a CPU is referred to as a host in this paper, whereas a GPU is referred to as a device. A block of code which will run on the device is called a kernel. Both CPU and GPU have their own separate dynamic memories. It is imperative to assure that all necessary data is present on the GPU before we can run a kernel on the device. As a result, we must first allocate the necessary amount of memory on the GPU and then transfer all data that has to be accessed by the GPU from the host to the device. These data must be transferred back to the host for additional usage or storage after the computation on the GPU is completed. This expensive transfer of data between the host and the device is a major bottleneck in GPU computing.

Generally all kernel computations on a GPU follow this basic pattern \textsuperscript{2}:

\textsuperscript{1}In this paper the term kernel is always used in reference to a GPU kernel
\textsuperscript{2}This is traditional GPU computing. This research did not consider Unified Memory.
Allocate: Space is allocated on the GPU for all the data required to perform the computations in the kernel.

HostToDevice: Data is copied from the host to the device as needed.

Compute: The kernel computations are carried out on the device.

DeviceToHost: Upon completion of kernel computations, data from the device is copied to the host.

Free: Finally, the allocated device memory is freed.

Since the core logic of the program is written in the kernel, most application developers expend considerable time and effort to optimize it. Handling data between a CPU and GPU is quite burdensome and just an added hassle for them. Many codes do not exploit the significant performance gains obtainable from utilizing the GPU because of the challenges involved in creating this code. Extracting any sort of near full-scale performance on a GPU is no walk in the park and using it comes with its own set of challenges:

- There are hundreds of thousands of CPU programming experts, but very few GPU experts. It may be hard to get help for developing and debugging complex code.
- There are many sources of potential performance problems in GPU code, including the manner in which different device memories are accessed, operations like coalesced read/write, divergent branches, local loads and stores, and much more.
- Explicit memory management between CPU and GPU is a serious impediment to using GPUs for high performance applications.
- The code is not suitable for the GPU, perhaps because it has too few computations, excessive control flow, or little concurrency. The time taken to transfer data and perform the calculation may be longer than the CPU needs to compute it.
- GPU programming can be quite cumbersome and time consuming, which is a major deterrent in practice.

6.2.1 OpenMP

OpenMP is a directive-based application programming interface that can be used in a Fortran, C or C++ application code to create a parallel program. It is designed for portability, enjoys wide vendor support, and compared to native programming models, like CUDA or HIP, offers a much faster rate of
learning for developers. From version 4.0 onward, OpenMP supports accelerator devices in a host-device model, for which it offers “target offloading” features. The specification provides several options to allow its users to control the duration of a data object’s allocation on the GPU, via the use of target data regions.

To further simplify the application development effort, OpenMP provides the option of choosing implicit data transfer for GPU kernels. A number of vendor compilers and open source implementations (gcc and Clang) support OpenMP’s GPU offloading features and are actively extending and enhancing their support in this respect. While still under development, Clang handles an implicit data transfer by moving all data needed for kernel computation without considering its availability on the device. Thus all data used in the kernel are transferred both to and from the device. In consequence, applications may spend a significant portion of their execution time on data transfer itself. Automatically identifying and exploiting GPU data reuse opportunities in an application can lead to better performance by reducing the amount of data transferred and hence the overall execution time. Such an optimization could avoid the performance penalty currently associated with the implicit transfer approach and ultimately help increase the number of codes that can utilize a GPU without extensive developer effort.

6.3 Data Transfer & Reuse

In this research we present an approach for automatically recognizing data reuse opportunities in an application that uses OpenMP for exploiting GPUs parallelism. It also identifies the data that need to be transferred between the CPU and GPU. Based upon this analysis, it inserts the pertinent OpenMP target data directives into the original source code to precisely manage data transfers between the host and the device.

When an OpenMP program begins, an implicit target data region for each device surrounds the whole program. Each device has a device data environment that is defined by its implicit target data region. Any declare target directives and the directives that accept data-mapping attribute clauses determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment. If a user chooses not to map any data to the device explicitly, then data is implicitly managed by the compiler. The Clang compiler simply identifies all variables used in a target
region and moves any data associated with them to the device. It does not take into consideration whether or not the data is already available on the device. Once the kernel execution is over, it again moves all data back to the host, irrespective of whether or not the data was updated on the kernel or needed beyond the kernel. This might not be the most efficient approach to handle data, but in its early stage it gets the job done accurately.

To make the performance better we need to understand when and what data is needed to transfer between the two devices. For better data management, we should have a better understanding of when and what data is needed on the CPU and the GPU. Figure 6.1 illustrates how our Clang optimization accurately identifies data which need to be moved between host and device. First we parse the AST (Section 6.3.1) and identify the kernels (Section 6.3.2) in the application. For each such kernel, we analyze it to identify all data (Section 6.3.3) which are used and store this information in the Kernel Information Table. Next we check whether the kernel is called from within a loop and check for proximity of kernel. Once kernels are identified to be in close proximity to each other, we analyze them to find common data which can be reused between multiple kernels. After identifying common data between kernels, we update the Data Reuse Table (Section 6.3.4) to record the location of the kernels, what data is being reused and information about loop and proximity. Finally with the help of our analysis, we update the original source code (Section 6.3.5) to insert the pertinent `target data map` directive to transfer data between the two devices.

### 6.3.1 Clang AST Parsing

Our goal is to modify the original source code by inserting directives that precisely specify the required data movement between host and device. The resulting source code can then be compiled and executed on a GPU-based cluster. The goal of our research is to update the original source code to support explicit data movement between host and device, and return a code that can further be compiled for and executed on a GPU based cluster. Our choice of compiler to implement this optimization is Clang, which supports source to source translation, via libTooling [22], even though it is not primarily used for this purpose. In the Clang/LLVM programming model [75] most of the analysis is done on the LLVM Intermediate Representation (IR) and not on the Abstract Syntax Tree (AST). But once LLVM applies its optimizations to the IR, it generally loses information about the original source code, mak-
6.3.2 Kernel Identification

The applications that we are targeting already uses OpenMP for exploiting GPU parallelism. For the purpose of this research we only target applications that uses OpenMP for exploiting GPU parallelism. To identify kernels, we parse the AST to search for all `omp target` pragmas in the source code. When we encounter an `omp target` pragma we issue a unique id to the kernel and add an entry for it in our Kernel Information Table. We also save the `start` and `end` location of the kernel, the function from which the kernel is called and loop (if any) from within which the kernel is called. This information will further be used to identify variables accessed inside the
6.3.3 Data Analysis

We implement a live variable analysis upon the Clang AST, focusing only on variables used inside a kernel. While traversing the AST we store information about the source code location related to all the variables declared & accessed. We analyze this information to classify data used inside kernels into 5 groups:

1. **alloc**: These are variables assigned inside the kernel for the first time. Data which falls under this category need not be transferred from the host to the device. During code generation these data are mapped with the map type “alloc”.

2. **to**: These are variables assigned before but accessed inside the kernel. Data which falls under this category need to be transferred from the host to the device. During code generation these data are mapped with the map type “to”.

3. **from**: These are variables that are updated inside the kernel and accessed after the kernel call. Data which falls under this category need to be transferred from the device to the host. During code generation these data are mapped with the map type “from”.

4. **tofrom**: These are variables that are assigned before a kernel call, updated inside it and accessed after the kernel execution is complete. Data which falls under this category need to be transferred both ways between the host and the device. During code generation these data are mapped with the map type “tofrom”.

5. **private**: Finally we have variables which are defined and used only inside the kernel. Data which falls under this category does not need to be transferred between the host and the device. During code generation these data are not mapped.

The results of this analysis are stored in our Kernel Information Table to be used during “common data” analysis (explained in sub-section 6.3.3), and while generating the source code (explained in sub-section 6.3.5). Based upon the classification of data, appropriate code is generated for them to facilitate proper data transfer between host and device.
Loop Check

If a kernel is called from inside a loop (we consider both for and while loops, as shown in Code 6.1), there is a high probability that data is reused in multiple calls to that kernel. We thus mark any data used inside the kernel as a potential candidate for reuse. This information will be exploited when we check for common data (see Section 6.3.3) on the kernel. If a variable is not assigned before the start of the loop we simply allocate the variable on the device. This way all subsequent calls to the kernel will use the same memory on the device instead of allocating new memory every time the kernel is called from inside the loop.

Proximity Check

We define two kernels to be in close proximity to each other if both of them are called from the same function (as shown in Code 6.2). In this case, if they
also use the same variables, there is a high possibility of data reuse between
them. All kernels called within a loop are, by default, considered to be in
close proximity to each other.

Common Data Check

We use pattern matching on all variables used inside the sets of kernels
identified as being in close proximity to each other to check for potential
data reuse. When we find variables that are common to any pair of kernels
we check whether that variable is accessed on the host or not. Next we check
if the matched data is modified on the host in-between kernel calls, i.e. after
the first and before the second kernel is called. If data is modified on the
host, it is not reused and hence data transfer to/from the CPU is required.
If a kernel is called from within a loop, all subsequent calls to the kernel are
analyzed as new kernel calls.

6.3.4 Data Reuse Table

Once common data are identified, we update the Data Reuse Table with
information on the location of the kernels, on what data is being reused,
etc. For example, the snippet shown in Code 6.3 has two kernels which are
called from within a loop. Both kernels use 3 earrays – v1, v2 and v3 and
a variable err. The kernels are considered to be in close proximity to each
other. Since array v1 is assigned inside Kernel 1 for the first time, there is
no need to transfer the data for this array to the GPU and we can allocate
memory for array v1 on the GPU. Given that the same array v1 is used on
the host, we need to transfer it back to the host after Kernel 1 is executed.
Array v2 is used both in Kernel 1 and on the host, but never modified in any
kernel. Hence we need to transfer v2 only once before the start of the loop
and its data can be used by all the kernels and the host. On the other hand,
array v3 is assigned inside Kernel 2, but never used on the host. Since array
v3 is already on the GPU, we do not need to transfer the data anywhere.
A single variable err is modified in Kernel 2 and used on host and Kernel
1, so this data need to be transferred to the device before calling Kernel 1
and from the device after executing Kernel 2 every time. All the relevant
information is stored in the Data Reuse Table, which is later consulted to
modify the source code accordingly.
func1(v1[], v2[], v3[]) {
    // assign variable err;
    #pragma omp target data map(alloc:v1) map(to:v2, v3)
    while(err) {
        #pragma omp target data map(from: v1) map(to: err)
        #pragma omp target ...
        { // Kernel 1 using array v1, v2, v3 and variable err
            // assign array v1;
            // access array v2 and v3;
            // access variable err
        }
        // access array v1, v2
        #pragma omp target data map(from: err)
        #pragma omp target ...
        {
            // Kernel 2 using array v1, v2, v3 and variable err
            // update array v3
            // access array v1 and v2
            // update variable err
        }
    }
}

Code 6.3: Example snippet of Data Check. Data Transfer code in red.

6.3.5 Updating Source Code

Finally once the analysis is complete and all information has been collected, we have two tables – the Kernel Information Table and Data Reuse Table – that enable us to determine what data is needed by which kernel, and what data is reused between multiple kernels. This allows us to generate and insert the pertinent OpenMP target data directive in the source code to meticulously manage data transfers between the host and the device.
6.4 Experimental Setup

To implement our optimization we used Clang/LLVM version 8.0. To evaluate our benchmarks we use the computational cluster at Stony Brook University – SeaWulf [144]. We ran our optimization on 4 micro benchmark application and 6 benchmark application defined in the rodinia benchmark suite [19].

The micro-benchmarks implemented by us are as follows:

1. **Matrix Multiplication (3mm)** - This is the most basic implementation of multiplying three large matrices. In this implementation we are executing 2 kernels. The first kernel multiplies 2 matrices and stores the result in an intermediate matrix, which is multiplied with the third matrix in the second kernel to get the resulting kernel. This is a benchmark where two kernels are reusing same data. For our experiment we used matrices of size $5000 \times 5000$ each.

2. **Gauss Seidel Method (gauss)** - The Gauss Seidel method for solving linear equations is an iterative method, in which the values for the given variables keep changing until a certain threshold of variance is reached. This is a benchmark where one kernel is called multiple times from within a loop. The experiment used a matrix of size $2^{13} \times 2^{13}$.

3. **Laplace Equation (laplace)** - The Laplace equation in two dimensions with finite differences using jacobi iteration. This is a benchmark where two kernels, reusing data, are called multiple times from within a loop. The experiment used a matrix of size $2000 \times 2000$.

4. **Single-Precision A·X Plus Y (saxpy)** - SAXPY is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. SAXPY is a combination of scalar multiplication and vector addition, and it takes as input two vectors of 32-bit floats $X$ and $Y$, and a scalar value $A$. It multiplies each element $X$ by $A$ and adds the result to $Y$. In its simplest form this is a benchmark where two kernels are reusing same data. The experiment used two vectors of size $2^{27}$ each.

We also used our approach on some applications from the Rodinia Benchmark Suite from different domains:
1. **Breadth First Search (bfs)** - BFS belongs in the Graph Algorithm domain. Graph algorithms are fundamental and widely used in many disciplines and application areas. Large graphs involving millions of vertices are common in scientific and engineering applications. This benchmark provides the GPU implementations of BFS algorithm which traverses all the connected components in a graph.

2. **HotSpot** - HotSpot belongs in the Physics Simulation domain. HotSpot is a widely used tool to estimate processor temperature based on an architectural floorplan and simulated power measurements. The simulation iteratively solves a series of differential equations for block. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. We re-implemented the transient differential equation solver from HotSpot using target offloading directives for GPU.

3. **k-Nearest Neighbor (knn)** - This algorithm belongs in the Data Mining domain. It finds the k-nearest neighbors from an unstructured data set. The sequential nearest neighbor algorithm reads in one record at a time, calculates the Euclidean distance from the target latitude and longitude, and evaluates the k nearest neighbors. The parallel versions read in many records at a time, execute the distance calculation on multiple threads, and the master thread updates the list of nearest neighbors.

4. **LU Decomposition (lud)** - This algorithm belongs in the Linear Algebra domain. LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The lud kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This is a good example of a benchmark where multiple kernels called from within a loop and some data shared by these kernels are also used on the host.

5. **Needleman Wunsch (nw)** - Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments, which belongs in the Bioinformatics domain. The potential pairs of sequences are organized in a 2D matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. The optimum alignment is the pathway through the array with maximum score, where the score
Table 6.1: Data Management in Optimized Code vs Base Code for benchmark applications

<table>
<thead>
<tr>
<th>Benchmark Application</th>
<th>Data Transfer (MB)</th>
<th>Alloc</th>
<th>Number of calls to</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Free</td>
<td>HtoD</td>
<td>DtoH</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>2289</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Base</td>
<td>763</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Gauss Seidel Method</td>
<td>6464</td>
<td>200</td>
<td>200</td>
<td>201</td>
</tr>
<tr>
<td>Base</td>
<td>64</td>
<td>101</td>
<td>101</td>
<td>102</td>
</tr>
<tr>
<td>Laplace Equation</td>
<td>1220703</td>
<td>25000</td>
<td>25000</td>
<td>25001</td>
</tr>
<tr>
<td>Base</td>
<td>61</td>
<td>5004</td>
<td>5004</td>
<td>5005</td>
</tr>
<tr>
<td>Saxpy</td>
<td>2048</td>
<td>7</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Base</td>
<td>1024</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Breadth First Search</td>
<td>8392</td>
<td>12</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>Base</td>
<td>4387</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>HotSpot</td>
<td>20480</td>
<td>48</td>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>Base</td>
<td>3072</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>k-Nearest Neighbor</td>
<td>10887</td>
<td>171800</td>
<td>171800</td>
<td>171801</td>
</tr>
<tr>
<td>Base</td>
<td>819</td>
<td>42953</td>
<td>42953</td>
<td>42953</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>5718994</td>
<td>7496</td>
<td>7496</td>
<td>7497</td>
</tr>
<tr>
<td>Base</td>
<td>1525</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Needleman Wunsch</td>
<td>27467</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Base</td>
<td>10300</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Particle Filter</td>
<td>808</td>
<td>131</td>
<td>131</td>
<td>132</td>
</tr>
<tr>
<td>Base</td>
<td>248</td>
<td>49</td>
<td>49</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 6.1: Data Management in Optimized Code vs Base Code for benchmark applications

is the value of the maximum weighted path ending at that cell. Thus, the value of each data element depends on the values of its northwesterly, northerly and westerly adjacent elements. In the second step, the maximum path is traced backward to deduce the optimal alignment.

6. **Particle Filter (p-filter)** - This algorithm belongs in the Medical Imaging domain. The particle filter is a statistical estimator of the location of a target object given noisy measurements of that target’s location and an idea of the object’s path in a Bayesian framework. The p-filter has a plethora of applications ranging from video surveillance in the form of tracking vehicles, cells and faces to video compression. This particular implementation is optimized for tracking cells, particularly leukocytes and myocardial cells.

For each application we ran two scenarios:

1. **Base** – which is the basic code without our optimization.
2. **Optimized** – in this case we first ran our optimization upon the base case to get the optimized code.
For each of these scenarios we ran the benchmark 10 times and collect information such as how much data is transferred and how much is the total run time.

In its current version, OpenMP in Clang uses CUDA to implement GPU offloading. So it is imperative that we analyze how many times CUDA APIs related to data transfer are called. As such we also collect the data of how many times the following CUDA APIs are called:

- **cuMemAlloc** - Allocates bytes of linear memory on the device and returns a pointer to the allocated memory.
- **cuMemFree** - Frees the memory space which must have been returned by a previous call to cuMemAlloc.
- **cuMemcpyHtoD** - Synchronous copies the specified amount of data from host memory to device memory.
- **cuMemcpyDtoH** - Synchronously copies the specified amount of data from device memory to host memory.

We ran these experiments on 3 different GPUs — NVIDIA Tesla K80, NVIDIA Tesla P100 and NVIDIA Tesla V100, all three using a PCI-e connector between the CPU and GPU.

### 6.5 Results

Code 6.4 gives a sample output of our optimization when applied to a code multiplying 3 matrices. Here the code marked in red is auto-generated by our optimization. In this particular example, two kernels use, and reuse, data from the array `temp`. Arrays `A`, `B` and `C` only need to be transferred to the device. Since they are not updated on the GPU, we do not need to transfer their values back to the host. Array `temp` is needed only on the GPU, while array `D` needs to be returned to the host. The array `temp` is assigned on the GPU, so we do not need to transfer their data from the host to the device.

After applying our optimization to all the benchmark applications, we collect the amount of data transferred for the base and optimized code and found that in all cases, the amount of data transferred was lower in the optimized code than in the base code. Table 6.1 tabulates the total
data transferred between CPU and GPU in MB. It can clearly be seen that in most applications the amount of data transferred decreased drastically after applying our optimization. Table 6.1 also shows how many times CUDA APIs such as cuMemAlloc (Alloc), cuMemFree (Free), cuMemcpyHtoD (HtoD) and cuMemcpyDtoH (DtoH) were called. These data remain common for all three GPUs.

After analyzing the difference in data transfers between the two codes, we calculated the time it took to perform the data transfer and the time taken
Table 6.2: Result comparing Base code with Optimized code for time taken by different APIs when run on NVIDIA Tesla K80

<table>
<thead>
<tr>
<th>Benchmark Application</th>
<th>Alloc (ms)</th>
<th>Free (ms)</th>
<th>HtoD (ms)</th>
<th>DtoH (ms)</th>
<th>Data (sec)</th>
<th>Kernel (sec)</th>
<th>% Data to Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>30.122</td>
<td>4.202</td>
<td>5.001</td>
<td>201.230</td>
<td>325.390</td>
<td>109.460</td>
<td>308.140</td>
</tr>
<tr>
<td>Gauss Seidel Method</td>
<td>925.910</td>
<td>923.060</td>
<td>134.000</td>
<td>134.220</td>
<td>1360.460</td>
<td>1361.130</td>
<td>1322.040</td>
</tr>
<tr>
<td>Laplace Equation</td>
<td>30106.700</td>
<td>3245.210</td>
<td>29922.300</td>
<td>2806.840</td>
<td>91088.700</td>
<td>90683.700</td>
<td>241.921</td>
</tr>
<tr>
<td>Breadth First Search</td>
<td>16.589</td>
<td>15.070</td>
<td>698.900</td>
<td>1161.910</td>
<td>1156.410</td>
<td>3.034</td>
<td>314.577</td>
</tr>
<tr>
<td>HotSpot</td>
<td>977.660</td>
<td>8.244</td>
<td>581.920</td>
<td>3014.380</td>
<td>3175.820</td>
<td>7.750</td>
<td>42.601</td>
</tr>
<tr>
<td>k-Nearest Neighbor</td>
<td>57466.100</td>
<td>19727.500</td>
<td>52508.500</td>
<td>1759.400</td>
<td>1912.960</td>
<td>1988.770</td>
<td>113.876</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>11806.800</td>
<td>3.698</td>
<td>76233.000</td>
<td>204.380</td>
<td>48153.000</td>
<td>82025.000</td>
<td>2442.035</td>
</tr>
<tr>
<td>Particle Filter</td>
<td>91.023</td>
<td>29.178</td>
<td>68.359</td>
<td>154.350</td>
<td>166.170</td>
<td>0.480</td>
<td>2.557</td>
</tr>
</tbody>
</table>

Table 6.3: Result comparing Base code with Optimized code for time taken by different APIs when run on NVIDIA Tesla P100

<table>
<thead>
<tr>
<th>Benchmark Application</th>
<th>Alloc (ms)</th>
<th>Free (ms)</th>
<th>HtoD (ms)</th>
<th>DtoH (ms)</th>
<th>Data (sec)</th>
<th>Kernel (sec)</th>
<th>% Data to Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>5.536</td>
<td>2.104</td>
<td>3.060</td>
<td>26.548</td>
<td>263.800</td>
<td>74.821</td>
<td>35.565</td>
</tr>
<tr>
<td>Gauss Seidel Method</td>
<td>271.280</td>
<td>35.367</td>
<td>65.377</td>
<td>177.449</td>
<td>1156.860</td>
<td>0.010</td>
<td>39.206</td>
</tr>
<tr>
<td>Laplace Equation</td>
<td>10833.600</td>
<td>1447.590</td>
<td>9514.790</td>
<td>14655.900</td>
<td>51862.500</td>
<td>137602</td>
<td>9.150</td>
</tr>
<tr>
<td>SAXPY</td>
<td>11.867</td>
<td>1.496</td>
<td>9.925</td>
<td>52.402</td>
<td>457.360</td>
<td>1.287</td>
<td>1.093</td>
</tr>
<tr>
<td>Breadth First Search</td>
<td>23.023</td>
<td>8.208</td>
<td>18.472</td>
<td>230.467</td>
<td>1479.820</td>
<td>3.304</td>
<td>68.579</td>
</tr>
<tr>
<td>k-Nearest Neighbor</td>
<td>22910.900</td>
<td>7932.470</td>
<td>16649.400</td>
<td>5416.200</td>
<td>1053.190</td>
<td>41.735</td>
<td>42855.92</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>39214.800</td>
<td>1.787</td>
<td>19999.200</td>
<td>48301.000</td>
<td>512812.000</td>
<td>1327785</td>
<td>308671</td>
</tr>
<tr>
<td>Particle Filter</td>
<td>43.537</td>
<td>13.766</td>
<td>28.921</td>
<td>23.045</td>
<td>134.700</td>
<td>0.511</td>
<td>0.595</td>
</tr>
</tbody>
</table>
Table 6.4: Result comparing Base code with Optimized code for time taken by different APIs when run on NVIDIA Tesla V100 to run the kernel. Table 6.2, 6.3 and 6.4 tabulates the time taken by each of these APIs and the time taken to compute the kernels in an application when run on NVIDIA Tesla K80, P100 and V100 GPUs respectively. The time taken by the 4 CUDA APIs is in milliseconds, while the time taken by the kernel is in seconds. For compute intensive applications, such as Matrix Multiplication, Gauss Seidel Method and Breadth First Search, the computation time exceeds the data transfer time. But for other applications, the two times are close to each other.

6.6 Analysis

Let us first analyze the amount of data transferred, in Table 6.1. For our Matrix Multiplication micro-benchmark, in the base case, more than 2 GB of data is transferred between the host and device. In our optimized code, only 763 MB of data is transferred. As can be observed in Figure 6.2(b) there is a reduction of 66.67% in data transfer, accomplished by automatically adding three lines of OpenMP target data directives to explicitly manage data transfer between the host and device (Code 6.4).
For Rodinia’s *LU-Decomposition* benchmark, in the base case 5.4 TB (TeraByte!) of data is transferred between the host and the device, as compared to 1.5 GB in the optimized code. This is a huge reduction (99.97%) as can be seen in Figure 6.2(a). The *Laplace Equation* micro-benchmark transfers 1.2 TB of data in the base case, in comparison to only 61 MB in the optimized code (99.99% reduction). As evident from Figure 6.2(b) for these benchmark applications we observed a tremendous reduction in the amount of data transferred between the host and device, with Breadth First Search being the lowest at 47.7%.

Let us look at the number of times the data transfer APIs are called for each application (Table 6.1). In its base case, *Laplace Equation* calls `cuMemAlloc` 25000 times, `cuMemFree` 25000 times, `cuMemcpyHtoD` 25000 times and `cuMemcpyDtoH` 25001 times. Upon further analysis of the base code, we discovered that it was calling a kernel from inside a loop which iterated for 5000 times. There were 4 arrays and 1 variable which were used inside the kernel. Only the variable was used both on the host and the device.

In our optimized code, the data transfer for the arrays was moved outside
the loop, drastically reducing the number of times the data was moved between the host and device. In this case cuMemAlloc was called 5004 times, cuMemFree 5004 times, cuMemcpyHtoD 5005 times and cuMemcpyDtoH 5001 times. To achieve this, our optimization added just one target data directive. Figure 6.3 shows that there is an enormous reduction in the number of times the APIs are called, with maximum of 99.96% reduction in LU-Decomposition and minimum of 45.71% reduction in Needleman Wunsch. But does this reduction in the number of calls to the APIs have any effect on the total execution time? To find out we examine Table 6.2, 6.3 and 6.4.

These tables provide the time taken by each of these APIs for both cases on NVIDIA Tesla K80, P100 and V100 GPUs respectively. Let us first look at the time taken on V100 (Table 6.4). In this table, the time taken by the four CUDA APIs is saved in milliseconds, while the time taken by the kernel is saved in seconds. From the last column of the table, which represents the time taken to compute the kernel, we observe there is no improvement/deterioration in the computation time of the kernel after our optimization. The differences are only in the time taken by the 4 APIs used to manage

<table>
<thead>
<tr>
<th></th>
<th>cuMemAlloc</th>
<th>cuMemFree</th>
<th>cuMemcpyHtoD</th>
<th>cuMemcpyDtoH</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) 3mm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) gauss</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) laplace</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d) saxpy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e) bfs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f) hotspot</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(g) k-nn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h) lud</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i) nw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(j) p-filter</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3: Number of calls to CUDA APIs for Data Management
data. For our micro-benchmark application, Matrix Multiplication, the base case takes 0.548ms for cuMemAlloc, 0.574ms for cuMemFree, 26.548ms for cuMemcpyHtoD and 32.754ms for cuMemcpyDtoH. Overall, for a 10.17s kernel it took only 60.424ms in data transfer, which is only 0.59% of the compute time. Even with our optimized code, the total data transfer time is 27.485s for a 10.17s kernel, which is only 0.27%. The overall benefit amounted to ca. 33ms, which is insignificant when compared to its overall runtime. We observe similar pattern in Gauss Seidel Method. If we only consider the % improvement in data management, then we observe a total of 96% improvement (376.78ms in base case vs 11.91ms in optimized case). But when compared with the computation time of the kernel (approx 38.7s) it is only a 0.9% overall improvement. Similarly Breadth First Search only has a 0.67% improvement when our optimization is applied.

In Figures 6.4(a), 6.4(b), 6.4(e), we observe that the majority of time is consumed in the kernel computation rather than data management.

K-Nearest Neighbor is the other extreme. It takes 4.382s in cuMemAlloc,
Figure 6.5: % Improvement in Compute time and Data Management time on NVIDIA Tesla P100 and NVIDIA Tesla V100, keeping NVIDIA Tesla K80 as baseline.

2.958s in cuMemFree, 0.209s in cuMemcpyHtoD and 0.193s in cuMemcpyDtoH, with overall 7.743s for data management, almost 91x the kernel computation time of 0.085s. After applying our optimization, data management was improved by 71% to just 2.26s. But even after our optimization, data management is about 25x longer than kernel computation. In Figure 6.4(g) we do not even see the compute bar as it is insignificant compared to the data management time.

For LU-Decomposition, the base case requires 282.514s for data management for a kernel compute time of 173.441s, which is almost 1.63x the computation time. But after our optimization, it consumes only 94.28ms, which is a 99.96% improvement over the base case. In Figure 6.4(h) a significant overall improvement can be observed.

Similarly, for Laplace Equation we see a total of 359.44s of data man-
agement time for the base case compared to 0.3s of data management time for the optimized case, a total improvement of 99.01%. Given the kernel compute time of about 15s, our optimization provides a significant performance improvement. We also observe considerable improvement in SAXPY (99.54%), HotSpot (85.95%), Needleman Wunsch (49.3%) and Particle Filter (83.96%). There is a similar improvement for the NVIDIA Tesla K80 (Table 6.2) and P100 (Table 6.3) GPUs. In no case do we see any loss of performance after applying our optimization.

Amongst the three NVIDIA Tesla GPUs, computation on K80 is the slowest while that on V100 is the fastest. This is expected, as V100 is the latest and the fastest NVIDIA GPU. But when we compare the improvement in compute time with the data management time, we do not see a similar pattern. In fact, in some instances we find that data transfer on newer GPUs was actually slower than on the older one. In Table 6.5 the values marked in red show deterioration.

Figure 6.5 gives a better representation of irregularity. Taking the values

<table>
<thead>
<tr>
<th>Benchmark Applications</th>
<th>Compute</th>
<th>Data Management</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P100</td>
<td>V100</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>88.46</td>
<td>96.70</td>
</tr>
<tr>
<td>Gauss Seidel Method</td>
<td>75.27</td>
<td>75.62</td>
</tr>
<tr>
<td>Laplace Equation</td>
<td>49.97</td>
<td>17.55</td>
</tr>
<tr>
<td>SAXPY</td>
<td>87.31</td>
<td>94.08</td>
</tr>
<tr>
<td>Breadth First Search</td>
<td>78.20</td>
<td>89.07</td>
</tr>
<tr>
<td>HotSpot</td>
<td>86.92</td>
<td>94.08</td>
</tr>
<tr>
<td>k-Nearest Neighbor</td>
<td>58.85</td>
<td>64.18</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>93.48</td>
<td>96.34</td>
</tr>
<tr>
<td>Needleman Wunsch</td>
<td>75.46</td>
<td>83.85</td>
</tr>
<tr>
<td>Particle Filter</td>
<td>76.72</td>
<td>78.66</td>
</tr>
</tbody>
</table>

Table 6.5: % Improvement in performance on NVIDIA Tesla P100 and NVIDIA Tesla V100 when compared to NVIDIA Tesla K80
of K80 as baseline, there is always an improvement in the compute time, but data transfer time shows deterioration for most benchmarks. Sub-figure 6.5(a) displays the expected major improvement in compute time in P100 and V100 when compared to K80. But sub-figure 6.5(b) shows a considerable degradation in data management performance in P100 and V100 when compared to K80. This is due to the fact that performance of data transfer depends upon the node architecture, and in particular upon what kind of bridge is used between the CPU and GPU. If the GPU is connected to the adjacent CPU, it will need to cross over the processor interconnecting bridge as well before moving data into the GPU memory, potentially increasing the transfer cost. Since code may well need to execute on different GPU generations, it can be crucial for performance to optimize data transfers.

6.7 Conclusion & Future Work

Careful management of data and its mapping between host and device is critical for the use of accelerators in HPC. The complexities involved are a deterrent to the exploitation of GPUs. Automatically identifying and exploiting GPU data reuse in an application can reduce the amount of data transferred and hence the overall execution time. In our experiments, this optimization generally improved the performance of code created using the OpenMP implicit transfer approach and in no case led to a loss of performance. It thus contributes to the ease of use of OpenMP.

The main advantage of this work lies in the automation of the entire process. In addition to improving data handling in applications, it is also a step toward fully automatic GPU offloading of code, which would further reduce the barriers to use of GPUs for scientific computing.

Potential future extensions of this work include the following:

- Extend proximity analysis to analyze kernels beyond same function.
- Analyze and use unified memory for data management.
- Manage data to avoid potential memory over-subscription on the GPU.
- Automatically pin data to GPU to reduce multiple data allocation.
- Split data transfer and computation into multiple GPUs in case of over-subscription due to data reuse.
- Manage data transfer between multiple GPUs.
- Manage data of CPU to minimize data movement between CPUs.
Chapter 7

COMPOFF: A Compiler Cost model using Machine Learning to predict the Cost of OpenMP OFFloading

The HPC industry is inexorably moving towards an era of extremely heterogeneous architectures, with more devices configured on any given HPC platform and potentially more kinds of devices, some of them highly specialized. Writing a separate code suitable for each target system for a given HPC application is not practical. The better solution is to use directive-based parallel programming models such as OpenMP. OpenMP provides a number of options for offloading a piece of code to devices like GPUs. To select the best option from such options during compilation, most modern compilers use analytical models to estimate the cost of executing the original code and the different offloading code variants. Building such an analytical model for compilers is a difficult task that necessitates a lot of effort on the part of a compiler engineer. Recently, machine learning techniques have been successfully applied to build cost models for a variety of compiler optimization problems. In this paper, we present COMPOFF, a cost model that statically estimates the Cost of OpenMP OFFloading using a neural network model. We used six different transformations on a parallel code of Wilson Dslash Operator to support GPU offloading, and we predicted their cost of execution on different GPUs using COMPOFF during compile time. Our results show that this model can predict offloading costs with a root mean
squared error in prediction of less than 0.5 seconds. Our preliminary findings indicate that this work will make it much easier and faster for scientists and compiler developers to port legacy HPC applications that use OpenMP to new heterogeneous computing environment.

7.1 Introduction

Since the end of Dennard scaling hardware developers have improved chip performance by configuring a growing number of compute cores. This multicore processor technology was rapidly adopted by the High Performance Computing (HPC) community. It requires the modification of application codes to exploit the cores, e.g. by inserting pthreads or OpenMP constructs into the source code. In the last decade, General Purpose Graphics Processing Units (GPGPUs) have been attached to the multicore processors on many HPC platforms in order to benefit from their ability to handle large amounts of data parallelism with low power consumption. The trend toward heterogeneous HPC platforms is clearly visible in the most recent TOP500 list; while there are notable exceptions, a large fraction of the systems are heterogeneous with in most cases either NVIDIA GPUs or Intel Xeon Phis delivering high performance per watt. The second-ranked Summit supercomputing cluster is composed of two IBM PowerPC9 multicore CPUs with six NVIDIA V100 GPUs per node and three kinds of memory. Moving forward, we expect HPC platforms to be more diverse and potentially include domain-specific accelerators designed for specialized paradigms like machine learning, neuromorphic computing and ultimately quantum computing, leading to an era of extreme heterogeneity.

The presence of a single type of accelerator on a node already poses a challenge to current programming environments; we are not yet well prepared for a future in which multiple types of heterogeneous accelerators may be configured. Many application developers are adapting their codes to exploit GPUs. Lattice Quantum Chromodynamics [15], an application developed under the US Department of Energy’s ECP project, is one such application that can greatly benefit from the use of accelerators such as GPUs. Unfortunately, effectively utilizing GPUs is a time-consuming endeavor that may necessitate re-engineering data structures as well as large regions of code in order to maximize the GPU’s computational power while minimizing overheads. It will be far more difficult to develop code for systems with extreme het-
erosogeneity containing different devices. It is therefore imperative to develop
techniques that will relieve the application developers of the burden of such
development.

Using a directive-based programming model, such as OpenMP, the de-
facto programming standard for parallel programming in C/C++ and For-
tran, is one way to handle portability across architectures. OpenMP now
supports GPU offloading and is planning a transition to extreme hetero-
gegeneity [50], rendering the underlying hardware transparent and allowing
for greater portability. Nevertheless, even with directive-based programming
models like OpenMP, optimizing large scale applications with tens to hun-
dreds of thousands of lines of code remains an arduous task.

In this work, we explore state of the art ML techniques to develop COM-
POFF (Cost of OpenMP OFFloading), a first of its kind compiler cost model-
ing tool that employs ML to predict the execution time of an OpenMP kernel
on GPUs during compilation. We first discuss some cutting-edge work that
is related to and precedes our work in Section 7.2, followed by Section 7.3’s
discussion of the challenges encountered during offloading using OpenMP. In
Section 7.4, we introduce static program features, that can be used to create
a static cost model using Machine Learning, independent of compilers and
hardware dependencies, as well as how we synthesize data to train our model.
The design of our model is then presented in Section 7.5. Section 7.6 cov-
ers the experiments carried out in this paper in order to predict the cost of
offloading the Wilson-Dslash stencil operator [80] for Lattice Quantum Chro-
modynamics application. The results are analysed in Section 7.7. Finally,
we conclude our work with future discussion in Section 7.8.

7.2 Related Work

Compiler engineers are developing a number of frameworks [90, 95, 116] to as-
sist application developers deal with extreme heterogeneity more effectively.
These frameworks require analytical cost models to help them make better
decisions when selecting the choice of optimization or transformation required
by the application. However, developing a cost function is time-consuming,
and almost all modern compilers, including LLVM/Clang, use a simple “one-
size-fits-all” cost function that does not provide the best performance in the
case of diverse architecture. Hand-tuned cost functions are currently popu-
lar, but calculating the costs and benefits of a compiler optimization requires
a deeper understanding of the underlying hardware. Despite its effectiveness, manually constructing a cost model for a single architecture can take months. Since cost functions are critical and manual tuning is rather laborious, compiler engineers are investigating Machine Learning (ML) techniques as a means of automating this process.

Early work exploiting ML in compilers, like [100], primarily explored its use to help optimize sequential programs. However, with the proliferation of multi-core platforms and, more recently, heterogeneous systems, its application to the task of optimizing parallel programs has received significant attention in the last decade. A decision-tree-based approach has been developed to predict the scheduling policy for an OpenMP parallel region [93]. Furthermore, [31] optimizes OpenMP programs for scheduling policies and thread count using ML techniques. ML has been used to determine the optimum degree of parallelism for transactional memory [153] and hardware resource allocation [64]. The work presented in [9] and [67] applies ML to complex parallel programs and divide them among the available multi-core resources. The Petabricks project [4] employs a genetic search to tune algorithmic choices at compile time to reduce searching overheads. Tree and graph-based features have also been used by Malik et.al. [86] who present a unique graph-based approach for feature representation. ML techniques were used to build classifiers to determine whether to offload OpenCL code [33], and to select a clock frequency at which the processor should operate [61]. The work was reported to be extremely accurate, but the benefits could not be quantified because no modified code was generated. The results of prior efforts from applying ML on compiler optimizations are encouraging. However, new feature engineering practices that can help ML learn more about a code and its computational needs must be investigated.

7.3 OpenMP

The OpenMP API 4.0 specification (released in 2013 [107]) includes a collection of directives that tell the compiler when to offload a block of code to devices, like GPU, FPGA etc. However, achieving scalable performance on large parallel machines still necessitates significant effort in performance tuning, particularly in terms of cache management and locality, data and work sharing, and synchronizations.

When each architecture supports either a distinct native language or al-
ternative optimization methods for the same language, program portability becomes a major problem. This is especially important for users whose programs must run smoothly on systems with diverse node architectures, such as manycore vs. GPU-accelerated nodes, or when dealing with multiple device memory accesses to the same data. In the future, the ultimate level of portability would be if we can have developers write a sequential code, like matrix multiplication in Code 7.1, and then build this code using any compiler on any platform and expect it to exploit all hardware and accelerator parallelism effectively.

Unfortunately, programming languages and compilers are far away from a state where they can handle portability without programmer intervention. Usually, a base language lacks all of the features that a software developer needs, and they rely on libraries to meet their performance goal. Architecture-specific libraries, like CUDA, can be used to extract all of NVIDIA GPU’s performance to achieve accelerator parallelism, but portability suffers as a result. Directive-based programming languages, like OpenMP (e.g. Code 7.2), serve as a middle ground, providing a portable way to augment the base languages, while filling in the performance gaps required to support a specific architecture.

7.3.1 GPU Offloading in OpenMP

OpenMP provides additional information to the compilers, which bridges a gap from serial programming languages to parallel programming languages. GPUs are highly parallel, and the programmer should fully utilize it’s par-

```c
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        float tmp = 0.0f;
        for (int k = 0; k < n; k++) {
            tmp += A[i*n+k] + B[k*n+j];
        }
        C[j*n+i] = tmp;
    }
}
```

Code 7.1: Sequential Matrix Multiplication program
```c
#pragma omp target
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        float tmp = 0.0f;
        for (int k = 0; k < n; k++) {
            tmp += A[i*n+k] + B[k*n+j];
        }
        C[i*n+j] = tmp;
    }
}
```

Code 7.2: Matrix Multiplication on GPU using OpenMP

Figure 7.1: Teams distribute parallel for

...allel capacity to extract maximum performance. Simply parallelizing a code using "omp parallel for" will parallelize it for CPUs, but will not offload the computation to a GPU. OpenMP device offloading consists of two major components:

- Data mapping between host and device
- Offloading computations from host to device.

All data is initially stored in CPU memory, and GPUs have no access to it. To access host data, the GPU must first move the data from the host (CPU) to the device (GPU) using the data map clause, and then move the
data back from the device to the host once the computation on the GPU is complete. The **omp** parallel for creates a single contention group of threads with shared memory and the ability to coordinate and synchronize, while the **omp target** directive marks a code section for offloading. But

```c
#pragma omp target teams distribute parallel for
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        float tmp = 0.0f;
        for (int k = 0; k < n; k++)
            tmp += A[i * n + k] * B[k * n + j];
        C[i * n + j] = tmp;
    }
}
```

Code 7.3: Combined parallelism

```c
#pragma omp target teams distribute
for (int i = 0; i < n; i++) {
#pragma omp parallel for
    for (int j = 0; j < n; j++) {
        float tmp = 0.0f;
        for (int k = 0; k < n; k++)
            tmp += A[i * n + k] * B[k * n + j];
        C[i * n + j] = tmp;
    }
}
```

Code 7.4: Split parallelism

```c
#pragma omp target teams distribute parallel for
for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++) {
        float tmp = 0.0f;
        for (int k = 0; k < n; k++)
            tmp += A[i * n + k] * B[k * n + j];
        C[i * n + j] = tmp;
    }
}
```

Code 7.5: Swap Combined parallelism
GPUs are not parallel worksharing machines.

On a platform like GPU, we expect a high degree of coarse grain parallelism across the entire device. This structure limits the degree of parallelism that a GPU can exploit. Instead, programmers can use multiple options provided by OpenMP directives, like `teams distribute`, which exposes coarse grained, scalable parallelism to the entire GPU. OpenMP teams and
distribute are directives that spawn additional level of parallelism, as shown in Figure 7.1. Only one team and one member thread are active at the start of a target region. If we want to have multiple teams, we use the directive `teams distribute` first, which distributes the entire loop iteration space among all teams. Furthermore, if there are more nested parallel loops, we use the `parallel for` directive upon them to distribute the iterations of the nested loops among threads within a team. When there is only one level of parallel loops, or when the outer loop has enough parallelism, we use the combined directive `teams distribute parallel for` to distribute the iteration space of one loop among teams and threads within a team.

Threads are organized into groups using `teams`, and `distribute` allows to schedule a group of teams to run a loop. As there is no synchronization primitive to act as a barrier between threads belonging to different teams, `teams` appear to be similar to `CUDA threadblocks` on NVIDIA GPUs. As is usual, `parallel for` is used to parallelize the threads within each team. Coarse grained parallelism may be combined (Code 7.3) or split (Code 7.4). There could be other transformation as well which we could use to fully utilize the parallelism on a GPU. For instance, in Code 7.5 and 7.6 the programmer could swap the outer (iterating over $i$) and inner (iterating over $j$) loops and apply combined and split parallelism on them as well. A programmer could also collapse the two loops as shown in Code 7.7 and 7.8 to exploit its full degree of parallelism. The implementation of collapse varies depending on the compiler, as does how it affects the outcome. But, how do we decide which of the these transformation will be better for our kernel and architecture? One possible approach is to take such decision during compile time using some static cost model.

## 7.4 Machine Learning in Compilers

Despite their efficacy, designing hand tuned cost models for each architecture is rather costly and time consuming. To automate the process, compiler developers are turning to machine learning techniques. Identifying the feature set that can be used to train an ML model is the first step toward using ML in compilers. Since the goal of this work is to maintain portability using OpenMP, we must look for features that are platform and compiler independent. This means we should only consider features from the original source code written by the programmer without applying any compiler optimiza-
<table>
<thead>
<tr>
<th>Types</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism</td>
<td>Outer</td>
<td>Outer Loop of Iteration</td>
</tr>
<tr>
<td></td>
<td>Inner</td>
<td>Inner Loop of Iteration (Set to 0 when there is no nested loop)</td>
</tr>
<tr>
<td></td>
<td>Schedule</td>
<td>static, dynamic, guided, auto, runtime</td>
</tr>
<tr>
<td>Memory</td>
<td>MemTo</td>
<td>Total memory transferred to GPU</td>
</tr>
<tr>
<td></td>
<td>MemFrom</td>
<td>Total memory transferred from GPU</td>
</tr>
<tr>
<td></td>
<td>VarDecl</td>
<td>Total number of variable declaration</td>
</tr>
<tr>
<td></td>
<td>RefVar</td>
<td>Total number of variable referenced</td>
</tr>
<tr>
<td></td>
<td>IntLiteral</td>
<td>Total number of integer constant</td>
</tr>
<tr>
<td></td>
<td>FloatLiteral</td>
<td>Total number of floating point constant</td>
</tr>
<tr>
<td></td>
<td>IntAssign</td>
<td>Total integer assignment</td>
</tr>
<tr>
<td></td>
<td>FloatAssign</td>
<td>Total float assignment</td>
</tr>
<tr>
<td>Computation</td>
<td>IntAddSub</td>
<td>Total integer add and subtract</td>
</tr>
<tr>
<td></td>
<td>FloatAddSub</td>
<td>Total float add and subtract</td>
</tr>
<tr>
<td></td>
<td>IntMult</td>
<td>Total integer multiplication</td>
</tr>
<tr>
<td></td>
<td>FloatMult</td>
<td>Total float multiplication</td>
</tr>
<tr>
<td></td>
<td>IntDiv</td>
<td>Total integer division</td>
</tr>
<tr>
<td></td>
<td>FloatDiv</td>
<td>Total floating point division</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td>Total logical, relational and bitwise operations</td>
</tr>
</tbody>
</table>

Table 7.1: Static Kernel features independent of architectures and compilers

Furthermore, because we need to make predictions at compile time, these features should be static in nature. For the purpose of this research, we only consider the six transformations explained in Section 7.3.1 and shown in Code 7.3 – 7.8.

After studying several GPU cost models ([20, 7, 156, 47]), we conclude that there are three major factors which affect the cost of execution of a kernel on a GPU - level of parallelism, memory access, and computation to be performed by the kernel. To train a model which could predict the cost of execution of above transformation, we extract static features from kernels,
which can be grouped in accordance to these factors. A list of features that we consider in this paper can be found in Table 7.1. Here we group the features based on level of parallelism, memory access and computation.

**Level of parallelism.**

To check for the level of parallelism we can look into the number of iterations of the *for loop*, the number of available threads and how OpenMP schedules the iterations between the threads. Usually, the number of available threads on a GPU and the scheduling-type used by OpenMP are present at the time of compilation, but the number of iterations are usually a variable. For this current study we make sure that the number of iteration in the *for loop* are also constant. Predicting the cost at runtime using variable iterations is part of the future work. The level of parallelism can be increased by collapsing the nested loops. In this work, we restrict the nesting of loops to two levels only, referring to them as the outer and inner loops.

**Memory access.**

In Section 7.3.1, we discussed how the memory of the CPU and GPU differs and how data must be synced between them for the kernel to function correctly. Data movement between CPU and GPU (to and from) is a big factor which affects the execution time of a kernel. Another factor which affects the execution time of a kernel is how many times memory locations are declared, referred and written to.

**Computation.**

Finally we count the number of operations that occur in a kernel. One primary reason why a generic cost model with a "one size-fits-all" cost function (like the LLVM cost model [85]) does not deliver the best results is that it considers the cost of running all operations to be equal. But, in practice this is not true. For instance, a multiplication operation will take more time than an addition or logical operation. Also, floating point multiplication, might take different time than integer multiplication. Hence, we need to extract the number of times each of the operations are called. Still there are some operations which we can assume to have same cost of execution. For instance, we combine addition and subtraction which has the same cost of execution.
7.4.1 Data Collection

The absence of publicly available data is the most significant challenge that any ML engineer faces for compiler problems. We faced the same problem as well. So, the first step in developing a machine learning-based cost model was to create a dataset that could be used to train our model. And we needed a dataset which would cover all the features, defined in our feature set. To cover all the features, we wrote mini benchmark applications like Matrix Multiplication, Laplace Equation, SAXPY, Gauss Seidel Method and use some benchmarks from the Rodinia benchmark suite, like the BFS, LU-Decomposition, Particle Filter, etc. We created over 10,000 different kernels by varying the levels of parallelism and data used for all these benchmark applications. Then, we extracted and collected data on the feature set defined in Table 7.1 as well as the execution time of each of these kernels.

7.4.2 Wilson Dslash Operator

In addition to the benchmark applications, we employ a kernel that represents the Wilson Dslash Operator [80] found in the Lattice Quantum Chromodynamics (LQCD) application. QCD is the theory of the strong nuclear force, one of nature’s fundamental forces that holds quarks together to form protons and neutrons, which then form the nuclei of atoms. LQCD is a computer-friendly numerical framework for QCD, that employs the Wilson Dslash kernel, which is essentially a finite difference operator.

The Wilson Dslash operator in four space-time dimensions is defined as

$$D^{ij}_{\alpha\beta}(x, y) = \sum_{\mu=1}^{4} \left[ ((1 - \gamma_\mu))_{\alpha\beta} U^{ij}_\mu(x) \delta_{x+\hat{\mu}, y} + (1 + \gamma_\mu)_{\alpha\beta} U^{ij\dagger \mu}(x + \hat{\mu}) \delta_{x-\hat{\mu}, y} \right]$$

where $x$ and $y$ are the coordinates of the lattice sites, $\alpha, \beta$ are spin indices, and $i, j$ are color indices. $U_\mu(x)$ is the gluon field variable and is an $SU(3)$ matrix. For the unpreconditioned Dirac operator, the complex fermion fields are represented by one-dimensional arrays of size $L_X \times L_Y \times L_Z \times L_T \times SPINS \times COLORS \times 2$ where $L_X, L_Y, L_Z$ and $L_T$ are the numbers of lattice sites in the $x, y, z$ and $t$ directions, respectively. $SPINS$ and $COLORS$ are the numbers of spin and color degrees of freedom, typically 4 and 3 respectively.

Equation 7.1 when represented in a C++ code has four nested for loops.
iterating over $L_T, L_Z, L_Y, L_X$ in that order. We modified the source code for this study to offload this kernel to the GPU using all six transformations described in Section 7.3.1. We run this operator for different values of $L_T$ and $L_Z$ (the two outer loops) to determine the runtime on Summit and Seawulf HPC clusters. On both clusters, we collected approximately 40 data points for each transformation. We used some of these data points to train our model and kept the rest to evaluate our model’s prediction accuracy, as described later in Section 7.6.

7.5 Training Models

The available data and the target execution time predictions lean towards a supervised machine learning (ML) approach, which involves learning a feature representation and fitting a predictive model using training data containing ground-truth targets. This can be accomplished with a number of machine learning model types, all designed to ultimately perform a regression task. There are several ways of performing regression using the collected data. In all cases, the available data – populated parameters and corresponding measured execution time – is divided into disjoint training and testing sets, with model’s parameters determined by fitting on the training set, and the model’s performance determined separately on the hold-out test set. This split allows for the evaluation of the models’ generalized performance on previously unseen data.

7.5.1 Why Regression?

Since the runtime is a positive real-valued variable, the prediction task for all ML models is a regression problem, as opposed to classification, which has a limited set of possible outcomes. The rationale for using machine learning regression models to predict the cost of OpenMP offloading is the reasonable expectation that when well-selected and measurable kernel parameters are combined, they will be strongly predictive of kernel execution time, but the nature of this predictive correlation may be complex. ML approaches are well suited to determining these correlations in a data-driven manner – without exhaustive reasoning and debate about the relative importance of and relationship between particular features.

The standard linear regression approach to this problem determines the
specific linear combination of all input parameters that best fits the target values. Although the restriction to linear relationships is constraining in practice, the easy-to-interpret weights and biases among features make this a useful and accessible baseline. One addition we made in this work was the extension of input parameters by appending a log-transformed duplicate set, thus enabling approximate representation of multiplicative and rational relationships (as sums and differences of logs) as well as linear ones.

### 7.5.2 Neural Network

Linear regression applies a linear fit across all input in the original parameter space, and the data must have a linear relationship in order for this to work. This condition is violated in the data we collect – some features are strongly correlated, while relationships for others cannot be determined in the original parameter space. In order to find a better fitting non-linear data relationship, stronger ML models can map inputs to a different parameter space (e.g. in a higher dimension). Support Vector Machines (SVMs) and Artificial Neural Networks (ANNs) are examples of ML techniques that can learn complex feature relationships that are impossible to capture with linear regression while maintaining efficiency at inference time. We chose neural networks over

![Diagram](https://example.com/diagram.png)

(a) Feature Extraction  
(b) Training Model  
(c) Prediction

Figure 7.2: Overview of the ML Model
SVMs because they can capture even more complex data relationships while being easier to parameterize and converge using gradient-based optimization algorithms (e.g., Stochastic Gradient Descent).

ANNs are a very diverse family of models; the class we employ is fully-connected feed-forward networks, also called Multi-Layer Perceptrons (MLPs). An MLP is essentially a stacked series of linear regression layers. The non-linearity is added by activation functions such as ReLU [36], sigmoid, tanh, functions etc, added on top of each linear layer. The key hyperparameters of MLPs are the number and size of the hidden layers which build up increasingly complex data representations before a final regression layer is applied to perform the ultimate prediction.

Figure 7.2(a) and 7.2(b), gives a basic idea of the flow of training our model. When we want to use this model we extract the required features from the new kernel, append a log-transformed duplicate set, standardize these features using the mean (\(\mu\)) and standard deviation (\(\sigma\)) from the original model, and pass the standardized data to our model, which returns the prediction of the kernel’s execution time (as shown in Figure 7.2(c)).

### 7.6 Experiments and Evaluations

Section 7.4 already discussed multiple code level transformations which we can use to offload an OpenMP parallel loop to the GPU. As a proof of concept for this paper, we only consider these six transformations, and create a model for each of them individually. We experiment on two different HPC clusters, each with an LLVM/Clang 13.0 compiler that supports GPU offloading:

1. Summit Supercomputer with NVIDIA V100 GPUs
2. Seawulf cluster with NVIDIA K80 GPUs

Despite the fact that each node on both clusters is connected to multiple GPUs, we only consider single GPU for the purposes of this research. Dealing with multiple GPUs will be left for future research. Each of these transformations has data in our dataset, with the kernel built on both clusters. We trained a total of twelve models, one for each of the six transformation, on both clusters.

COMPOFF employs an MLP model with six layers – 1 input, 4 hidden and 1 output neurons. Rather than selecting an arbitrary number of neurons in each hidden layer or perform an exhaustive grid search, we set the number
of neurons on multiples of the number of input features (number of neurons in the first layer). Therefore, with 33 input features, the first, second, third and fourth hidden layers have 66, 132, 66 and 33 neurons respectively. Glorot initialization, described in [40], is used to set the weights of linear layers. The bias is initialized to 0. In all runs, the batch size for training data is set to 16.

We experiment with SGD (Stochastic Gradient Descent), Adam [73] and RMSprop [51] as the underlying optimization algorithm. We choose the RMSprop optimization algorithm with an initial learning rate of 0.01 that is stepped down every 30 epochs by a factor of 0.1 and weight decay of 0.0001 for 150 epochs because it gives us optimal performance on transformations for both HPC clusters.

The objective function to train all models is based on the Mean Square Error loss function defined by:

$$l(y_i, \hat{y}_i) = (\hat{y}_i - y_i)^2$$

where $\hat{y}_i$ and $y_i$ represent the predicted and ground truth runtime, respectively.

Each training dataset is divided into train (72%), validation (8%) and testing sets (20%). The model does not learn from the validation and testing sets (20%).

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Cluster</th>
<th>R.M.S.E. (sec)</th>
<th>M.A.P.E.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collapse</td>
<td>Seawulf</td>
<td>0.279</td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.090</td>
<td>0.211</td>
</tr>
<tr>
<td>Combined</td>
<td>Seawulf</td>
<td>1.368</td>
<td>0.053</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.399</td>
<td>0.138</td>
</tr>
<tr>
<td>Split</td>
<td>Seawulf</td>
<td>0.420</td>
<td>0.044</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.112</td>
<td>0.227</td>
</tr>
<tr>
<td>Swap Collapse</td>
<td>Seawulf</td>
<td>1.242</td>
<td>0.055</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.128</td>
<td>0.120</td>
</tr>
<tr>
<td>Swap Combined</td>
<td>Seawulf</td>
<td>0.669</td>
<td>0.027</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.276</td>
<td>0.062</td>
</tr>
<tr>
<td>Swap Split</td>
<td>Seawulf</td>
<td>0.919</td>
<td>0.059</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.454</td>
<td>0.284</td>
</tr>
</tbody>
</table>

Table 7.2: Evaluation results on the micro-benchmark data files for respective transform and cluster
<table>
<thead>
<tr>
<th>Transformation</th>
<th>Cluster</th>
<th>R.M.S.E.(sec)</th>
<th>M.A.P.E.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collapse</td>
<td>Seawulf</td>
<td>0.143</td>
<td>0.576</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.067</td>
<td>4.201</td>
</tr>
<tr>
<td>Combined</td>
<td>Seawulf</td>
<td>1.500</td>
<td>0.323</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.333</td>
<td>0.259</td>
</tr>
<tr>
<td>Split</td>
<td>Seawulf</td>
<td>0.384</td>
<td>1.044</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.041</td>
<td>0.584</td>
</tr>
<tr>
<td>Swap Collapse</td>
<td>Seawulf</td>
<td>8.972</td>
<td>3.305</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.512</td>
<td>2.871</td>
</tr>
<tr>
<td>Swap Combined</td>
<td>Seawulf</td>
<td>1.771</td>
<td>3.321</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.584</td>
<td>2.686</td>
</tr>
<tr>
<td>Swap Split</td>
<td>Seawulf</td>
<td>2.668</td>
<td>7.083</td>
</tr>
<tr>
<td></td>
<td>Summit</td>
<td>0.032</td>
<td>0.515</td>
</tr>
</tbody>
</table>

Table 7.3: Evaluation results on Wilson Dslash kernel data

testing sets. Z-score standardization is the only augmentation applied to training and validation data. We use the training set to train the model and the unseen validation set to validate its growth. The computed statistics are then applied to testing data. For our second set of experiments with Wilson Dslash Operator data, we add 20 data points to the original dataset and hold the remaining 20 data points for evaluation. We calculate Root Mean Square Error (R.M.S.E.), as the standard deviation of the prediction errors. The lower this value, the more accurate our model is. In addition, we calculate the Mean Absolute Percentage Error (M.A.P.E.) to determine the accuracy of our model.

### 7.7 Results and Analysis

A good statistical validation technique provides us with a thorough measure of our model’s performance across the entire dataset. The validation set is selected at random for each data file. The validation set helps us in understanding the models progress. Once a training epoch is complete, we freeze the network and compute the R.M.S.E. and M.A.P.E. on the validation set. These values are expected to decrease as we train and then increase eventually due to overfitting.

Following the training protocol outlined in the previous section, we inves-
tigate the models’ fit across multiple data files. On regular microbenchmark data, we can see that a simple MLP performs impressively. The results are shown in Table 7.2. Due to the small number of data points collected for the Wilson Dslash Operator, we can see that the model can adapt to the new application relatively well in some cases but not so well in others, as shown in Table 7.3. The new application’s few data points are clustered further apart from the data collected by the microbenchmark application, resulting in a data imbalance. This is commonly referred to as the long-tail problem seen in many real-world applications, and is not in the scope of this work.

The goal of this work is to show that ML regression models can be used to predict the cost of OpenMP offloading for different kernels, and it is evidenced by the strong correlation between actual and predicted data in Figures 7.3
and Figure 7.4. When looking at Figure 7.4(a), it may appear that the predictions on Summit are dispersed. However, when the scale of the steps on its y-axis and the R.M.S.E. values are taken into account, we can see that the actual and predicted data are strikingly similar. Furthermore, we see the model perform well on a wide range of data (from 100s of ms to a few hundred seconds), implying that a single model can be trained to predict high variance data with careful finetuning, hyperparameter setup, and a large amount of data. Because the R.M.S.E. for these predictions is so low, we can ignore the higher M.A.P.E. values observed in some cases. This is also observed in the prediction for the Wilson Dslash operator data as evident in Figure 7.5 (the x-axis represents different Wilson Dslash kernels, sorted by actual execution time). The blue bar in the figure represent the actual runtime of the kernel, while the orange dashed bar represents the prediction by COMPOFF. We can see that in all cases, the prediction is close enough to the actual runtime.
7.8 Conclusion and Future Work

This work is a proof of concept that an ML model can be trained and used by compiler developers to make better decisions in offloading a kernel to a GPU using OpenMP. Our findings show that this model can predict offloading costs of several benchmark applications and one real application (Wilson Dslash operator) with a Root Mean Squared Error of less than 0.5 seconds. It has some limitations though, biggest of which is the lack of an exhaustive dataset. The current dataset that we collected for the purpose of this work is sufficient for a proof of concept. But to create a portable cost model across multiple applications and architectures, we need to train the ML model extensively across several platforms, compilers, and applications.

Even if we have an exhaustive dataset across various environments, another big challenge is runtime parameters tuning. It is observed that the same kernel, when run on the same platform multiple times show different execution time. It is seen that the execution time differ by a couple of seconds. Therefore, in terms of percentage variation, for smaller kernels which run for only a second or so, this variation could be huge. This brings in a lot of noise in our training set for smaller kernels. However, in large kernels, a couple of seconds of variation in the prediction is usually not a big issue and can be ignored. In actual practice, if a kernel is small and does not have enough computational work, a GPU execution is not justified. Therefore, currently, we leave the decision of how to interpret the prediction for smaller kernel upon the users of COMPOFF.

Other parameters that we have not considered in this work and that may affect the execution time of a kernel are:

- **Data affinity** – Data affinity is not yet supported by OpenMP and hence we are also not considering it.

- **Data reuse** – We have previously demonstrated [97] that reusing data between multiple kernels improves the overall execution time of an application. Currently, this work does not consider data reuse between multiple kernels, and each of the kernels are considered mutually independent with respect to data reusability.

- **Unified memory** – Previous studies [78, 96] have shown that using unified memory between CPU and GPU, an application can signifi-
cantly improve OpenMP GPU offloading performance. However, we haven’t yet considered unified memory in this project.

- **Data overfitting** occurs when a statistical model fits exactly against its training data, but fails against unseen data, defeating its purpose. Even though this is partially mitigated by the use of validation sets, regularization and dropout layers, it is not a foolproof method.

- **Constants** – As with most compile time cost models, COMPOFF can only predict the execution time of kernels whose data size and level of parallelism are available during compile time.

For our future work, we plan to apply ML techniques to aforementioned challenges. We also plan to work on improving feature selection and representation for parallel programming models. Recently, Graph Neural Network is gaining popularity for its performance for learning graph representation. We plan to use it in the future to learn characteristics of parallel code.
Chapter 8

Experiments and Results

We used several clusters and compilers, as shown in Table 8.1, to test and evaluate our tool. For the purposes of this study, we only use one GPU per node on the cluster. The management of multiple GPUs is left for future research. The three modules explained in Chapter 5 need different experiments.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>GPU</th>
<th>Compiler</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit [109]</td>
<td>NVIDIA Tesla V100</td>
<td>LLVM/clang (nvptx)</td>
<td>13.0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GNU/gcc (nvptx-none)</td>
<td>9.1.0</td>
</tr>
<tr>
<td>Corona [74]</td>
<td>AMD Radeon Instinct MI50</td>
<td>LLVM/clang (rocm-5.3)</td>
<td>15.0.0</td>
</tr>
<tr>
<td>Ookami [17]</td>
<td>NVIDIA Tesla V100</td>
<td>LLVM/clang (nvptx)</td>
<td>14.0.0</td>
</tr>
<tr>
<td>Wombat [111]</td>
<td>NVIDIA Tesla A100</td>
<td>LLVM/clang (nvptx)</td>
<td>15.0.0</td>
</tr>
<tr>
<td>Seawulf [144]</td>
<td>NVIDIA Tesla K80</td>
<td>LLVM/clang (nvptx)</td>
<td>12.0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NVIDIA/NVC</td>
<td>21.7</td>
</tr>
<tr>
<td>Intel DevCloud [60]</td>
<td>Intel Xeon E-2176 P630</td>
<td>Intel/icpx</td>
<td>2021.1.2</td>
</tr>
<tr>
<td>Exxact</td>
<td>NVIDIA GeForce RTX 2080</td>
<td>LLVM/clang (nvptx)</td>
<td>14.0.0</td>
</tr>
</tbody>
</table>

Table 8.1: Cluster and Compilers used in experiment
8.0.1 Experiment 1 - Data Analysis

First, we test our Advisor against all benchmark applications to determine whether or not data is correctly identified and generated. In order to conduct this experiment, we made use of our advisor to generate code that used the correct data between the host and the device. Additionally, we manually altered each benchmark algorithm to map all data to and from the GPU. We executed all the codes on each cluster, as shown in Table 8.1, and gathered data about the volume and the duration of the data transfer.

We found that the Advisor improved the data management in all cases. Figure 8.1 shows the amount of data transferred (in GB) between the CPU and the GPU before and after transformation for all benchmark applications. After applying our transformation, we can clearly see that the amount of data

![Figure 8.1: Total Data transfer for all Benchmarks Before and After Data Analysis](image)

126
transfer has indeed been considerably reduced. Reduced data transfer has an impact on all applications’ data transfer times. On all the available clusters, we ran these applications and collected the data transfer times. Figure 8.2 shows the data transfer time for the Wilson Dslash Operator across different clusters.

8.0.2 Experiment 2 - Code Generation

In the first experiment, the code generation for data analysis has already been tested. But that experiment does not generate different variants of code. In the second experiment, we use our Advisor to generate every possible code combination using each of the Benchmark applications, as discussed in Section 5.3.1. Table 8.2 shows the result of this experiment. We compiled all of these codes for various clusters using the compilers shown in Table 8.1. Some compilers (NVIDIA/nvc on Seawulf and LLVM/Clang 15 on Wombat) do not support dynamic or guided scheduling on a GPU, resulting in compilation failure. Apart from that, all of the codes successfully compiled and ran on their respective clusters. We collected the runtime of each of the kernels in this experiment, to be used by our cost model.

We collected the data for the Intel architecture a while ago, and we don’t currently have access to the cluster to conduct new experiments. As a result we had very limited data for Wilson Dslash Operator and no data for our proxy_app on the Intel architecture. We were unable to gather many data points for the Exxact machine (with NVIDIA GeForce GPUs) due to the unavailability of compute nodes. Both these clusters has only around 2,000 data points each.

Seawulf has NVIDIA K80 GPUs, which is the slowest of the GPUs we’re
Table 8.2: Number of variants generated for all applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Kernel</th>
<th># Loops</th>
<th>Variants Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>kernel1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>kernel2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>correlation</td>
<td>kernel1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>covariance</td>
<td>kernel1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>kernel2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>gauss</td>
<td>kernel1</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>laplace</td>
<td>kernel1</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>kernel2</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>knn</td>
<td>kernel1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>mm</td>
<td>kernel1</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>mv</td>
<td>kernel1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>mt</td>
<td>kernel1</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>kernel2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>kernel3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>kernel4</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>kernel5</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>kernel6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>kernel7</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>particle</td>
<td>kernel7</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Proxy App</td>
<td>kernel1</td>
<td>4</td>
<td>84</td>
</tr>
<tr>
<td>Wilson Dslash</td>
<td>kernel1</td>
<td>4</td>
<td>84</td>
</tr>
</tbody>
</table>

using in our experiment. So each kernel runs longer on Seawulf than it would on any other cluster. On the other hand, most variants of kernels failed to compile on Wombat due to their compilers not supporting dynamic and guided scheduling on GPUs. Due to these reasons, we could only collect around 3,000 data points on Seawulf and Wombat. All our kernels compiled and ran successfully on Summit, Corona and Ookami and we were able to collect over 10,000 data points on each of these architectures.

8.0.3 Experiment 3 - Cost Model

To build our cost model, we extended the COMPOFF model from six variants to all 84 variants. We build our cost model in the testing mode and then use it to predict the runtime in the prediction mode. Our cost model utilizes an MLP model with six layers: one input layer, four hidden layers, and one
output layer. We set the number of neurons on multiples of the number of input features rather than choosing a random number of neurons in each hidden layer or conducting an exhaustive grid search (number of neurons in the first layer). As a result, the first, second, third, and fourth hidden layers, with 33 input features, have 66, 132, 66, and 33 neurons, respectively.

The weights of linear layers are set using the glorot initialization method, which is described in [40]. The bias is set to 0, and the batch size for training data is set to 16 in all runs. As the underlying optimization algorithm, we evaluate SGD (Stochastic Gradient Descent), Adam [73] and RMSprop [51]. Since it provides optimal performance on transformations for both HPC clusters, we chose the RMSprop optimization algorithm with an initial learning rate of 0.01 that is stepped down by a factor of 0.1 every 30 epochs and weight decay of 0.0001 for 150 epochs. The Root Mean Square Error (RMSE) loss function is used to train all models, as defined by Equation 8.1, where \( \bar{y}_i \) and \( y_i \) represent the predicted and ground truth runtimes, respectively.

\[
RMSE(\bar{y}, y) = \sqrt{\frac{1}{N} \sum_{i=0}^{N} (\bar{y}_i - y_i)^2}
\]  

(8.1)

We split the dataset used by all benchmark applications into two parts: training (80%) and validation(20%). The validation set do not occur in any
learning for the model. The only augmentation applied to the training and validation data is Z-score standardization. The model is trained using the training set, and after that, testing data are given to the model to test its performance. In order to determine the standard deviation of the prediction errors, we compute the RMSE. The lower this value, the better our model. However, what constitutes a good RMSE is determined by the range of data on which we are computing RMSE. One way to determine whether an RMSE value is “good” is to normalize it using the following formula:

\[
NRMSE(\bar{y}, y) = \frac{RMSE(\bar{y}, y)}{(y_{max} - y_{min})}
\] (8.2)

This yields a value between 0 and 1, with values closer to 0 indicating better fitting models. Having a model with a normalized RMSE of less than 0.1 is considered successful in this study.

We can see that a simple MLP performs admirably in all of our applications, as evidenced by the strong correlation between actual and predicted data in Figure 8.3. It was anticipated that Intel and Exxact’s model would perform the worst because of the lack of data. However, the model for Exxact performed much better (and also showed signs of convergence) than that on Intel because we were able to collect some data for our proxy app on the Exxact system. Both Wombat and Seawulf performed moderately well when compared to other models trained on a larger dataset. It is still an open question that how much data is enough data to train an ML model. We have observed from Figure 8.3, however, that if we have more than 10,000 data points for our model, we will be able to train a model that is much more acceptable.

8.0.4 Experiment 4 - Prediction

Finally, for our final set of experiments we use our Advisor to predict the top 10 best variants for the Wilson Dslash Operator. Once the top 10 variants are identified we use the Code Transformation module to generate those 10 code variants and return them back to the user. The Advisor takes as input the base Wilson Dslash kernel, and generates the top 10 best kernels as predicted by the cost model. As shown in Figure 8.4, we plot the actual and predicted runtimes of all the 84 generated variants (sorted by actual runtime) of one such kernel when run on the Summit supercomputer. We can clearly see a strong correlation between the actual and predicted runtime.
Max error for kernels $> 45s = 3.04s$

Max error for kernels $< 2s = 0.76s$

Figure 8.4: Wilson Dslash operator’s Actual and Predicted Runtimes (in sec) on Summit for all variants sorted by runtime, where $L_X, L_Y, L_Z$ and $L_T$ are set to 32, 32, 32 and 16.

for all the variants. The same correlation can be found in almost all kernels across all clusters. In Figure 8.5, we display the Wilson Dslash operator’s RMSE and normalized RMSE for each cluster. The range of runtimes (in

Figure 8.5: RMSE and Normalized RMSE for predicting the runtime of all variants of Wilson Dslash operator on different clusters for $L_X, L_Y, L_Z, L_T$ set at 32, 32, 32, 16 each. The range of runtimes for each cluster is mentioned below their name.

131
seconds) for each cluster is mentioned below their name in the plot. We currently do not have access to the Intel cluster to conduct new experiments, and the Intel dataset contained very few data from the targeted Wilson Dslash kernel and none from our proxy_app. So, even if we make a prediction using this model, there is no way to validate it. Consequently, we did not conduct this experiment on the Intel architecture and the result is marked as $-\text{NA}$–. As expected on Exxact, the target kernel’s RMSE increased significantly (11.153s) due to less data in its dataset. Even with a normalized RMSE of 0.279, it fell short of our expectation of 0.1. Nonetheless, this model demonstrated some correlation between the actual and predicted data. In contrast, Wombat and Seawulf performed reasonably well and were able to predict the top 10 kernel variants despite having an RMSE of 4.273s and 3.375s, respectively. However, with 0.033 and 0.066, respectively, their normalized RMSE was well within our expectation. As per our observation, their RMSE can also be improved by adding more data for these clusters. Finally, as shown in Figure 8.5, the RMSE rates for Summit, Corona, and Ookami are less than one second each, and they were able to accurately predict the top ten kernel variants.
Chapter 9
Metadirective Extension

OpenMP 5.0 introduces many new directives to meet the demand of emerging high performance computing systems. Among these new directives, the metadirective and declare variant directives are important to control the execution behavior of a given application by compile-time adaptation based on the OpenMP context. The metadirective directive allows the selection of OpenMP directives based on the enclosing OpenMP context as well as on user-defined conditions. The declare variant directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used. Support for these directives are available in few compilers with some limitations.

Support for metadirective is not available in Clang. This paper presents our implementation of the metadirective directive in Clang. In this paper, we present an implementation which supports the OpenMP 5.0 metadirective specification. However, in addition, this work also implements a dynamic extension to the user-specified conditions. A dynamic evaluation of user-defined conditions provides programmers more freedom to express a range of adaptive algorithms that improve overall performance of a given application. For example, a program can estimate the cost of execution, with respect to time taken or energy consumption, of a kernel based on some dynamic or static variables and decide whether or not to offload the kernel to GPU using the metadirective. Since there is a significant lack of knowledge about the usage and performance analysis of metadirective, the work also studies its impact on application characteristics.

To achieve this, we have modified several benchmark codes in the Rodinia benchmark suite. The Rodinia benchmark includes applications and
kernels which target multi-core CPU and GPU platforms which helps programmers study the emerging computing platforms. Our modification to the Rodinia benchmarks enables the application developer to study the behavior of metadirective. Our analysis reveal that the main advantage of the dynamic implementation of metadirective is that it adds minimal to no overhead to the user application, in addition to allowing flexibility to the programmers to introduce portability and adaptability to their code. Our modification of the Rodinia benchmark suite provides several guidelines for programmers to achieve better performance with metadirective.

9.1 Introduction

To meet the demand of productive parallel computing on existing and emerging high performance computing systems, the OpenMP standard has evolved significantly in recent years [29]. Since the creation of the standard in 1997, that specified a handful of directives, substantial amount of new constructs have been introduced and most existing APIs have been enhanced in each revision. The latest version of OpenMP 5.0, released in 2018, has more than 60 directives [25]. Variant directives is one of the major features introduced in OpenMP 5.0 specification to facilitate programmers to improve performance portability. These directives can enable adaptation of OpenMP pragmas and user code at compile and runtime. The OpenMP specification defines traits to describe active OpenMP constructs, execution devices, functionality provided by an implementation, and context selectors based on the traits and user-defined conditions. It also defines variant directives like metadirective and declare variant which uses context selectors to choose various directives or functions.

- The metadirective directive is an executable directive that conditionally resolves to another directive at compile time by selecting from multiple directive variants based on traits that define an OpenMP condition or context.
- The declare variant directive has similar functionality as metadirective but selects a function variant at the call-site based on context or user-defined conditions.

The mechanism provided by these directives for selecting variants is more convenient to use than the C/C++ pre-processing since it directly supports
variant selection in OpenMP and allows an OpenMP compiler to analyze and determine the final directive from variants and context as shown in Figure 9.1.

```c
// code adaptation using preprocessing directives
int v1[N], v2[N], v3[N];
#if defined(nvptx)
#pragma omp target teams distribute parallel for \
    map(to:v1,v2) map(from:v3)
    for (int i= 0; i< N; i++)
        v3[i] = v1[i] * v2[i];
#else
#pragma omp target parallel for map(to:v1,v2) map(from:v3)
    for (int i= 0; i< N; i++)
        v3[i] = v1[i] * v2[i];
#endif

// code adaptation using metadirective in OpenMP 5.0
int v1[N], v2[N], v3[N];
#pragma omp target data map(to:v1,v2) map(from:v3)
#pragma omp metadirective \
    when(device=arch("nvptx"):
        target teams distribute parallel for) \
    default(target parallel for)
for (int i= 0; i< N; i++)
    v3[i] = v1[i] * v2[i];
```

Figure 9.1: C Pre-processing vs OpenMP Metadirective usage comparison.

However, OpenMP 5.0 restricts context traits to be completely resolvable at compile time. This constrains the potential to optimize an OpenMP application based on runtime behavior or input data. An extension to allow runtime adaptation, based on properties like system architecture or input characteristics, may increase the efficiency of an application. Applications that would benefit from this feature include those that use traits based on problem size, loop count, and the number of threads. For example, most libraries parallelize and optimize matrix multiplications depending on the sizes of the input matrix. In this work we explore the possibility of extending metadirective user-condition to dynamically resolve the context selector based on runtime attributes.
The high performance computing community has been exploring novel platforms to push performance forward [146]. Field Programmable Logic Arrays (FPGAs) and Graphics Processing Units (GPUs) have been widely used as accelerators for computational intensive applications. The heterogeneous cluster is one of the most promising platforms as it combines characteristics of multiple processing elements to meet the requirements of various applications. In this context, a self-adaptive framework for heterogeneous clusters would be an ideal solution [10]. OpenMP context based directives like metadirective can provide an excellent solution to build such frameworks.

Current compilers do provide implementation of OpenMP 5.0 with some limitations, but as far as we know only the ROSE [117] source-to-source compiler infrastructure has partial implementation of metadirective [154]. Other compilers which claim active development of OpenMP 5.0 specifications are:

- **Cray** – The lastest CCE 10.0 compiler has partial support for OpenMP 5.0 [53]. However, the list of enhancements claimed by the compiler does not include the metadirective.

- **GNU** – GCC 10 [41] adds a number of newly implemented OpenMP 5.0 features on top of the GCC 9 release such as conditional lastprivate clause, scan and loop directives, order(concurrent) and use_device_addr clauses, if clause on simd construct or partial support for the declare variant directive, and they are getting closer to the complete OpenMP 5.0 standard. However, metadirective is not implemented yet.

- **Intel** – The Intel compiler provides most of the OpenMP Version Technical Report 4, version 5.0 features [58]. Their implementation of OpenMP 5.0 specification is still incomplete and metadirective is not included yet.

The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, OpenMP, and RenderScript) for the LLVM project [83]. Clang is evolving rapidly under the Exascale Computing Project’s (ECP) [92] SOLLVE project [34] and now supports many OpenMP 5.0 features that are essential for ECP applications. These features do not include metadirective yet and to the best of our knowledge, our work is the first one which successfully add OpenMP 5.0 specified metadirective to the Clang framework. In addition to this, we also implement a dynamic extension to the user-specified
The main contributions of our work are:

- Enhancing Clang capability by implementing OpenMP’s metadirective directive in the LLVM framework.
- The implementation supports the compile time selection of hardware support and compiler implementations for code portability across different architectures.
- The work also presents a novel implementation of the semantics for user-defined contexts to enable runtime directive selection using LLVM.
- We also modified the Rodinia benchmarks to study the impact of the directives. These benchmarks can provide a guide line to end users to help them apply these features in real applications.

The remainder of this paper is organized as follows. Section 9.2 presents the motivation for using metadirective and hence the need for it to be supported in compilers, especially in Clang. Section 9.3 gives details of the implementation in Clang. Section 9.4 introduces our experimental setup and Section 9.5 provides, and analyzes, the results of our experiments. Section 9.6 gives related work. Finally, Section 9.7 presents our conclusion and future directions.

### 9.2 Motivation

OpenMP is an integral part of many important HPC codes that rely on hybrid programming models (e.g. MPI + OpenMP). Therefore, tuning an OpenMP code to get better per-node performance for a given resources is an important research problem. In this section, we explain how the metadirective is relevant for tuning resource-constrained OpenMP applications. Whether the OpenMP metadirective is beneficial depends on the response to the following questions:

- Does the best configuration for a given OpenMP region remain the same across different resource levels and workloads?
- Does the performance gain due to the best configuration persist across all configurations?
To answer these questions, we did some experiments with the BT benchmark [8]. We took an OpenMP region from the BT benchmark application and ran it with different power levels or power caps using different numbers of threads, scheduling policies, and chunk sizes (150 different configurations). The region studied belongs to the *xsolve* function, and has coarse grain parallelism, i.e., the outermost loop is parallelized. Figure 9.2 compares the execution time using the best identified configuration with the default configuration at different power levels. The default configuration uses the maximum number of available threads, static scheduling, and chunk sizes calculated dynamically by dividing the total number of loop iterations by the number of threads. The figure clearly shows that the best configuration is different from the default configuration at all power levels.

Figure 9.2: Execution time comparison for the *xsolve* region of BT using different OpenMP runtime configurations at different power levels. Smaller value is better. The function was run on Intel Sandy Bridge [10].

It also shows that the best configuration improves the execution time of the parallel region by up to 20% compared to the default configuration at the same power level. Also, we can see that the best configuration at a lower power level gives better execution time performance than the default configuration with maximum power level prescribed by the manufacturer or
Thermal Design Power (TDP). For example, the best configuration at a 70W power cap improves execution time by almost 9% in comparison to the default configuration at TDP (115W in our case).

We also experimented with OpenMP regions from other NAS Parallel benchmark applications [8] using different runtime configurations. We observed that a significant number of the OpenMP regions showed similar behavior. We observed these OpenMP regions to have poor load balancing and cache behavior with the default configuration. We also saw that these poor behaviors persist across different power levels and workloads for these kernels with the default configuration. As a result, irrespective of power level or workload size an optimal configuration always shows consistent performance improvement compared to the default configuration for these kernels. However, we observed that the optimal configurations for these kernels change across different power levels and workloads.

In the future HPC facility, the application workload may change dynamically. If the facility is operating under a resource constraint, the resource manager may add or remove nodes and adjust their power level/frequency dynamically. To get the best per-node performance for a given resource constraint, the runtime configurations need to be changed dynamically. Our proposed OpenMP metadirective extension of dynamically selecting user-condition gives the user the ability to achieve this efficiently with minimal code change.

```c
for (idev=0; idev<omp_get_num_devices(); idev++) {
    #pragma omp target device(idev)
    #pragma omp metadirective
    when( implementation={vendor(nvidia)}, device={arch("kepler")}: 
    teams num_teams(512) thread_limit(32) ) \
    when( implementation={vendor(amd)}, device={arch("fiji")}: \
    teams num_teams(512) thread_limit(64) ) \
    default(teams)
    #pragma omp distribute parallel for
    for (i=0; i<N; i++) work_on_chunk(idev,i);
}
```

Figure 9.3: Compile time selection of hardware support.
The other strong motivation for metadirective is an efficient portability across different hardware architectures. The key application kernels to be ported on a given device need to be coded specific to the device’s architecture for the best performance gain. The metadirective gives the user ability to write multiple kernels, each specific for different devices, and the compiler then selects the required kernel during the compilation phase based on the available back-end device. Figure 9.3 explains this more clearly. The implementation selector set is specified in the `when` clause to distinguish between AMD and NVIDIA platforms. Additionally, specific architectures are specified with the device selector set. In the code, different teams constructs are employed as determined by the metadirective directive. The number of teams is restricted by a `num_teams` clause and a thread limit is also set by a `thread_limit` clause for vendor AMD and NVIDIA platforms and specific architecture traits. Otherwise, just the teams construct is used without any clauses, as prescribed by the default clause.

### 9.3 Metadirective Implementation in Clang

#### 9.3.1 Metadirective

The syntax of metadirective is shown in Figure 9.4, where clause is one of the `when` clause or the `default` clause as defined. In the when clause, context-selector-specification specifies a context selector. In the when and default clauses, directive-variant has the form shown Figure 9.4. It specifies a directive variant that specifies an OpenMP directive with clauses that apply

<table>
<thead>
<tr>
<th>Syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp metadirective [clause[[,] clause] ... ] new-line</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metadirective Clauses:</th>
</tr>
</thead>
<tbody>
<tr>
<td>when(context-selector-specification: [directive-variant])</td>
</tr>
<tr>
<td>default(directive-variant)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Directive-variant:</th>
</tr>
</thead>
<tbody>
<tr>
<td>directive-name [clause[ [,] clause] ... ]</td>
</tr>
</tbody>
</table>

Figure 9.4: Metadirective Syntax
A metadirective, by definition [25], is a directive that acts as though it is either ignored or substituted by the directive variant specified in one of the \texttt{when} or \texttt{default} clauses appearing on the metadirective. Whether a particular directive is selected or not depends upon the context selector as specified in the when clauses.

\section*{9.3.2 Context Selector}

The syntax of a context selector specification is shown in Figure 9.5. A trait-selector can be one of the following:

1. construct
2. device
3. implementation
4. user

The construct selector set defines the construct traits that should be active in the OpenMP context. We can define the following selectors in the construct set: \texttt{target, teams, parallel, for} and \texttt{simd}. Each selector’s properties are the same properties that are specified for the respective trait.

The device set includes traits that define the characteristics of the device.

\begin{verbatim}
context-selector-specification:
    trait-set-selector[,trait-set-selector[,...]]

trait-set-selector:
    trait-set-selector-name={trait-selector[, trait-selector[, ...

trait-selector:
    trait-selector-name=([trait-score: ] trait-property[, trait-property[, ...]])

trait-score:
    score(score-expression)
\end{verbatim}

Figure 9.5: Context Selector Syntax
being targeted by the compiler at that point in the program. A device set can have one of the following trait properties:

1. **kind**: This property specifies the general kind of the device - host, nohost, cpu, gpu, fpga.
2. **isa**: This property specifies the Instruction Set Architectures supported by the device which is implementation defined.
3. **arch**: This property specifies the architectures supported by the device.

The implementation set includes traits that describe the functionality supported by the OpenMP implementation at that point in the program. An implementation set can have one of the following trait property:

1. **vendor**: This property specifies the vendor identifiers of the implementation.
2. **extension**: This property specifies vendor specific extensions to the OpenMP specification.
3. A trait with a name that is identical to the name of any clause that can be supplied to the required directive.

The user selector set defines the condition selector that provides additional user-defined conditions.

For each when clause that appears in a metadirective, the specified directive variant, if present, is a candidate to replace the metadirective, if the corresponding context selector is compatible with the OpenMP context. A given context selector is compatible with a given OpenMP context if the following conditions are satisfied:

1. All selectors in the user set of the context selector are true. (See Section 9.3.5 for our extension to this rule).
2. All selectors in the construct, device, and implementation sets of the context selector appear in the corresponding trait set of the OpenMP context.
3. For each selector in the context selector, its properties are a subset of the properties of the corresponding trait of the OpenMP context.
4. Selectors in the construct set of the context selector appear in the same relative order as their corresponding traits in the construct trait set of the OpenMP context.
9.3.3 Clang/LLVM

For our implementation\(^1\), we used the current development version of llvm monorepo (version 12.0.0) from their git repository - https://github.com/llvm/llvm-project. LLVM follows the popular three phase design (Figure 9.6) for a traditional static compiler, whose major components are the front end, the optimizer and the back end.

![Common 3-Phase Compiler Design](image)

Clang is the frontend for C language family compilers for LLVM. Clang parses the source code, checks it for errors, and builds an Abstract Syntax Tree (AST) to represent the input code. This AST is then translated into an intermediate LLVM IR, upon which all LLVM optimizations are performed and finally the backend translates this to generate the machine code.

A core design principle for clang is the use of a library-based architecture, where different parts of the frontend are separated cleanly into several libraries, which can then be combined for different needs and usages. This approach encourages good interfaces and makes it easier for new developers to get involved. To implement a new directive, like metadirective, we need to modify several of these libraries.

We primarily updated the following clang libraries:
1. **libast** - This library provides classes to represent the AST node and various helpers for analyzing and manipulating the AST.
2. **libparse** - The parsing library invokes coarse-grained 'Actions' provided by the libsema.
3. **libsema** - Semantic analysis provides a set of parser 'Actions' to build a standardized AST for programs.

\(^1\)Github link - https://github.com/almishra/metadirective.git. A patch is submitted to the LLVM mono-repo
4. **libcodegen** - This library lowers the AST to LLVM IR for optimization and code generation.

### 9.3.4 Design

The major design challenge lies in the parsing and evaluating the context selector. We do this in `libparse`, where we parse the context selector and save all the information in a data structure (OMPTraitInfo) which represents all the traits. The outer level of this data structure is an ordered collection of selector sets, each with an associated kind and an ordered collection of selectors. A selector has a kind, an optional score/condition, and an ordered collection of properties. We save this information to be used by `libsema` to make the decision on which directive variant need to be selected. In `libsema`, we check the kind of the selector and its properties. If the selector is of the kind **user_condition**, we read its condition and try to resolve it into an integer constant expression. If the expression is true, the user condition passes successfully. Otherwise, it is not selected. For all other selector kind, we compare the properties associated with the selector with the compiler parameters. For instance, if the context selector in a when clause is specified as `device={arch("nvptx")}`, then its selector kind will be `device_arch` and property will be "nvptx". In this case we will resolve the selector based upon whether the architecture of the device for which the code is being built is "nvptx" or not.

Based on the resolution of the context selector, we decide at compile time which directive variant need to be selected. If only one compatible context selector specified by a when clause has the highest score and it specifies a directive variant, the directive variant will replace the metadirective. If more than one when clause specifies a compatible context selector that has the highest computed score and at least one specifies a directive variant, the first directive variant specified in the lexical order of those when clauses will replace the metadirective. If no context selector from any when clause is compatible with the OpenMP context and a default clause is present, the directive variant specified in the default clause will replace the metadirective. We parse the statement associated with metadirective in the context of the selected directive variant. If no directive variant is selected to replace a metadirective according to the above rules, metadirective has no effect on the execution of the program.
9.3.5 Dynamic User Condition

Yan et. al. explored a strategy for extending the metadirective [154] by relaxing its restriction to compile-time only selection and allowing runtime evaluation of user defined conditions. They implemented this in the ROSE compiler [117]. We have implemented the same feature in LLVM to establish that dynamic selection of user defined conditions provides greater flexibility to users. By definition, all the context selectors in metadirective need to be resolved at compile time. Based upon the compatibility of the context selector, as defined above in Section 9.3.2, the compiler will replace metadirective with the directive variant specified by the user. In this extension, we extend metadirective to resolve a user condition at runtime. So instead of substitut-

```c
// Code written using metadirective in OpenMP 5.0
#pragma omp metadirective
  when(user={condition(N>1000)}): target teams distribute parallel for
  when(user={condition(N>100 && N<=1000)}): parallel for
  default()
  for(int i=0; i<N; i++)
    compute();

«*****

// Code resolved as if-then-else statement
if(N>1000) {
  #pragma omp target teams distribute parallel for
  for(int i=0; i<N; i++)
    compute();
} else if(N>100 && N<=1000) {
  #pragma omp parallel for
  for(int i=0; i<N; i++)
    compute();
} else {
  for(int i=0; i<N; i++)
    compute();
}

Figure 9.7: OpenMP Metadirective Dynamic User Selection.
```
ing metadirective with the given directive variant, we create an if-then-else statement whose condition will be evaluated and resolved during execution. For this, we parse all the when and default clauses in the metadirective and create statements for each of the directive variants specified in the clause. An example of this resolution is shown in Figure 9.7.

The major design challenge lies in the implementation of the dynamic user selection code. In the static case, we only need to parse and evaluate the context selector. Based upon the outcome, we generate just one directive statement. In the dynamic case, we need to parse all the when and default clauses and generate a directive statement for each of the valid clauses, and build a corresponding if-then-else statement for it. After careful consideration of the above mentioned libraries, we designed our implementation as follows:

1. In libast, we define the classes corresponding to the AST nodes for

```c
if(N<10) { // Block 1
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            for(int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
} else if (N<100) { // Block 2
    #pragma omp parallel for collapse(2)
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            for(int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
} else { // Block 3
    #ifdef NVPTX
    #pragma omp target teams distribute parallel for
    #else
    #pragma omp parallel for collapse(2)
    #endif
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            for(int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

Figure 9.8: Matrix Multiplication implementation without metadirective
metadirective directive and when & default clauses.

2. In libparse, we parse the AST node and collect information related to all the when and default clauses and the statement associated with metadirective. For each when clause we parse and store the context selector. For each when and default clause we parse the directive variant and create a statement corresponding to this directive variant and store it. If no directive variant is provided, we store the statement associated with metadirective as it is.

3. In libsema, we gather all the stored directive variants and create the if-then-else statement based upon the condition parsed in the context selector. We store this statement in the AST node of metadirective.

4. In libcodegen, we read the AST node for metadirective and generate LLVM IR code corresponding to the stored if-then-else statement.

9.3.6 Example of Dynamic Metadirective

Let us consider a well known scenario where a programmer is writing a library function to multiply two matrices, where the size of the matrix will only be known at runtime. If the size of the matrices are small (< 10), the programmer might want to run the multiplication in serial to avoid parallel overhead. If the size of the matrices are fairly large (>= 10 && < 100), programmers might want to run the computation in parallel while collapsing the two loops on the CPU. But if the size of the matrices are larger (> 100), and the code can be built for offloading to an NVIDIA GPU, then programmers might want to offload the computation onto the GPU, else continue with the parallel execution on CPU. The resulting code for such a scenario will look like shown in Figure 9.8.

Since the size of the matrix can only be resolved at runtime, there is no way the programmer can use only preprocessing directives to take the decision of which block to compile. They need to write the same block of code multiple times, with different directives. Even with static metadirective code, unfortunately the programmer will only be able to use metadirective in the last else statement (Block 3 in the Figure 9.9). But with dynamic metadirective programmers can write a much cleaner code (Figure 9.10) without replicating the blocks multiple times. The code in Figure 9.10 will be resolved and substituted by the compiler to act like the code in Figure 9.8.

In our implementation of dynamic metadirective we have made sure that all properties of static metadirective as mentioned in the OpenMP 5.0 spec-


9.4 Experimentation

To test and evaluate our implementation, we used the Summit Supercomputing Cluster at the Oak Ridge Leadership Computing Facility [109] and the SeaWulf computational cluster at Stony Brook University [144]. Summit has a hybrid architecture, where each node contains 2 IBM POWER9 CPUs and 6 NVIDIA Volta (V100) GPUs all connected together with NVIDIA’s high-speed NVLink. We use GCC version 6.4.0 and CUDA version 10.1.105

```c
if(N<10) {
    // Block 1
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            for(int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
} else if (N<100) {
    // Block 2
    #pragma omp parallel for collapse(2)
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            for(int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
} else {
    // Block 3
    #pragma omp metadirective
        when(device=arch("nvptx64")):
            target teams distribute
                    parallel for
        default(parallel for collapse(2))
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            for(int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

Figure 9.9: Matrix Multiplication implementation with static metadirective
```c
#pragma omp metadirective
   when(device={arch("nvptx64")}, user=condition(N>=100): \
       target teams distribute parallel for) \
   when(user={condition(N>=10 & N<100)}: \
       parallel for collapse(2)) \
   default()
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++)
    for(int k=0; k<N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Figure 9.10: Matrix Multiplication implementation with dynamic metadirective

to build our implementation of LLVM. A Seawulf node, on the other hand, contains 2 Intel Xeon E5-2683v3 CPUs and 4 Nvidia Tesla (K80) GPUs all interconnected via a high-speed InfiniBand® network by Mellanox® Technologies. We use GCC version 6.5.0 and CUDA version 9.1.185 to build our implementation of LLVM.

We compiled the three codes as shown in Figure 9.8, 9.9 and 9.10 on Summit to be executed using 1 NVIDIA V100 GPU. The size of the executable without metadirective was 204448 bytes, with static metadirective was 204456 bytes and with dynamic metadirective was 204712 bytes. We can observe that the overhead on the size of executable for dynamic metadirective is minimal. When we ran these executables for varying input sizes, we found

```
5 50 100 500 1000
0 1 2 3
```

Figure 9.11: Execution time (in sec) of matrix multiplication for 3 different implementations

there was no overhead added due to the use of static or dynamic metadirective.
tive as can be seen in Figure 9.11. For the remainder of the experiments we only used our dynamic implementation of metadirective.

We modified benchmark applications\(^2\) from the Rodinia Benchmark Suite [19] to use our dynamic metadirective implementation. We also implemented four microbenchmarks to represent several common methods: Matrix Multiplication, Gauss Seidel Method, Laplace Equation and SAXPY. All the benchmarks used in our experiments are listed in Table 9.1. The Rodinia Benchmark Suite was chosen primarily because of the diversity of the domains in which each of its applications falls. They are also intended to help the application developer learn how to use dynamic metadirective in real applications from different domains. We built these application for compute capability 7.0 on Summit (for Volta V100) and 3.5 on SeaWulf (for Tesla K80). An example of one such implementation can be seen in Figure 9.8 and Figure 9.10, which shows a comparison of matrix multiplication implementations using preprocessing directives and the metadirective, respectively. All benchmarks were modified to replace preprocessing directives with dynamic metadirective.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{B+Tree} ( (b+tree) ) \cite{66}</td>
<td>\textbf{Search} domain. This is a data structure used as an index to facilitate fast access to the elements of a larger body of data.</td>
</tr>
<tr>
<td>\textit{Back Propogation} ( (backprop) ) \cite{102}</td>
<td>\textbf{Pattern Recognition} domain. is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network.</td>
</tr>
<tr>
<td>\textit{Breadth First Search} ( (bfs) ) \cite{48}</td>
<td>\textbf{Graph Algorithm} domain. This benchmark provides the GPU implementations of BFS algorithm which traverses all the connected components in a graph.</td>
</tr>
<tr>
<td>\textit{Heart Wall} ( (heartwall) ) \cite{137}</td>
<td>\textbf{Medical Imaging} domain. This application tracks the movement of a mouse heart over a sequence of 104 609×590 ultrasound images to record response to the stimulus.</td>
</tr>
<tr>
<td>\textit{Hotspot} \cite{56}</td>
<td>\textbf{Physics Simulation} domain. We re-implemented the transient differential equation solver from HotSpot using target offloading directives for GPU.</td>
</tr>
</tbody>
</table>

\(^2\)Github - https://github.com/almishra/rodnia_3.1_meta.git
| **K-means**  
|  
| (kmeans) [115]  
|  
| **Data Mining** domain. K-means is a clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity. Many data-mining algorithms show a high degree of data parallelism.  
|  
| **k-Nearest Neighbor**  
| (knn) [37]  
|  
| **Data Mining** domain. In the implementation of this application, it finds the k-nearest neighbors from an unstructured data set.  
|  
| **Lava MD**  
| (lavamd) [136]  
|  
| **Molecular Dynamics** domain. The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space.  
|  
| **LU Decomposition**  
| (lud)  
|  
| **Linear Algebra** domain. This benchmark is a good example where multiple kernels care called from within a loop and some data shared by these kernels are also used on the host.  
|  
| **Needleman Wunsch**  
| (nw) [119]  
|  
| **Bioinformatics** domain. Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments.  
|  
| **Particle Filter**  
| (p-filter) [42]  
|  
| **Medical Imaging** domain. This particular implementation is optimized for tracking cells, particularly leukocytes and myocardial cells.  
|  
| **Path Finder**  
| (p-finder)  
|  
| **Grid Traversal** domain. PathFinder uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead.  
|  
| **Speckle Reducing Anisotropic Diffusion**  
| (srad) [155]  
|  
| **Image Processing** domain. This is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs).  
|  
| **Stream Cluster**  
| (stream) [13]  
|  
| **Data Mining** domain. For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric.  
|  
| **151**
### Three Matrix Multiplication (3mm)

This is the most basic implementation of multiplying three large matrices. This is a benchmark where two kernels are reusing same data. The experiment used matrices of size $5000 \times 5000$ each.

### Gauss Seidel Method (gauss)

The method for solving linear equations is an iterative method, in which the values for the given variables keep changing until a certain threshold of variance is reached. The experiment used a matrix of size $2^{13} \times 2^{13}$.

### Laplace Equation (laplace)

The equation in two dimensions with finite differences using Jacobi iteration. The experiment used a matrix of size $2000 \times 2000$.

### Single-Precision A·X Plus Y (saxpy)

SAXPY is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. In its simplest form this is a benchmark where two kernels are reusing same data. The experiment used two vectors of size $2^{27}$ each.

<table>
<thead>
<tr>
<th>Table 9.1: Updated benchmarks from the Rodinia Benchmark Suite</th>
</tr>
</thead>
</table>

### 9.5 Results and Analysis

For each of the application benchmark we have two codes – the original code with preprocessing directives and the modified code using dynamic metadirective. In the original code, the preprocessing directives are processed at compile time and based upon the result, only one particular code version is selected and compiled. In our modified code with dynamic metadirective, based on the condition provided by the user, we generate multiple blocks of code for each if-then-else statement. This will obviously increase the size of our code. In order to assess the impact on code size, we built both versions of all the benchmark applications listed in Table 9.1 using Clang on both Summit and SeaWulf. Summit has IBM Power9 CPUs and NVIDIA Volta V100 GPUs. For GPU codes LLVM, used CUDA version 10.1.105 in the backend. Also the applications are built for compute capability of 7.0. Seawulf has Intel Xeon E5-2683v3 CPUs and NVIDIA Tesla K80 GPUs. For the GPU codes, LLVM used CUDA version 9.1.85 in the backend on this
Figure 9.12: Percentage increase in size of the executable built with our dynamic metadirective implementation in LLVM 12.0.0 platform. Also the applications are built for compute capability of 3.5. Due to the difference in architecture and backend, the executables for the original benchmark code already vary in size, as can be seen in Table 9.2.

As expected there is some increase in the size of the executable for our dynamic metadirective implementation, depending upon the size of the kernel
which is generated multiple times. But by how much? This question can be
answered by looking at Figure 9.12 which shows the percentage increase in the
size of executable of the original and dynamic metadirective codes. The size
of all the executables on Summit are in the range of 70 kB to 980 kB, while
on Seawulf they are in the range of 12 kB to 622 kB. For some applications
like bfs, hotspot, k-nn, gauss and jacobi this change is less than 0.01%, hence
shown as 0% in the figure. On Summit the maximum difference is 4.94%,
while on SeaWulf it is 10.08%. The maximum difference on both platforms
is in the Kmeans application, which has fairly larger kernels compared to
the other benchmark applications analyzed. Overall, the change in the size
of the executable will be directly related to the size of the kernels for the
application and the number of when clauses used with user defined context
selector.

But does using dynamic metadirective effect the actual runtime of the

<table>
<thead>
<tr>
<th>Application</th>
<th>Summit Orig</th>
<th>Summit Metadirective</th>
<th>Seawulf Orig</th>
<th>Seawulf Metadirective</th>
</tr>
</thead>
<tbody>
<tr>
<td>B+tree</td>
<td>77440</td>
<td>77736</td>
<td>50640</td>
<td>53944</td>
</tr>
<tr>
<td>Backprop</td>
<td>114576</td>
<td>115432</td>
<td>63960</td>
<td>64200</td>
</tr>
<tr>
<td>BFS</td>
<td>735512</td>
<td>735544</td>
<td>438696</td>
<td>438752</td>
</tr>
<tr>
<td>Heartwall</td>
<td>76536</td>
<td>76800</td>
<td>70288</td>
<td>70472</td>
</tr>
<tr>
<td>Hotspot</td>
<td>270640</td>
<td>270648</td>
<td>146040</td>
<td>146040</td>
</tr>
<tr>
<td>Kmeans</td>
<td>94784</td>
<td>99464</td>
<td>43248</td>
<td>47608</td>
</tr>
<tr>
<td>LavaMD</td>
<td>72288</td>
<td>72552</td>
<td>17632</td>
<td>17808</td>
</tr>
<tr>
<td>LUD</td>
<td>73976</td>
<td>74280</td>
<td>23040</td>
<td>23040</td>
</tr>
<tr>
<td>Needleman</td>
<td>1003360</td>
<td>1003864</td>
<td>636856</td>
<td>637376</td>
</tr>
<tr>
<td>NN</td>
<td>72640</td>
<td>72640</td>
<td>13160</td>
<td>13672</td>
</tr>
<tr>
<td>ParticleFilter</td>
<td>73544</td>
<td>73544</td>
<td>31104</td>
<td>31104</td>
</tr>
<tr>
<td>PathFinder</td>
<td>71920</td>
<td>72184</td>
<td>13344</td>
<td>13520</td>
</tr>
<tr>
<td>SRAD</td>
<td>72656</td>
<td>72920</td>
<td>21920</td>
<td>22104</td>
</tr>
<tr>
<td>StreamCluster</td>
<td>77576</td>
<td>77832</td>
<td>38384</td>
<td>38560</td>
</tr>
<tr>
<td>Gauss</td>
<td>260560</td>
<td>260560</td>
<td>137680</td>
<td>137680</td>
</tr>
<tr>
<td>Jacobi</td>
<td>401808</td>
<td>401808</td>
<td>207664</td>
<td>207672</td>
</tr>
<tr>
<td>MM</td>
<td>105024</td>
<td>105032</td>
<td>63936</td>
<td>64072</td>
</tr>
<tr>
<td>SAXPY</td>
<td>204032</td>
<td>204264</td>
<td>104624</td>
<td>106712</td>
</tr>
</tbody>
</table>

Table 9.2: Size of executables on Summit and Seawulf (in bytes)
Figure 9.13: Percentage change in running time of the executable built with our dynamic metadirective implementation

application? To answer this question we executed both the original code and our modified code 10 times and collected the average runtime for each application. The result can be seen in Figure 9.13 which shows the percentage change in the runtime of the modified code when using the dynamic

metadirective from the original code. The difference in runtime is almost negligible for all the application on both the platforms, which could easily be caused due to multiple hardware or environmental reasons. On Summit the maximum difference of the time is -1.76% in the SAXPY application, whose runtime changed from 1.705s to 1.675s. Whereas on Seawulf, the maximum change is -2.07% for the same application, where the runtime changed from 24.443s to 23.937s. Looking at this result we can conveniently conclude that there is very minimal to no runtime overhead added to the application due to dynamic metadirective.

9.6 Related Work

The OpenMP 4.5 version has been relatively well supported in the LLVM/-Clang framework. OpenMP 5.0 has introduced many new features especially related to accelerated devices to take care of heterogeneity in the future HPC machines. These new features are being actively added in LLVM/Clang under the ECP SOLLVE project [34]. The site, https://clang.llvm.org/docs/OpenMPSupport.html, gives up to date feature support in Clang. The SOLLVE team has also developed an OpenMP verification and validation suite to support and test the OpenMP 5.0 features [32]. The new OpenMP features are heavily towards accelerated devices. Performance wise, the IBM team has implemented several optimizations to have a decent GPU offloading performance. For instance, GPU device runtime has been inlined to reduce the register pressure and increase concurrency; the default thread scheduling policy has been changed for GPU to suit for its execution model; adaptive thread organization is implemented to coalescing memory accesses of different threads, and etc.

There are still many things that can be done to optimize the performance of OpenMP GPU offloading implementation, several OpenMP features are still missing. Many new GPU features such as unified memory feature provided by Kepler and later GPUs are supported in LLVM/Clang [77, 78]. However, these new features are not effectively used in the LLVM backend optimizations for OpenMP. It is important for the compiler to be aware and to leverage these new features for better performance. Currently, OpenMP dialects using the MLIR framework [76] are trying to fill this performance gap.

The work [154] uses the Smith-Waterman algorithm as an example to
show the need for adaptive selection of parallelism and devices at runtime, and present a prototype implemented in the ROSE compiler using OpenMP metadirective. Pennycook et al. [113] the benefits of using two OpenMP 5.0 features, including metadirective and declare variant, for the miniMD benchmark from the Mantevo suite. The authors concluded that these features enabled their code to be expressed in a more compact form while maintaining competitive performance portability across several architectures. However, their work only explored compile-time constant variables to express conditions. Many researchers have studied using GPUs to speedup Smith-Waterman algorithm, beginning as far back as Liu et al. in 2006 [81]. Our implementation resembles some of these early attempts in terms of data motion and synchronization behavior, mainly as a simple case study. Later work uses a variety of techniques to reduce the data movement and memory requirement by doing backtracing on the GPU [152] and even exploring repeating work to accomplish the backtrace in linear space [122]. These techniques would likely make the inner-loop optimization we discussed more attractive by removing the high cost of moving the complete cost matrix to and from the device, and may be worth exploring in the future.

9.7 Conclusion and Future work

OpenMP 5.0 introduces the metadirective directive that conditionally resolves to another directive at compile time by selecting from multiple directive variants based on traits that define an OpenMP condition or context. We have implemented this feature in LLVM for general use.

OpenMP 5.0 restricts the selection to be determined at compile time, which requires that all traits must be compile-time constants. Our analysis of real applications indicates that this restriction limits the usefulness of metadirective, so we explored an extension of user-defined contexts to allow directive variant selection at runtime. We successfully modified the open source LLVM compiler to extend the user-defined condition of context selector, as used in metadirective, to be resolved at runtime. A dynamic evaluation of user-defined conditions can provide more flexibility for programmers to express a variety of adaptive algorithms that boost overall performance.

To study its impact we have modified 14 benchmarks application in the Rodinia benchmark suite and 4 common micro benchmark applications. Rodinia includes applications and kernels which target multi-core CPU and
GPU platforms across a variety of domains in computer science. Our modifications to the Rodinia benchmark suite enabled us to explore the impact of dynamic metadirective in an OpenMP context. It also provides a guideline to the end users to help them apply these features in real applications. The results of our evaluation reveal that dynamic user condition can easily be used by programmers to achieve better portability and adaptability in their code.

The main advantage of the dynamic metadirective is that it adds minimal or no overhead to the user application. In addition to allowing flexibility to the programmers to introduce portability and adaptability to their code, it can enable research aimed at automatic code generation and computation on heterogeneous devices, such as Mishra et al’s [95] work on automatic GPU-offloading, or Mendonça et al’s [90] work on automatic parallizer or Poesia et al’s [116] work on static placement of computation on heterogeneous devices. All these efforts take decisions based on compile-time analysis and can be significantly assisted by a dynamic metadirective, which would further reduce the barriers to use of GPUs for scientific computing. Potential future extensions of this work include the following:

- Provide optimization of selected code and help better data handling between heterogeneous devices. This can be an extension to Mishra et al’s [97] work on data reuse analysis.
- Explore more complex user-defined conditions, where a function can be called from a user condition.
- Adding template definition to metadirective for type manipulation.
Chapter 10

Future Works

“We can only see a short distance ahead, but we can see plenty there that needs to be done.”

– Alan Turing in Computing Machinery and Intelligence [142]

This beautiful quote by Alan Turing is the closing statement of an article published in the British journal Mind in 1950. “Can machines think?” he asked in 1950, and his simple question changed the way people thought about computing. Today, the horizon has changed, but Turing’s words remain true. Computing, machine learning, and artificial intelligence have not only blossomed, but exploded, seven decades later. And no work is ever finished in this rapidly changing environment. There is plenty there that needs to be done.

Writing applications for future heterogeneous systems that use accelerators, like GPUs, necessitates thorough familiarity with the underlying architecture, the algorithm, and the programming model for the interfacing. In this research, we introduced the OpenMP Advisor, a practical implementation of a compiler framework that can automatically identify OpenMP kernels, suggest a number of possible OpenMP variants for offloading that kernel to the GPU, and predict the cost of running each of these kernels individually using a novel Machine Learning-based compile time cost model. Using our Advisor, we were able to generate several variants of offloading code to the GPU for multiple micro-benchmark applications as well as the Wilson Dslash operator, which is used in the ECP’s LQCD application.

However, this is an evolving project that has the potential to develop into something incredible. This thesis provides a proof of concept for the
idea that compiler developers can train and use an ML model to help them choose more judiciously when offloading a kernel to a GPU using OpenMP. Future extensions of the tool could support numerous compiler optimizations and code generation techniques in addition to GPUs and other heterogeneous devices. One major downside of this work is that the cost model used in this tool requires knowledge of the size of the data structure and the number of iterations used at the time of compilation. If the size is unknown at the time of compilation, the Artificial Neural Network (ANN) model we used for our Advisor is likely to fail, and any prediction will require a combination of better model and compiler techniques like using metadirective as discussed in Chapter 9.

Apart from that, there are a number of ways to broaden on this work. We have identified four areas where this work could be expanded:

1. Future Models
2. Data Collection
3. More compiler optimization
4. IDE tools and plugins development

10.1 Future Models

We based our model on Artificial Neural Network model which trains on data collected by us and predicts the time it will take to execute a kernel on a GPU. For our model, since the runtime is a positive real-valued variable, the prediction task is a regression problem. We tried several predefined regression models, like the Linear Regression, Support Vector Machines (SVM) and Artificial Neural Networks. We chose ANN over SVMs because they can capture even more complex data relationships while being easier to parameterize and converge using gradient-based activation functions. The class we employ is fully-connected feed-forward networks, also called Multi-Layer Perceptrons (MLP), with six layers – one input, four hidden and one output neurons. The model’s parameters were fine-tuned through several rounds of trial and error training. As a proof of concept this was a successful experiment.

However, this is obviously not a foolproof solution, as evident by the long-tail problem which arises especially when a new benchmark for prediction is introduced. An easy fix of this problem is to use more data. Although
this kind of solution is crucial for the development of machine learning, it is extremely expensive in terms of time, compute, and data labeling. Additionally, it does not ensure that the new data won’t reintroduce such a problem. The cost of increasing long tail performance frequently jeopardizes the viability of the economics of AI products. Even though it’s somewhat necessary, merely adding more data won’t make these issues go away. We need to develop smarter and better ML models to predict the runtime of a kernel. We already have two new models proposed that could improve the predictions. Based on the use case scenario, more such models could be developed in the future.

10.1.1 Classification and Regression

![Dataset Splitting Approach](image)

In this first model, we can improve on the ML strategy developed in COMPOFF by employing various ML modeling techniques and a recursive data splitting approach to split training data based on kernel runtime. The recursive approach divides the dataset and represents it as a tree structure with branches created to minimize error, as shown in Figure 10.1. As a first proof of concept, we evaluated three ML model regressors:

1. Multi-Layer Perceptron (MLP) used in COMPOFF
2. Random Forest
3. Decision Tree
A separate model was trained for each leaf of the dataset tree. After optimizing the parameters for each model, we observed that both Random Forest and Decision Tree outperformed MLP. For the purposes of this experiment, we only used a partial dataset that lacked datapoints for data transfer. According to the experimental results, an MLP had a 17% error rate, while the random forest and decision tree both had around 1.2% error rate. Our success in reducing the error demonstrates that ML technologies have enormous potential for future work in developing cost models for complex compiler optimization problems.

10.1.2 Graph Neural Network

Another model that we propose uses a graph neural network (GNN) that has been trained to create the model and the prediction. It is a challenging task in and of itself for a compiler developer to write a plugin to count the number of computations in COMPOFF. In order to train the GNN model, we propose a technique to represent an AST of a kernel as a weighted graph representation of programs that captures properties specific to HPC kernels. This task will not only be simpler, but it may also be used in the future to get around the issue of having to know the size of the data structure and the number of iterations at compile time. The AST of a kernel contains a lot of data about the kernel that a compiler can understand, but it can be challenging to extract information from it if something (like an ML model) does not understand how to properly parse it. Even for COMPOFF, we had to create a new plugin in order to extract computation information. Similar to this, a GNN model cannot be given a simple AST because it cannot learn anything from it. In order to solve this problem, additional nodes, edges, and attributes are added to the AST in to generate a graph representation of programs that includes information about HPC kernel properties. In the past, a number of works have proposed ideas on how ASTs can be augmented to better present semantically information [2, 148, 3]. Although, these program representations were effective to train deep learning models for downstream tasks such as variable misuse detection or clone detection, to the best of our knowledge there does not exist any work to target loops and if conditions. This program representation is based on AST and includes extra attributes to preserve semantic and syntactic information as well as present the number of loops’ iterations and if statements in a more elegant way by adding weights to the edges. These edge weights also contain implicit data about
the workload distribution across teams and threads. All these augmentations can be applied statically.

**Augmenting Abstract Syntax Tree**

![Diagram of AST augmentation](image)

Figure 10.2: Modification to AST to create Augmented AST for ParaGraph

ASTs typically provide a program’s detailed structural and syntactical information. Although a compiler would find these details quite informative, a neural network would benefit very little from an AST. It has been demonstrated that it is possible to avoid wasting the capacity of deep learning models to learn obvious facts by adding additional attributes and information to an AST. To represent program control and data flow, we add the following edges to AST, as shown in Figure 10.2.

- **AST Edge** represent the parent-child relationship between the AST nodes and are standard AST edges.
• **NextToken** that connect each syntax node to the one after it in order to represent this information in a graph. This imposes an order on syntax nodes by using the breadth-first traversal method, which is similar to how a compiler walks the AST.

• **NextSib** connects each syntax node to its sibling on the right.

• **Ref** connect a DecRefExpr to where the corresponding variable is defined, in order to present where the used variable is declared in the code.

• **ForExec** connects the ForStmt condition node, the counter initialization node, and the loop body in the sequence of how the next iteration of the loop will be executed.

• **ForNext** connects the ForStmt's body node to the counter modifier node.

• **ConTrue** indicates that the flow will move to the connected child if the condition of the IfStmt is true.

• **ConFalse** indicates that the IfStmt condition is not met.

• **Weighted Edges** are added only to Child edges, since these are AST edges, and compiler uses the AST nodes and child edges to transform AST to lower-level for machine execution. Weights of edges are calculated based on the region and edge type. For instance, whenever we encounter a loop in a program, the body of the loop could be executed multiple times. Or the branches of an if statement will now execute the same number of items.

We evaluate our proposed representation as an initial proof of concept by training a Graph Neural Network (GNN) to predict the runtime (in seconds) of an OpenMP code region. We take the runtime from the COMPOFF dataset and add graph information to it. The model’s predicted runtime is used to determine which transformation offers the best performance. We were able to achieve similar performance to COMPOFF with few experiments, indicating that this model will work just as well, if not better, in the future.

### 10.2 Data Collection

Over the past few decades, artificial intelligence has taken over as the driving force behind all aspects of modern life, from grocery shopping to the creation
of spacecraft. However strong it may now be, its foundation will always be in the hands of data scientists with enough training data for machine learning. Machine learning begins with the process of providing real data and algorithms to artificial intelligence (AI) so that it can act like humans and learn from its own operating experience, much like we all do throughout our lifetimes. However, there is a significant problem with the lack of data in machine learning for compilers. For analysis, training, and performance in AI, a certain amount of data is essential; otherwise, a trustworthy model cannot be trained.

This poses a significant problem for compilers. Despite the fact that ML has been used in compilers for a while, there is a severe lack of publicly accessible data for compilers. Consequently, in the absence of data, any new compiler optimization based on machine learning would also need to first gather its own set of data before training its model.

Lack of training data is a serious problem because, even if AI is uncertain about the prediction, it won’t indicate this uncertainty and will proceed with the prediction normally. And that’s where things get complicated. Lack of data leads to more unanswered questions about AI and subpar results. As in our situation, it makes us realize that it is impossible to prepare data for machine learning without access to GPUs, and that even the best software will be useless without sufficient data filling. No access also signifies that the data is either nonexistent or too difficult to obtain. Because compiler developers have limited access to GPUs for gathering and preparing data for machine learning, promising ML-based compiler optimization often doesn’t work out as well as anticipated. Additionally, since there is no definite answer, the question regarding the right amount of data will never become irrelevant.

But collecting more data is not a straightforward solution. The two models covered in Section 10.1.1 and 10.1.2 could be worked on more quickly because we already had the data or partial data available. The first model (Section 10.1.1) made use of the exact same dataset as our COMPOFF model. In the second model (Section 10.1.2), the feature collection step was modified to train a GNN rather than an ANN. However, it made use of the COMPOFF dataset’s runtime, which took the longest time to collect. Depending on the optimization or prediction that each compiler project wants to make, a dataset of a specific type is required. Therefore, we require a project that will collect data for numerous benchmark applications with numerous optimizations. More research must be done on the best ways to collect these data and make them accessible to the public for future use.
10.3 More compiler optimization

This thesis concentrated on simple GPU offloading that only took into account one GPU. Additionally, we took into account data synchronization between host and device only prior to and following kernel execution. Even with GPU offloading, we only focused on the following variations for our optimization:

- Value of collapse used in `distribute` directive
- Value of collapse used in `parallel for` directive
- Position of the `parallel for` directive
- Scheduling type of the loop iterations
- Data transfer between the host and the device

However, there are numerous variations that can be used for GPU offloading. The following tasks could be added to this research as a next step:

1. Data synchronization between host and device during kernel execution.
2. Offload computation to multiple GPUs [69] via tasks.
3. Check for kernels called from recursive functions.
4. Our cost model can currently predict the runtime of kernels whose data size and level of parallelism are known at compile time. We could extend our model to predict the best variants for a variety of data sizes, and then use the OpenMP metadirective [98] directive to generate multiple directive variants for each range.
5. According to previous studies [96, 78], OpenMP GPU offloading performance can be enhanced by using unified memory between the CPU and GPU and such analysis should be a part of our Advisor.
6. Support OpenMP Advisor for all OpenMP directives, not just target offloading.

These are just some of the directions where this research can be extended while using OpenMP. Although this research works well with OpenMP GPU offloading, it can also be extended to work with CPU-related directives or with other directive-based models, like OpenACC.
10.4 Tools and plugins development

This thesis provides a proof of concept for the use of a compiler to assist application scientists easily port their code to a GPU. However, the work is not yet complete. Compiler engineers can use the methods outlined in this thesis to create standalone tools or extensions for integrated development environments (IDEs) that make it simple for application scientists to generate any code for offloading to a GPU with the click of a button. This research could be applied to the backend of an IDE to build a drag-and-drop environment that would make it easier for application scientists to develop their applications, in addition to GPU offloading. This thesis has endless applications and engineering extensions.
Chapter 11

Conclusion

“In literature and in life we ultimately pursue, not conclusions, but beginnings.”

– Sam Tanenhaus in Literature Unbound [138]

Sam Tanenhaus’ statement about literature and life also applies to Computer Science. No work is ever concluded; rather, it is always the start of another work. The same is true of the work that this thesis defines. The work described in this thesis serves as a foundation for future GPU offloading using OpenMP and machine learning.

The HPC sector has begun to explore beyond Moore’s law, and hardware designers have enhanced chip performance by configuring an increasing number of computation cores. The most recent TOP500 list amply depicts the trend toward heterogeneous HPC platforms, notably those using GPUs, notwithstanding some notable exceptions. With the AMD or NVIDIA GPUs often offering great performance per watt, a sizable fraction of the systems are heterogeneous. Despite the success of the GPU architecture over the last decade, GPU programming remains difficult. It is essential to update programming languages and modify application code to take advantage of the cores due to the hardware’s quick change in architecture, such as by adding pthreads or OpenMP structures to the source code. Numerous high-level directive-based programming models have arisen in recent years, and they make GPU programming easier while retaining great performance. Even so, it is an enormous work for an application scientist to comprehend the physics, convert it into a computer program, analyze the offloadable kernel, and then think about how to parallelize it to run well on an HPC cluster.
This thesis introduces **OpenMP Advisor**, a first-of-its-kind compiler tool that advises application scientists on various OpenMP code offloading strategies. In Chapters 1 and 2, it provides an overview and background information on HPC, ECP, and supercomputers. It also discusses how important GPUs are to the HPC industry. The development of benchmark applications to support Unified Virtual Memory in OpenMP marked the beginning of this thesis’ journey, as discussed in Chapter 3. This chapter demonstrates how challenging it is for application scientists to offload their code to a GPU. It turns out that GPU offloading is much more difficult than it appears, even when using OpenMP. The framework proposed in Chapter 5 is intended to assist application scientists in offloading their novelty code to accelerators such as GPUs. The challenges involved in managing data between a host and device effectively were thoroughly analyzed, as described in Chapter 6. Then, using artificial neural network, COMPOFF, a novel cost model that statically estimates the Cost of OpenMP OFFloading, is discussed in Chapter 7. The validity of this work is demonstrated in Chapter 8, which discusses the thesis’ final experiments and results analysis. The thesis also discusses our implementation of OpenMP metadirective in Chapter 9, which can be used for future extension of this work, and then goes on to discuss what the future holds for this work in Chapter 10.

The resulting framework evolved into the OpenMP Advisor compiler tool, which is the focus of this thesis. It demonstrates how easily the Advisor may be adapted to different applications, compilers, and GPUs. The variants generated for various applications were able to build using the LLVM/clang, Intel/icpx, GNU/gcc, and NVIDIA HPC/nvc compilers for a number of HPC clusters with various GPUs. Successful experiments were conducted using clusters like Summit (NVIDIA Tesla V100-SXM2-16GB), Ookami (NVIDIA Tesla V100-PCIe-32GB), Intel Devcloud (Intel HD Graphics P630), and Sea-wulf (NVIDIA Tesla K80-PCIe-12GB). Using our novel machine learning-based compile time cost model COMPOFF, the tool is also capable of predicting the runtime of each generated variant for offloading. The tool successfully used a proxy application based on matrix-vector multiplication that had computations similar to the Wilson Dslash operator and used the resulting data to train the model rather than using data from the Wilson Dslash operator to train the ML model. This proves the adaptability of our framework. When attempting to predict for a new application for which real-time data collection is difficult or impractical, the application scientist could create a proxy application similar to the target application and train the model using
data collected on the proxy application.

This framework is a stepping stone to future ease of use of GPU offloading using OpenMP and Machine Learning. Although the tool is currently limited to GPUs, it is extensible to other OpenMP-capable devices. Machine learning has a big impact on compiler optimization now and will continue to do so in the future. This thesis already establishes the viability of a machine learning-based methodology for compiler development. More work needs to be done in terms of gathering more diverse data in order to aid any future machine learning model for compilers. The modular approach taken in the development of the Advisor ensures that, with minimal code changes, future developers could easily extend it to use the various loop transformations offered by OpenMP, such as unroll and tile, as well as other OpenMP concepts like tasking, unified memory, and so on, to have more control over the user code. It also creates opportunities for IDE developers to incorporate our different modules into their IDE’s plugins to better assist application scientists in the development of their application, who could generate code to offload their kernel to a GPU with the click of a button. To create a smarter and better cost model for predicting a kernel’s runtime, more research on various ML models can be conducted to take this work forward. The Advisor could be extended in the future to generate code for multiple GPUs and to guide the code for data affinity.
Bibliography


[18] C++ heterogeneous-compute interface for portability, 2016. URL: https://github.com/ROCm-Developer-Tools/HIP.


[52] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In


[124] David Schneider. The exascale era is upon us: The frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spectrum*, 59(1):34–35, 2022.


[139] Top500 supercomputer sites, 2021. URL: https://www.top500.org/lists/top500/2022/06/.


[153] Yang Xiao, Thireshan Jeyakumaran, Ehsan Atoofian, and Ali Jan-nesari. Improving performance of transactional memory through ma-
chine learning. *Concurrency and Computation: Practice and Experi-

[154] Yonghong Yan, Anjia Wang, Chunhua Liao, Thomas RW Scogland, 
and Bronis R de Supinski. Extending openmp metadirective semantics 
for runtime adaptation. In *International Workshop on OpenMP*, pages 


[156] Yao Zhang and John D Owens. A quantitative performance analysis 
model for gpu architectures. In *2011 IEEE 17th international sympo-
sium on high performance computer architecture*, pages 382–393. IEEE, 
2011.