

**EXPLORING HIERARCHICAL PARALLELISM IN DIRECTIVE-BASED  
MODELS FOR EFFICIENT GPU EXECUTION**

by  
Eric Wright

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Fall 2023

© 2023 Eric Wright  
All Rights Reserved

**EXPLORING HIERARCHICAL PARALLELISM IN DIRECTIVE-BASED  
MODELS FOR EFFICIENT GPU EXECUTION**

by

Eric Wright

Approved: \_\_\_\_\_

Weisong Shi, Ph.D.

Chair of the Department of Computer and Information Sciences

Approved: \_\_\_\_\_

Levi T. Thompson, Ph.D.

Dean of the College of Engineering

Approved: \_\_\_\_\_

Louis F. Rossi, Ph.D.

Vice Provost for Graduate and Professional Education and

Dean of the Graduate College

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
Sunita Chandrasekaran, Ph.D  
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
James Clause, Ph.D  
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
Leila Barmaki, Ph.D  
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
Matthias Rempel, Ph.D  
Member of dissertation committee

## ACKNOWLEDGEMENTS

The LLVM-SIMD project 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. The research is also supported by the NSF under grant no. 1814609. The views and opinions of the author do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein. This work was in parts prepared by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-851052).

The MURaM project was in part supported by MPS and NCAR; sponsored by the NSF under Cooperative Agreement No. 1852977. We would like to also acknowledge NSF grant No. 1814609. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program, grant agreement No. 695075. We acknowledge support from NASA’s SDO/AIA (NNG04EA00C) contract. We acknowledge Derecho supercomputer at NCAR and Raven supercomputer system at (MPCDF). We also acknowledge the OpenACC organization for their technical support.

## TABLE OF CONTENTS

<b>LIST OF TABLES</b> . . . . .	<b>viii</b>
<b>LIST OF FIGURES</b> . . . . .	<b>ix</b>
<b>ABSTRACT</b> . . . . .	<b>xiv</b>
 <b>Chapter</b>	
<b>1 INTRODUCTION, BACKGROUND AND MOTIVATION</b> . . . . .	<b>1</b>
1.1 GPUs in High Performance Computing (HPC) . . . . .	2
1.2 Parallel Programming Models and Methodology . . . . .	3
1.3 Modern Supercomputing . . . . .	4
1.4 OpenMP offloading and LLVM/Clang . . . . .	5
1.5 OpenMP's simd Directive . . . . .	7
 <b>2 EXPERIENCES APPLYING DIRECTIVES TO LARGE APPLICATIONS</b> . . . . .	 <b>12</b>
2.1 Introduction and Motivation . . . . .	12
2.2 Performance Profiling . . . . .	19
2.3 General approach with OpenACC . . . . .	23
2.4 Optimizing Radiation Transfer . . . . .	28
2.5 Deconstructing the OpenACC runtime for performance . . . . .	34
2.6 Scaling Results . . . . .	35
2.6.1 Experimental Setup . . . . .	36
2.6.2 Single GPU Performance . . . . .	37
2.6.3 Strong Scaling . . . . .	37
2.6.4 Weak Scaling . . . . .	38
2.6.5 Results Summary . . . . .	39
2.7 Summary and Take-aways . . . . .	41

<b>3</b>	<b>LIMITATIONS OF DIRECTIVE-BASED MODELS FOR REAL-WORLD APPLICATIONS . . . . .</b>	<b>42</b>
3.1	Introduction . . . . .	42
3.2	Methodology . . . . .	44
3.2.1	High-performance FFT Libraries . . . . .	44
3.2.2	Debugging and Test Cases with PCAST . . . . .	45
3.2.3	Optimizing Radiation Transport . . . . .	47
3.2.4	MURaM I/O . . . . .	48
3.2.5	Singularity Containers . . . . .	48
3.2.6	Programming Struggles . . . . .	49
3.3	Exploring Performance Vs Portability . . . . .	50
3.3.1	Limitations of OpenACC . . . . .	50
3.3.2	Integration with Grid-Level Synchronization . . . . .	52
3.3.3	Integration with CUDA Graphs . . . . .	53
3.3.4	Integration Performance . . . . .	55
3.4	Scaling Results . . . . .	58
3.4.1	Experimental Setup: Raven and Derecho . . . . .	59
3.4.2	Weak Scaling . . . . .	59
3.4.3	Strong Scaling . . . . .	60
3.5	Analysis and Future Work . . . . .	63
<b>4</b>	<b>AUTOMATIC PARALLELIZATION OF SERIAL CODES WITH OPENACC ANNOTATIONS . . . . .</b>	<b>65</b>
4.1	Related work . . . . .	65
4.2	Intermediate Representation with Clang’s Abstract Syntax Tree . . . . .	69
4.3	Implementation of Automatic Annotation . . . . .	70
4.4	Performance and Code Generation Results . . . . .	81
<b>5</b>	<b>THREE-LEVELED HIERARCHICAL GPU PARALLELISM IN THE OPENMP PROGRAMMING MODEL . . . . .</b>	<b>88</b>
5.1	Introduction . . . . .	88

5.2	Parallel Loop Code Generation in LLVM/Clang . . . . .	91
5.2.1	Clang AST and Code Generation . . . . .	91
5.2.2	LLVM's OpenMP IR Builder . . . . .	92
5.2.3	Worksharing Loops in LLVM/OpenMP's GPU Runtime . . . . .	93
5.3	SIMD in LLVM/OpenMP's GPU Runtime . . . . .	94
5.3.1	OpenMP/GPU Hardware Mapping . . . . .	94
5.3.2	Overview of General Runtime Changes . . . . .	97
5.3.3	CPU-Centric SIMD-Generic Mode . . . . .	98
5.3.4	Variable Sharing . . . . .	102
5.3.5	GPU-Centric SIMD-SPMD Mode . . . . .	103
5.3.6	Towards AMD GPU Support . . . . .	105
5.4	LLVM/OpenMP SIMD Results . . . . .	105
5.5	Experimental Improvements to LLVM/OpenMP SIMD . . . . .	109
5.5.1	Updated Code Generation for LLVM 17 . . . . .	109
5.5.2	Loop Reductions . . . . .	110
5.5.3	Automatic conversion of SIMD-Generic code . . . . .	110
5.5.4	Towards AMD Support . . . . .	114
<b>6</b>	<b>CONCLUSION AND SUMMARY . . . . .</b>	<b>116</b>
	<b>BIBLIOGRAPHY . . . . .</b>	<b>119</b>

## LIST OF TABLES

4.1	Performance analysis of the OpenACC collapse clause . . .	81
4.2	Results of AutoACC . . . . .	86

## LIST OF FIGURES

1.1	Simplified mapping of the OpenMP programming model to the GPU. Top row: Outermost parallelism across streaming multiprocessors (SMs) onto which OpenMP teams are mapped. All threads on this level share the global memory. Middle row: A single SM corresponding to a team of threads (team main + parallel workers) and, through this work, also SIMD vector lanes (simd workers) in OpenMP. Both shared and global memory are accessible by these threads. Bottom row: A single GPU/OpenMP thread which has exclusive access to local memory and registers. . . . .	10
1.2	An OpenMP offloading code with branching execution paths. . . . .	11
1.3	A possible implementation of the above OpenMP code in the kernel language CUDA. . . . .	11
2.1	MURaM single-core profile results generated with the ARM MAP profiler. 192x128x128 dataset simulation on 1 core of an Intel Xeon E5-2697V4 (Broadwell) CPU. . . . .	21
2.2	MURaM multi-core profile results generated with the ARM MAP profiler. 192x128x128 dataset simulation on 36 cores (full node dual socket) of an Intel Xeon E5-2697V4 (Broadwell) CPU. . . . .	22
2.3	Theoretical and achieved GPU occupancy of various GPU kernels within MURaM. . . . .	24
2.4	Effect of GPU register usage on GPU occupancy in the MHDRES kernel. . . . .	25
2.5	Maximum Relative Error in GPU calculated Q_Radiation over 50 iterations. Red: max threshold and Blue: what was observed . . . .	26
2.6	A simplified code example of the core computational loop in RTS. . . . .	30

2.7	Representation of the wavefront dependency in RTS integration. a) shows the preloaded boundary conditions with no computation done. b) shows the first set of grid points in red to be computed in parallel. c) shows the next set of grid points to be computed in parallel, each blue grid point depends on up to four of the red grid points. d) shows the last grid points to be computed in parallel, each green grid point depends on up to four of the blue grid points. . . . .	31
2.8	Nvprof profiler highlights the process of reducing kernel launch overhead in the RTS <i>integrate()</i> kernel. (a) Overhead between 2 launches, (b) Reduced overhead between kernels launches, and (c) Further reduced overhead . . . . .	36
2.9	CPU-GPU strong scaling of a multi-band $288^3$ dataset . . . . .	38
2.10	GPU Strong Scaling of a $288 \times 576 \times 576$ dataset . . . . .	39
2.11	GPU weak scaling of a $288^3$ dataset for RTS & other MURaM routines. Note that FFTW is a CPU only library . . . . .	40
3.1	Basic loop structure of the Integrate RT kernel, assuming the dependency is in the Y dimension and moving forward. . . . .	51
3.2	A possible implementation for thread block synchronization in OpenACC. . . . .	52
3.3	Integrate kernel with grid-level synchronization using CUDA Cooperative Groups. . . . .	53
3.4	Integrate using CUDA Graphs to manage launching of individual 2D slice kernels. . . . .	54
3.5	Relative speedup of the different variations of <i>Integrate</i> over MURaM's OpenACC baseline implementation. . . . .	55
3.6	Breakdown of time spent in computation and overhead for the different <i>Integrate</i> variations, captured using the NSight Systems profiler. "Time per slice" in the computation time of a single 2D parallel slice of <i>Integrate</i> , "Overhead" is the time taken from the end of one kernel to the start of the next, and "Total" is the time taken for the entire integration, i.e over the entire 3D domain. All times are using the $128^3$ dataset. . . . .	55

3.7	Weak scaling on Raven of a high-resolution simulation ranging from 4 to 500 A100 GPUs. Dotted lines represent ideal scaling. . . . .	60
3.8	GPU strong scaling for a high-resolution simulation, 4-band RT, from 4 to 512 A100 GPUs on the Raven system. (a) shows total time per timestep, (b) show time of RT per timestep, (c) shows total time - RT per timestep. . . . .	61
3.9	GPU strong scaling for a lower-resolution, larger-scale simulation from (a) 48 to 320 A100 GPUs and (b) 4,096 to 65,536 AMD CPU cores on the Derecho system. . . . .	62
4.1	Visual representation of an AST for an OpenACC parallelized for loop.	70
4.2	Looping development cycle of incrementally parallelizing sequential codes with directives. . . . .	73
4.3	Workflow of the AutoACC tool from input sequential code to automatically generated parallel code. . . . .	74
4.4	Matvec kernel annotated with AutoACC. . . . .	82
4.5	Matvec automatic conversion to OpenACC. . . . .	83
4.6	Laplace2D . . . . .	84
4.7	Laplace2D Tuned . . . . .	85
5.1	Sparse matrix vector multiplication kernel with OpenMP offloading. The code highlights an small, variably sized inner-loop that benefits from the intermediary warp-level parallelism. . . . .	90
5.2	Function for executing <code>simd</code> loops. Each thread will execute a portion of the total iterations. . . . .	94
5.3	LLVM optimization for efficiently calling outlined functions. . . . .	95
5.4	A possible mapping of a single OpenMP team on an NVIDIA GPU using four warps, totaling 128 threads for the computation. One SIMD group per warp, meaning one SIMD main and 31 SIMD workers per warp. One additional warp is included to act as the main thread in the team, which is required when the teams region is executing in generic mode. . . . .	96

5.5	A portion of the <code>__parallel</code> runtime function showing the two different execution modes that parallel regions can be. If the <code>parallel</code> region is SPMD mode then all threads within the SIMD group will execute it. If it is instead generic mode only the SIMD main thread will execute the parallel region while all SIMD workers enter into a separate SIMD state machine. . . . .	98
5.6	Runtime function for OpenMP <code>simd</code> loops in SPMD or generic mode. If the <code>parallel</code> region is generic mode then all variables needed within the <code>simd</code> loop must be shared from the SIMD main thread, and the SIMD workers must be notified of what loop should be executed and for how many iterations. If the <code>parallel</code> region is instead in SPMD mode then this information is already local to each thread and no communication needs to occur. . . . .	99
5.7	Program flow diagram of the GPU runtime assuming all regions are executed in generic mode. . . . .	100
5.8	New state machine for SIMD workers when the containing <code>parallel</code> region executes in generic mode. Workers immediately reach a warp-level barrier. Once the SIMD main thread completes the barrier all workers will fetch the function pointer of the current <code>simd</code> loop, as well as any variables shared from the SIMD main thread. If the function pointer is a null pointer then the worker threads exit the state machine, as this signifies the end of the current <code>parallel</code> region. . . . .	101
5.9	Flow diagram for SIMD worker threads upon encountering a parallel region. If the region should be executed in SPMD mode, worker threads will execute the entire region under the assumption that no side-effects will be produced. If the region should instead be executed in generic mode, worker threads will enter into the state machine and wait for a <code>simd</code> loop to be encountered. . . . .	104
5.10	Results for various kernels comparing our <code>simd</code> implementation to the original two levels of parallelism. Experiments with all possible SIMD group sizes. . . . .	106
5.11	Relative speedup of the different combinations of execution modes. The modes are all combinations of <code>teams</code> Generic or SPMD / <code>parallel</code> Generic or SPMD. The performance similarity for SPMD/SPMD and “No SIMD” suggests low performance overhead in our SPMD implementation. . . . .	107

5.12	3-dimensional heat diffusion OpenMP offloaded code. . . . .	108
5.13	An example of a summation reduction. . . . .	110
5.14	An example of how a reduction could be implemented. . . . .	111
5.15	Tightly-nested parallel regions inherently allow for SPMD execution.	111
5.16	Extra code in parallel region causes side-effects and prevents SPMD mode. . . . .	112
5.17	Thread guarding and variable broadcasting to allow the code to execute in SPMD mode. . . . .	113
5.18	An alternative version where instead of broadcasting variables a shared allocation is created and all threads will reference the same shared allocation. . . . .	114
5.19	Another alternative where a new runtime function that allows all threads to utilize the same shared memory stack. This reduces the need completely for additional synchronization. . . . .	115

## ABSTRACT

The advent of general-purpose GPUs in parallel computing brings several new languages, tools and programming models. One popular way to program GPUs is using high-level directives in common languages such as C and Fortran that provides an easy to understand and familiar environment to the programmer. However, unlike dedicated GPU languages, these directive-based models often struggle to express low-level features of the GPU hardware as well as introduce performance overheads in the form of complex runtime libraries. Two such model are OpenMP, which has been a staple for parallel CPU programming for several decades and has supported GPUs as of version 4.0, and OpenACC, released in 2012 was designed by many developers from the OpenMP community to bring a standard that supports code offloading from the beginning.

This work introduces two impactful projects within these programming models. Firstly, the MURaM project is developed by an interdisciplinary team of domain scientists and HPC research software engineers who seek to examine the strengths and limitations of these directive-based models, as well as the difficulties faced by application developers when bringing large code bases to advanced parallel hardware. Secondly, the LLVM/OpenMP SIMD project aims to fill an important implementation hole within the LLVM compiler to fully utilize the parallelism available to GPUs. OpenMP, like other similar models, allows for three distinct levels of hierarchical thread parallelism which matches the GPUs hardware layout. However, due to the implementation complexity, OpenMP compilers often omit user control of the middle level of parallelism. This greatly limits the achievable performance for codes that do utilize all three explicit layers and often requires restructuring of these applications as a workaround. In this proposal we outline our design and prototype of this middle level of parallelism through OpenMP’s “simd” directive using the open-source compiler LLVM and its OpenMP

GPU runtime, which includes both a CPU-centric model for OpenMP conformability, and a GPU-centric optimized model for higher performance.

## Chapter 1

### INTRODUCTION, BACKGROUND AND MOTIVATION

High Performance Computing (HPC) centers around highly-parallel and scalable computational systems and software. Nearly all modern computational devices include multiple, parallel computational units; for example, CPUs using multiple cores. HPC has largely seen a shift away from homogeneous CPU-based systems and has introduced a hybrid host/device model, with a typical CPU executing general-purpose code, and highly-parallel, computationally-intensive code “offloaded” to a specialized co-processor. There have been many different types of co-processors that have been developed such as Intel’s Knights Landing, FPGAs, and GPUs.

With the increased complexity and scale of these systems there is an increased strain on application developers for utilizing these powerful parallel processors to their fullest. HPC is a constantly evolving ecosystem where software often struggles to keep up with new hardware developments. Additionally, since many architectures originate from competing vendors there is unfortunately low compatibility and reuse among different hardware and software. Developers wishing to use these systems often need guidance from software engineers specializing in such systems, which brings about interesting collaborations of interdisciplinary teams.

The work presented in this thesis focuses on improving the ways that application developers and scientists interact with HPC systems and discusses the difficult issue of hardware abstraction, where low-level hardware features are expressed in such a way as to be accessible and high-performing for non-specialized programmers. While this topic is broadly applicable to various programming models, compilers, tools and software, this work focuses specifically on abstract, high-level, directive-based parallel programming models and how they perform and can be improved for HPC developers.

Three key projects are discussed, which all relate to this topic:

- AutoACC: an automatic, source-to-source compiler for annotating serial code with directives to enable parallel execution.
- MURaM: a state-of-the-art solar physics application developed by an interdisciplinary team using the directive-based model OpenACC for GPU programming.
- LLVM/OpenMP SIMD: designing and implementing new functionality to the LLVM compiler to give programmers finer control over low-level hardware intrinsics with the OpenMP directive-based programming model.

### 1.1 GPUs in High Performance Computing (HPC)

GPUs were initially developed for rendering computer graphics. However, by reformatting computational problems in terms of computer graphics primitives the GPU can utilize graphics shaders and APIs such as OpenGL and DirectX to solve these problems with the many-core GPU hardware [70, 24]. General-purpose computing on graphics processing units (GPGPU) has become commonplace in HPC, and many compilers, APIs, and programming languages have allowed for GPUs to execute code that would have traditionally been used for CPUs.

These GPUs are developed by a handful of different vendors (namely NVIDIA, AMD and Intel) and each have their own programming ecosystem. Each of these architectures utilizes different programming languages and software stacks to achieve highly parallel programs. NVIDIA provides the CUDA [7] language for programming NVIDIA GPUs. AMD is developing a new language called HIP [5], which is highly reminiscent of CUDA, for AMD GPUs (such as those in Frontier). Intel is developing a new programming model called OneAPI [8] to program Intel CPUs, GPUs and other co-processors.

In general, these GPUs have very similar architectures, and the concepts and best-practices of one architecture are often applicable to all other GPUs. To highlight general GPU architecture, we use NVIDIA as an example: NVIDIA GPUs are composed

of multiple streaming multiprocessors (SM). Each SM will contain some number of warps, and a warp contains a single program counter, memory unit, and 32 computational cores, each with a floating point computational unit and an interger computational unit. A warp executes in lock-step and all cores execute the same instruction. Global memory reads take many more clock cycles than on a CPU, so when a global memory read occurs a warp-stall is usually encountered. While the warp is stalled, those threads can be made idle and the cores will be reused to execute the next set of threads until those threads also hit a stall. This is called latency-hiding.

## 1.2 Parallel Programming Models and Methodology

The three primary methods for creating GPU applications: 1) kernel languages, such as CUDA and HIP, provide low-level control of GPU execution. 2) GPU-enabled libraries. 3) directive-based models.

Kernel-languages limit portability but has the highest potential for performance.

GPU libraries offer high performance, but are limited to specific domains and cannot cover all possible codes.

Directive-based models, such as OpenMP and OpenACC, offer high portability and allows programming in a common language, but often limit access to hardware-specific features, and have some overhead associated with their implementation.

OpenACC uses a generic host/device execution model where general-purpose code is executed on the host (primarily a CPU), and compute-intensive parallel code is offloaded to the device. This device is typically a dedicated co-processor such as a GPU or FPGA, but utilizing the host as the device is also possible, usually representing that offloaded regions should be run on multiple cores of the CPU.

OpenMP is a directive-based parallel programming model traditionally used for programming multicore CPU architectures. At the rise of GPGPU programming new directive-based programming models emerged, the most popular being OpenACC. In 2013, OpenMP also started to support an accelerator offloading model with the OpenMP 4.0 standard to target GPGPU [1]. Directives, commonly referred to as

compiler hints, allow the maintenance of a single source code targeting both CPUs and accelerators.

OpenMP and OpenACC have several important advantages over using low-level languages such as CUDA or HIP. Directives create native C, C++, or Fortran applications that can be accelerated via offloading to any parallel device as long as compiler support exists. For example, a common use-case of OpenACC is to have a traditionally-serial C or Fortran code compiling for and running on a single core of a single CPU, manycores of a single CPU, or a heterogeneous CPU/GPU machine. Many modern applications are written using OpenMP 4.0+ or OpenACC, including a considerable number of the applications running on Summit.

Regardless of these advantages, directives are still not perfect. In general, porting a serial code from C, C++ or Fortran to GPU-based architectures using directives is a very tedious and error-prone process. Also, there exists several cases where native C, C++ or Fortran code is by nature not optimized for GPU execution, and may produce poor performance when using a naive OpenMP or OpenACC implementation compared against a hand-tuned GPU language-specific approach. A well-tuned code written in a native GPU language often performs better than using directives, but some work has shown that with modern compilers a well-tuned directive-based code is not far off in performance [58].

### 1.3 Modern Supercomputing

In recent years, High Performance Computing (HPC) has made a transition from homogeneous CPU-based to heterogeneous CPU and GPGPU (general-purpose graphical processing unit, or GPU) computing architectures. These GPUs are developed by a handful of different vendors (namely NVIDIA, AMD and Intel) and each have their own programming ecosystem. There have been several shifts in supercomputing technology recently and in the near future the estimated top three performing supercomputers will each use different GPU hardware from these different vendors. Currently, the Summit supercomputer [2] from the Oak Ridge National Lab (ORNL) is the fastest

supercomputer in the world and utilizes IBM CPUs and NVIDIA GPUs. The Frontier supercomputer [3], also from ORNL, is set to release in 2021 and will have AMD CPUs and AMD GPUs. The Aurora supercomputer from Argonne National Lab will have Intel CPUs and Intel GPUs.

Each of these architectures utilize different programming languages and software stacks to achieve highly parallel programs. NVIDIA provides the CUDA [7] language for programming NVIDIA GPUs. AMD is developing a new language called HIP [5], which is highly reminiscent of CUDA, for AMD GPUs (such as those in Frontier). Intel is developing a new programming model called OneAPI [8] to program Intel CPUs, GPUs and other co-processors. This means that in the near future HPC application developers using supercomputer technology will potentially learn and write their codes in either CUDA, HIP or OneAPI. These architectures are also likely to be seen in smaller compute clusters as many universities and organizations are switching to GPU computing.

Many state-of-the-art HPC applications are either locked into a specific hardware vendor or are still only implemented for CPU computing. Architecture portable solutions have been a highly researched topic since the rise GPGPU programming. Currently, some of the most popular solutions are OpenMP [1], OpenACC [4], Kokkos [52] and Alpaka [125]. Of these models, OpenMP and OpenACC are the most widely adapted by the HPC community.

#### **1.4 OpenMP offloading and LLVM/Clang**

LLVM is a compiler infrastructure that seeks to standardize compiler front-ends. LLVM uses an intermediate representation (LLVM IR) that is somewhere between a high-level programming language and a low-level assembly-like language. Using LLVM IR, LLVM supports various compiler back-ends. The goal being that any compiler able to build to LLVM IR then has access to every back-end architecture that LLVM supports, as well as all of the optimization tools implemented in LLVM.

Clang is a C/C++ compiler that was developed in tandem with LLVM. Clang builds C/C++ code to LLVM IR, then LLVM handles all code optimizations and provides the compiler back-end. IBM’s XL compiler is based on LLVM/Clang, often leading to features developed at IBM being also implemented into LLVM/Clang. Flang a Fortran compiler frontend targeting LLVM IR, similar to Clang.

OpenMP is a long popular directive-based programming model in HPC, originally for multi-threaded CPU programming. OpenMP has supported GPU offloading since the 4.0 standard with the inclusion of new `target` and `target data` directives, and has been extended and improved in subsequent OpenMP versions. User experiences of applying OpenMP target offloading can be found in some of the recent work including using the SPEChpc2021 benchmarking suite [28], other applications including HPGMG [39], miniMD [91], UK mini-apps [82], LULESH [68] among others. Work in [31, 92] discusses at length experiences gained and practices adopted from OpenMP hackathons when applying offloading features on HPC applications and mini-apps based on different computational motifs (BerkeleyGW, WDMApp/XGC, GAMESS, GESTS, and GridMini) targeting heterogeneous systems. Support for OpenMP offloading is implemented in some degree in the LLVM, GNU, Cray, IBM XL and Intel compilers.

LLVM/Clang is a fairly mature OpenMP offloading compiler providing back-ends for both NVIDIA and AMD GPUs.

The OpenMP offloading support for GPUs in LLVM can be traced back to the two works discussed in [23, 22]. The (PGI) Fortran front-end, known as Flang, supported OpenMP offloading via the LLVM OpenMP runtime [90]. Since then, researchers have been working on compiler and runtime optimization for LLVM OpenMP. The first front-end-based optimizations for NVIDIA GPUs that can avoid idle threads and reduce register usage was introduced in [16]. Work in [48] presented the TRegion interface which delays the discovery of SPMD regions to compiler middle end, contrary to the front-end based approach used before, which can support more kernels to execute in SPMD mode. Runtime support for concurrent execution of OpenMP target tasks was introduced in [115]. Results in [62] discusses OpenMP-aware program analyses and

optimizations that allow efficient execution of the generic, CPU-centric parallelism model provided by OpenMP on GPUs. A co-design methodology is presented in [49] for optimizing applications using a specifically crafted OpenMP GPU runtime inducing near-zero overhead in most cases.

OpenMP offloading utilizes a host-device execution model where the host (CPU) schedules and synchronizes target tasks, in the form of *kernels*, and handles memory allocation and movement between the host and target devices (e.g. GPUs). Computational kernels are executed on the device by launching a league of *teams*. Each team has a *main thread* that will begin executing the code region contained by the `teams` directive. Additional *worker threads* can be spawned by using the `parallel` directive. There are three worksharing constructs: `distribute`, `for` and `simd`. `distribute` schedules loop iterations across the league of teams, `for` schedules loop iterations across threads within a team and `simd` uses single instruction multiple data (SIMD) parallelism for the loop.

A simple mapping of the OpenMP model to GPU hardware is a team per each SM, and threads within the team to hardware threads within the SM.

The `simd` construct specifies that the attached loop should be executed using SIMD parallelism. For GPUs this is typically done using single instruction, multiple thread (SIMT) parallelism, meaning that multiple threads within a computational unit execute the same instruction. This means that in terms of GPU offloading a `simd` loop should be executed in parallel by a set of adjacent threads. We can achieve this by separating the threads in each team into distinct groups. These *simd groups* will contain a single main thread and multiple workers. The main thread executes `parallel` regions and all threads in the group execute `simd` loops.

## 1.5 OpenMP's `simd` Directive

The OpenMP programming model is a CPU-centric model that allows for sequential and parallel regions to be used interchangeably. This aligns well with a CPU execution where alternating between single-threaded execution and spawning additional threads is fairly easy. However, the GPU execution model generally requires all threads

to begin execution at the start of the kernel and stay active until termination. In a kernel-language sequential regions of code would require guarding to assure that only a single thread executes them, and any side-effects created during that region need to be communicated to other threads.

Prior work [23] discusses several problems in creating a portable solution to this in LLVM/Clang. In the case of parallel regions existing within branching if/then statements all threads will need to handle all potential paths that could be taken to ensure that all threads reach the correct parallel region. Handling the thread guarding needed for branching paths require extensive code generation, which is antithetical to Clang's design, and limits portability to other compilers.

An implementation that utilizes a state machine where threads alternate between idle, during sequential regions, and active, during parallel regions is described in [23]. A single thread is designated as the team main thread and will run the user code. All other threads are considered as worker threads and will enter the state machine where they will encounter a thread barrier and become idle until the main thread encounters a parallel region. This solves the problem of branching paths since only one thread needs to traverse the branching statements. When a parallel region is encountered some variables may need to be communicated to other threads if they are side-effects from the sequential code. The main thread will specify which parallel region was encountered before completing the thread barrier, signaling to the workers that they should become active and begin executing the parallel region. This implementation is what is currently used in LLVM/OpenMP.

Work in [116] introduces an execution mode in the IBM XL C/C++ compiler that avoids the generic state machine when all threads can execute in parallel. This new mode is referred to as single program multiple data (SPMD) mode and has since been upstreamed into the LLVM/Clang. The key characteristic of SPMD mode is the assertion that all threads can safely execute the target region and will encounter the same parallel regions and any sequential regions of code will not produce side-effects. The simplest case for when SPMD is applicable is when all affected OpenMP regions

are tightly nested, since this means there is no sequential code between the parallel regions. This allows OpenMP to behave similarly to GPU kernel-languages where all threads are active at the beginning of the kernel.

SPMD mode is further extended in [62] to be applicable to a larger variety of codes by introducing thread guarding of sequential code regions. The work introduces an inter-procedural analysis at the LLVM IR level to check sequential regions for potential side-effects that can be eliminated using thread guarding. If these guarded regions create values outside of the region the values would be broadcasted to other threads. The *SPMDization* of these codes avoid the use of the generic state machine at the cost of additional synchronization in the guarded regions and data broadcasting.

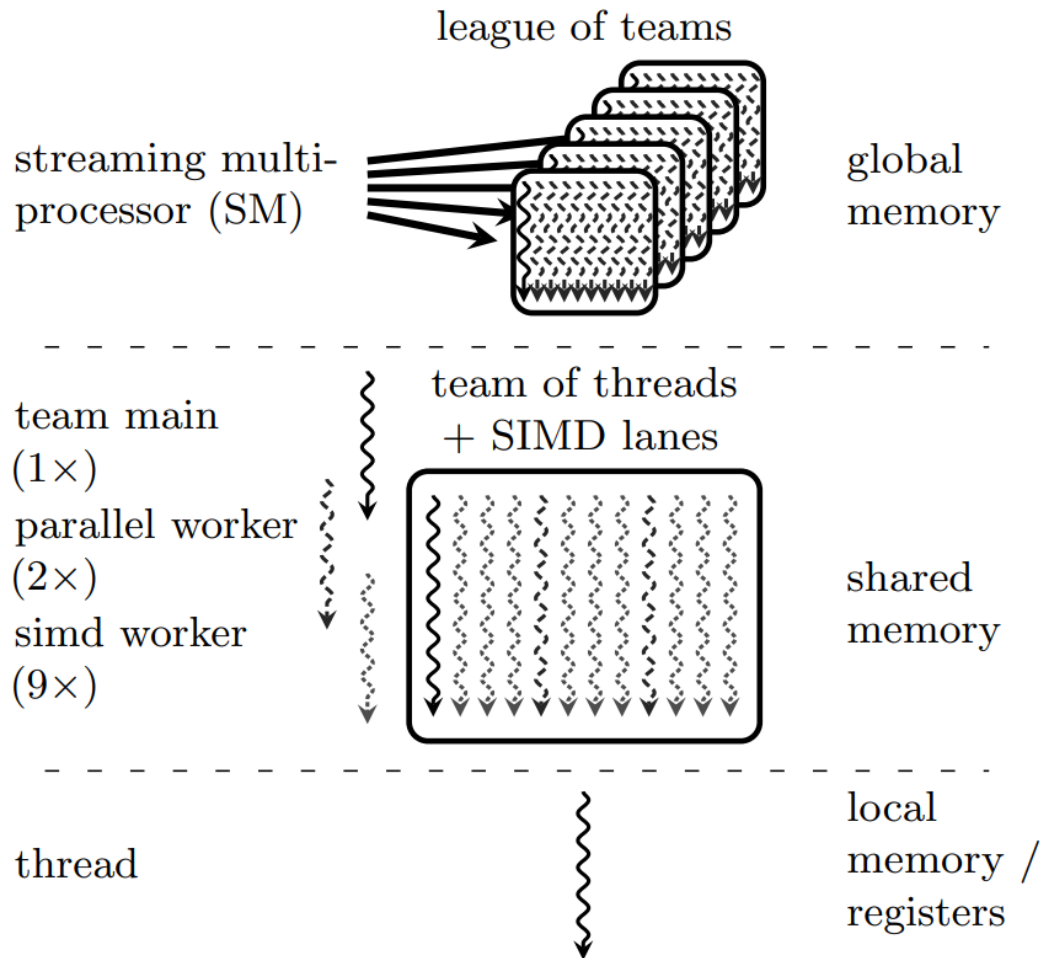


Figure 1.1: Simplified mapping of the OpenMP programming model to the GPU. Top row: Outermost parallelism across streaming multiprocessors (SMs) onto which OpenMP teams are mapped. All threads on this level share the global memory. Middle row: A single SM corresponding to a team of threads (team main + parallel workers) and, through this work, also SIMD vector lanes (simd workers) in OpenMP. Both shared and global memory are accessible by these threads. Bottom row: A single GPU/OpenMP thread which has exclusive access to local memory and registers.

```

#pragma omp target teams
{
    int Val;
    some_func(&Val);
    if(Val) {
        #pragma omp parallel
        <Parallel Region 1>
    } else {
        #pragma omp parallel
        <Parallel Region 2>
    }
}

```

Figure 1.2: An OpenMP offloading code with branching execution paths.

```

__global__ void omp_offloaded_region() {
    __shared__ bool Branch1;
    int Val;
    if(threadIdx.x == 0) {
        some_func(&Val);
        Branch1 = Val;
    }
    __syncthreads();

    if(Branch1) {
        <Parallel Region 1>
    } else {
        <Parallel Region 2>
    }
}

```

Figure 1.3: A possible implementation of the above OpenMP code in the kernel language CUDA.

## Chapter 2

### EXPERIENCES APPLYING DIRECTIVES TO LARGE APPLICATIONS

Large parts of the text of this chapter has been published in the Platform for Advanced Scientific Computing (PASC 2021), ACM <https://dl.acm.org/doi/10.1145/3468267.3470576>.

This chapter will discuss our experiences porting a real-world solar physics simulation application to GPU-based HPC systems using the OpenACC directive-based programming model, observing the difficulties that application developers face when interfacing with these highly-parallel compute resources and creating a general “best practices” for porting large applications to GPUs.

#### 2.1 Introduction and Motivation

Originally based on a magnetohydrodynamics (MHD) module from the University of Chicago, MURaM (Max Planck University of Chicago Radiative MHD) is jointly developed and used by the National Center for Atmospheric Research (NCAR), the Max Planck Institute for Solar System Research (MPS) and the Lockheed Martin Solar and Astrophysics Laboratory (LMSAL). The MURaM code [121, 95, 97] is a state-of-the-art solar model for simulations of the upper convection zone, photosphere (visible surface of the sun) and corona. MURaM simulations have contributed substantially to our understanding of solar phenomena ranging from the origins of quiet sun magnetism [120, 95, 98], the structure and evolution of sunspots and active regions [99, 37, 96, 33], to solar flares and the initiation of coronal mass ejections [97, 36]. MURaM also plays a key role in interpreting high resolution solar observations, e.g. [41, 40, 44, 104].

The collaboration of domain and computer sciences can greatly benefit scientific applications, accelerating ground-breaking research using state-of-the-art parallel programming techniques, tools and hardware. Such a strong synergy between both sciences not only opens up more scientific discoveries but also opens up newer questions in the field of computer science pushing computer engineers to advance computing technologies and their software. The work presented in this chapter is strongly motivated by the need for exploring the scientific capabilities of a solar physics application. This is being possible due to the convergence of domain and computer science expertise in the team. We, as a team of solar physicists, computer scientists along with a Research Software Engineer (RSE) explore the acceleration and scalability of the code on two large scale heterogeneous computing systems equipped with state-of-the-art GPUs housed in NCAR and MPS.

The code is made publicly available via GitHub [12]. Raw data used for the plots are available via Zenodo [13].

This work focuses on the GPU acceleration of the MURaM (Max Planck University of Chicago Radiative MHD) code [121, 95, 97], a widely used state-of-the-art solar model for simulations of the atmosphere, encompassing the upper convection zone, photosphere (visible surface of the sun) and corona. The code couples magneto hydrodynamics (MHD) with radiation transport (RT) and the applications of this code range from quiet sun magnetism [120, 95, 98], to the structure and evolution of sunspots and active regions [99, 37, 96, 33], the solar corona [97, 27, 34] and to solar flares and the initiation of coronal mass ejections [36, 35, 100].

With the construction of the NSF funded Daniel K. Inouye Solar Telescope (DKIST), resolution of ground based observational solar physics is poised to take an order of magnitude leap forward. The solar chromosphere, lying between the photosphere and the transition to the corona, is one of the least understood parts of the Sun. New high resolution observations from DKIST will allow us to observe the chromosphere in greater detail than ever before. Modeling the solar chromosphere is challenging

since radiation and the ionisation state and level populations of atoms are no longer determined by the temperature alone and are coupled to each other.

In the problem under study, MURaM is capable of simulating the coupled solar atmosphere from the upper convection zone into the lower solar corona, covering a density stratification of more than 25 scale heights. This makes MURaM a crucial tool for studying how magnetic field emerging from the solar interior is energizing the solar atmosphere and is leading to rapid release of energy in form of flares and coronal mass ejections. Currently these simulations are “stand-alone” setups that are inspired by solar events, but do not aim at modeling observed solar events in detail. Using these simulations in the future as a utility to study the solar drivers of space weather events will require data ingestion through boundary driving or full data assimilation and the ability to run such simulations in real-time. Starting from the current CPU baseline of the code, an increase of computational capabilities by 1-2 orders of magnitude is needed.

Previous production runs use up to 20,000 CPU cores, however in it’s current status it is not feasible for MURaM to reach the 100,000 to 1 million core range. Instead, these ambitious science goals require the use of exascale technologies.

The project to refactor MURaM for GPUs has multiple goals. First and foremost, we sought to improve the throughput of the model, as measured by site updates per second, or simulated seconds per wall clock second to the maximum extent possible. The project’s target throughput speedup was parity between one NVIDIA V100 GPU and 100 Intel Xeon v4 processor cores [105]. Experience refactoring similar models suggested that this speed-up was achievable for well-implemented code with sufficient data parallelism to saturate the GPUs. Our secondary goal was maintainability, by preserving portability of the model between CPUs and GPUs to the maximum extent possible. Finally, we sought, as a tertiary goal, performance-portability; namely, finding a maintainable implementation that did not compromise (significantly) model performance on CPUs or GPUs. At the time this project began in early 2018, the directive-based parallelization approach appeared to offer the best prospect of meeting

the first two goals, while performance portability was an open question. Of the directive-based systems, OpenACC [64] was deemed to be the most production ready in 2018.

The key contributions of this work are as follows:

- Specific GPU algorithmic enhancements, namely: asynchronous programming, loop fusion, and array replication, to the method of discrete ordinates used in MURaM’s radiation transport model. Radiation transport is the single most expensive part of most astrophysical codes, so these optimizations are broadly applicable across astrophysics.
- Acceleration of the radiation transport in MURaM will in turn make routine use of multi-band radiation transport possible in solar physics models. Multi-band radiation transport will advance understanding of the solar chromosphere when combined with non-equilibrium treatment of atomic populations.
- Demonstration of the efficacy of the use of OpenACC directive-based approach to achieve performance-portability across CPUs and GPUs in a solar physics model.
- Limitations and difficulties of producing low-level and highly-efficient GPU code using OpenACC.

The primary focus of the MURaM code is the detailed studies of the solar atmosphere with sufficient realism to allow for forward modeling of synthetic observables from visible to EUV and X-ray emission for direct comparison with a wide range of solar observations. To this end the MURaM code is typically applied to small Cartesian local regions on the Sun with lateral and vertical simulation domains ranging from a few 1,000 km to more than 100,000 km, with typical numerical grid spacings in a range of 2 – 200 km (for comparison, the solar radius is about 700,000 km). The code combines a fourth order conservative MHD scheme with short characteristics radiation transport as described in [121]. The numerical scheme for MURaM’s MHD model uses a cell centered finite difference approach, the  $\nabla \cdot \vec{B} = 0$  constraint is enforced through hyperbolic divergence cleaning [43]. The radiation transport resolves the angular

dependence of the radiation field by computing rays in (typically) 24 directions based on a Carlson quadrature [30]. Radiation transport is an intrinsically non-local problem, which poses implementation challenges on distributed memory architectures. The short characteristics solver of MURaM treats radiation transport locally, by using intensities from a previous time step (or iteration) as the starting point for the ray integration on each shared memory block. Achieving global convergence requires typically 3 to 4 iterations in the radiation transport solver. The above formulation has been extensively used to study magneto-convection in the solar photosphere and upper convection zone e.g., [121, 120, 99, 37, 79, 95, 33]. The code has been expanded to also include the overlying solar corona [97, 36].

This requires consideration of optically thin radiative losses that are computed from a tabulated lookup function based on the CHIANTI package [74] and efficient magnetic field aligned heat conduction [110]. Simulations that cover the full vertical extent from upper convection zone into the lower solar corona have typically a density contrast of 12 to 14 orders of magnitude and consequently suffer from a large separation in length and time-scales. This results in severe Courant–Friedrichs–Lewy (CFL) [42] timestep constraints that arise in particular from the Alfvén velocity and heat conduction in the corona. In order to cope with these challenges, the peak Alfvén velocity is limited through the use of the Boris correction [26, 57], which is essentially semi-relativistic MHD with an artificially reduced speed of light. Severe timestep constraints from effective heat conduction in the corona are alleviated through implementation of a hyperbolic heat conduction problem, i.e. a damped wave equation for the temperature [56, 107]. Unlike the classic parabolic heat conduction that leads formally to infinite signal propagation speeds, the hyperbolic approach has a well defined cutoff for conductive heat transport. Since the assumptions that lead to classic Spitzer conductivity [110] break down in the hot corona, where a free streaming limit for electrons is more physical [54], artificially limiting transport velocities to a value of a few times the ion speed of sound is not only avoiding a severe timestep constraint, but is also physically more accurate.

These implementations lead to a fully explicit code that can be parallelized for shared memory systems through domain decomposition and MPI communication. The major computational routines of the MURaM code, their algorithms and purpose, are summarised below:

- **MHDRES** - Calculate the right hand side of the MHD equations in conservative form. Calculate the derivatives of the fluxes using a fourth order central difference scheme.
- **TVDDIFF** - Numerical diffusion required to stabilise the solution. Calculated using a slope limited diffusion scheme, which is adapted from a total variation diminishing (TVD) scheme as described in [95].
- **EOS** - Using the density and energy from the MHD solution calculate the Equation of State (EoS) variables; temperature, pressure, electron number, entropy.
- **RTS** - Calculate the radiation field using the thermodynamic variables of the current MHD snapshot. A short-characteristics algorithm is used [72]. Consists of 4 main functions:
  - Interpolate - Perform trilinear interpolation of the MHD variables from the MHD grid to the staggered RTS grid. Calculate the radiation source function and opacities to be used in the integration routines.
  - Integrate - Integrate along each ray of the quadrature.
  - Exchange - Communicate intensity information to downstream processors in the ray direction.
  - Flux - From the specific intensity calculate the average intensity and radiative fluxes.
- **DIVBCLEAN** - Diffuse and disperse numerical  $\vec{\nabla} \cdot \vec{B}$  errors through a hyperbolic  $\vec{\nabla} \cdot \vec{B}$  cleaning approach.

- **INTEGRATE** - Calculate the updated variables for the next stage of the Runge Kutta algorithm using the divergence of fluxes from **MHDRES** and additional source terms such as gravity and radiative heating/cooling.
- **DST** - Exchange subdomain ghost cells with neighbours.
- **SYNC** - Determine the maximum time-step and synchronise the timestep between subdomains.
- **VLM** - Dynamically adjust the velocity, energy and Alfvén velocity limits to prevent extreme cells causing overly restrictive timestep constraints.
- **BOUNDARY** - Implement the vertical boundary conditions; A stratified open boundary at the bottom with passive field advection. Upper boundary is open to outflows, the magnetic field is matched to a potential magnetic field. The latter requires fast Fourier transforms.

Accelerating solar physics simulation models through GPU devices is a relatively new direction which attracts broad interest from the scientific community. Related work [29] shows the “Time-to-solution” performance results of a flux rope eruption simulation with their OpenACC implementation of the Magnetohydrodynamic Algorithm outside a Sphere (MAS) code, an in-production MHD code, which is part of the CORHEL suite hosted at the Community Coordinated Modeling Center (CCMC). The same group of researchers also summarized the implementation of OpenACC into MAS, including specific code example, strategies and development tips. Recently [118] investigated the interaction of large- and small-scale dynamos in a GPU accelerated MHD simulation. Their approach used CUDA and scaling was limited to a single node with 4 GPUs. Other non-GPU MHD codes include BATS-R-US [93] that solves 3D MHD equations using a finite volume numerical method. The code uses MPI and the Fortran90 standard. Similar PLUTO [85], a numerical code for computational astrophysics is a multi-physics, multi-algorithm, high resolution framework that includes MHD as one of the four independent physics modules of PLUTO.

As presented in this thesis, one of the major kernels of interest to us in MURaM is the radiation transport. Related work include the recent acceleration of minisweep, a MiniApp of the Denovo [53] radiation transport application on GPUs [106, 105] using OpenACC. Results demonstrate that OpenACC running on NVIDIA’s Volta V100 GPU boasted an 85.06x speedup over a serial CPU code, which is larger than CUDA’s 83.72x speedup over the same serial implementation. Other MiniApps demonstrating radiation transport approaches include Kripke [6] that uses RAJA [61] and TestSNAP [113] (mimicking communication patterns of PARTISN [19] transport code) investigating different data layout patterns and parallelism using Kokkos and OpenMP offload model [51]. Coarray Fortran-based Sweep3D’s comparable performance to that of the MPI is discussed in [38]. The work on Ardra [73] discusses porting a discrete ordinate transport code to CUDA using the RAJA model with CHAI [66] and Umpire [21] to manage multiple memory spaces.

Accelerating solar physics simulation models through GPU devices is a relatively new direction which attracts broad interest from the scientific community. Several radiation MHD codes are in use in solar physics [69, 59, 83, 111, 80, 101]. While efforts of GPU refactoring are ongoing, currently published production runs are still performed mostly on CPUs.

Kripke [6] that uses RAJA [61] and TestSNAP [113]

## 2.2 Performance Profiling

Code profiling reveals several metrics that show in-depth information about code performance and behaviors. There are various profiling applications that serve a variety of purposes. We have used GNU gprof, ARM MAP and NVIDIA nvprof tools during the entirety of this development process. Since the beginning of this project, nvprof has been deprecated by NVIDIA and replaced with NVIDIA NSight Systems and Compute, which offers many of the same functionalities. For these results, since many of our experiments were conducted before Nsight System’s release, our results are computed using nvprof.

To effectively port and optimize the performance of a code, it is important to gain a high-level view of the code’s performance as-is to identify areas that are most computationally intensive and consume the greatest percentage of the total runtime. Additionally, observing the scalability of the CPU code will give insight into any overheads caused by MPI communication.

Two key metrics observed during CPU profiling would be the percentage of time spent in each routine considering a single MPI rank on a single CPU core and the MPI communication of using many CPU cores as well as multiple compute nodes. Using ARM MAP for the single core profile enabled the creation of a graphical function call graph that showed the flow of code execution as well as the percentage of runtime taken by each step. ARM MAP is also used to capture the MPI communication when using multiple MPI ranks across several CPUs. Nvprof also allows for some CPU profiling, however it was primarily used to observe CPU performance side-by-side with GPU performance.

The single core profiling results are shown in Figure 2.1. From these results, there are three portions of the code that account for 71% of the total runtime: *MHDRES*, *TVDDIFF* and *RTS*. These profiling results are for single-band radiation transport. In multi-band applications *RTS* could be up to 12 times more expensive. Since this is only showing performance of a single CPU core it gives insight as to what routines are computationally intensive without concerns of any MPI overhead. With these results we have a clear starting point of what routines should be immediately targeted for parallelization.

*RTS* is significantly more complicated than *MHDRES* and *TVDDIFF* as the radiation portion of the code contains many smaller subroutines. The parallelization of *RTS* presented many interesting challenges and was handled very differently than the rest of the code.

The MPI profiling using ARM MAP results are shown in Figure 2.2. In contrast to the single core profiling, additional routines like *FFTW* and *DST* affects the total runtime, with *FFTW* being the MPI version of the FFTW 3.8 library and *DST*

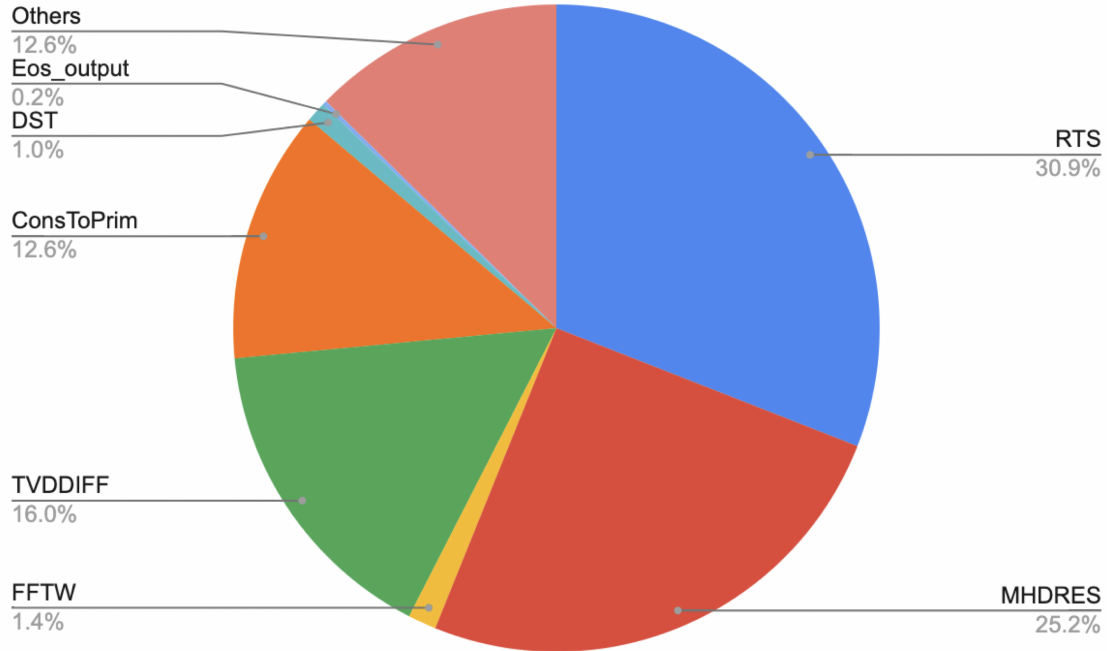


Figure 2.1: MURaM single-core profile results generated with the ARM MAP profiler. 192x128x128 dataset simulation on 1 core of an Intel Xeon E5-2697V4 (Broadwell) CPU.

communicating ghost cells through MPI. *RTS* also includes a significant amount of MPI communication. Figure 2.2 shows that the MPI communication within *RTS* alone accounts for 6.2% of the total runtime. Optimizing the MPI communication is an important task for the performance of both CPU and GPU code.

GPU profiling tools can reveal important information about the achievable performance of a code. In the future, NSight Systems will give an easy interface for creating roofline models [78], but for now we have done some manual arithmetic intensity (AI) computations with nvprof. To determine the AI of several key kernels, we compare the peak FLOP rate (in FLOP/s) and the memory bandwidth (in byte/s) for a NVIDIA V100 GPU to the observed FLOPS and memory read/writes within a given kernel. For a V100 GPU, the peak FLOP rate is 7,000 GFLOP/s and the memory bandwidth is 900 Gb/s. So if the calculated AI is greater than 7.77 (7,000 GFLOP/s / 900 Gb/s), then the kernel is compute bound, or memory bound otherwise. All of the kernels we have measured were memory bound, the AI is as follows: *RTS::integrate* (0.61), *RTS::flux* (0.4), *RTS::interpol* (2.4), *MHDRES* (0.46) and *TVDDIFF* (0.81).

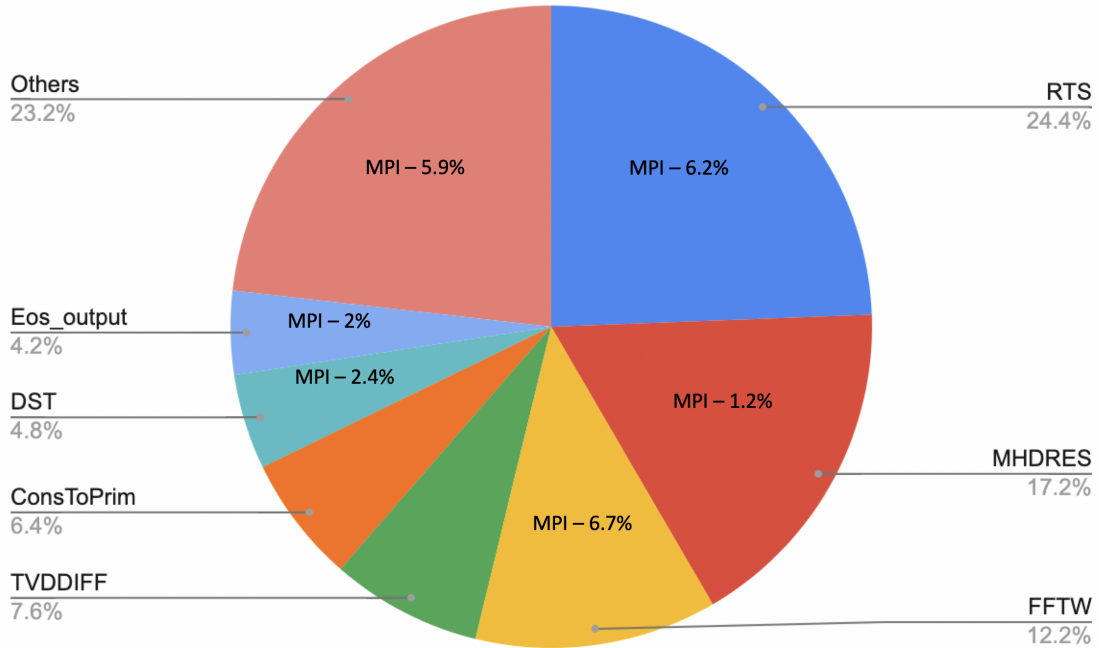


Figure 2.2: MURaM multi-core profile results generated with the ARM MAP profiler. 192x128x128 dataset simulation on 36 cores (full node dual socket) of an Intel Xeon E5-2697V4 (Broadwell) CPU.

Another important performance metric that significantly guided our development process was the achieved and theoretical GPU occupancy. GPU occupancy for NVIDIA GPUs refers to the percentage of warps active at a given time. **Theoretical** GPU occupancy can be determined by examining the GPU register and shared memory usage when a kernel is compiled. Low theoretical GPU occupancy is generally caused by a kernel needing to use too many registers or too much shared memory, which results in the GPU not having enough resources to support using every warp simultaneously.

**Achieved** GPU occupancy is simply the GPU occupancy that is observed during kernel execution. It is ideal to have theoretical GPU occupancy as close to 100% as possible and to have achieved GPU occupancy as close to the theoretical GPU occupancy as possible. There are many potential reasons for achieved occupancy to be lower than theoretical occupancy, some of which was observed in our work with MURaM. The GPU occupancy of several important kernels is shown in Figure 2.3 and using this metric we can identify two key performance issues within MURaM.

Firstly, many of the kernels are reaching very low theoretical occupancy, with *MHDRES* and *TVDDIFF* reaching 25% and 33% respectively. For these two kernels, the low theoretical occupancy is due to an over-allocation of GPU registers per thread. Figure 2.4 shows the potential theoretical occupancy of the *MHDRES* kernel when the registers per thread changes. If any more than 32 registers are allocated per thread the theoretical occupancy will go below 100%, and since *MHDRES* is compiled to use 122 registers per thread, it can only reach 25% theoretical occupancy.

This is an interesting problem in directive-based programming models, such as OpenACC, because the programmer relies on the compiler to assign GPU registers when generating the GPU code. When using a device-specific language, such as CUDA, the programmer can fine-tune this register allocation to a higher degree. In the case of *MHDRES*, the NVHPC compiler has determined that 122 registers is the optimal number to achieve the most performance, and incorporating a hard register limit of 32 registers sees a significant performance decrease. Solving the problem of register pressure may offer a large speedup throughout the code.

Secondly, *RTS::integrate* shows a different problem where the theoretical occupancy is very high while the achieved occupancy is very low. This means that when *RTS::integrate* is compiled, the number of registers and the amount of shared memory assigned is such that the GPU could potentially support every warp running simultaneously. However, in practice only 10% of those warps are used when executing the kernel. This is due to a lack of parallelism being exposed in a way that we are only able to parallelize across two dimensions of our three dimensional domain. Typically, modern GPUs are expected to perform computation on several millions of data points, but in this kernel we are only hitting a few ten thousands resulting in the 10% GPU occupancy.

### 2.3 General approach with OpenACC

Maintaining correctness during the migration of MURaM’s large and complex code-base to support GPUs presented a challenge. To address this key issue, we chose an

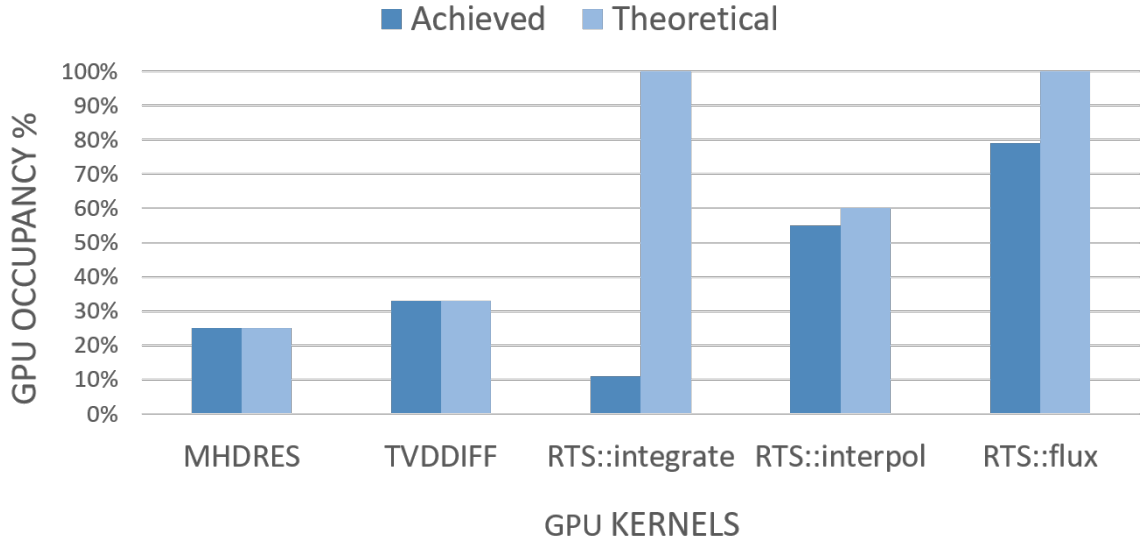


Figure 2.3: Theoretical and achieved GPU occupancy of various GPU kernels within MURaM.

incremental, test-driven development approach throughout. This involved three steps: 1) identifying suitable baseline test cases; 2) creating a correctness validation build-test system; and 3) incremental migration to GPUs by applying OpenACC directives and validation of the results. It is worth noting that steps 1 and 2 required close collaboration with the solar physicists on our team.

We chose a test case with a grid size of  $192 \times 64 \times 64$  for our validation suite because it's small enough to run on a single CPU core while still capturing the solar atmosphere from the upper convection zone into the lower solar corona and therefore testing all implemented physics in the code. We used this setup to generate the CPU reference data required to validate the GPU implementation against. During the porting and development process, its small size allowed us to quickly and repeatedly run the model for the 50 time-steps required by the validation suite without exhausting our limited cluster resources. It's also large enough to decompose and test in different x, y, and z core layouts.

To validate correctness, we developed a validation suite which builds and runs the ported code using different combinations of processor layouts. At specific timesteps, diagnostic variable data is output into files. The test diagnostics can be compared to

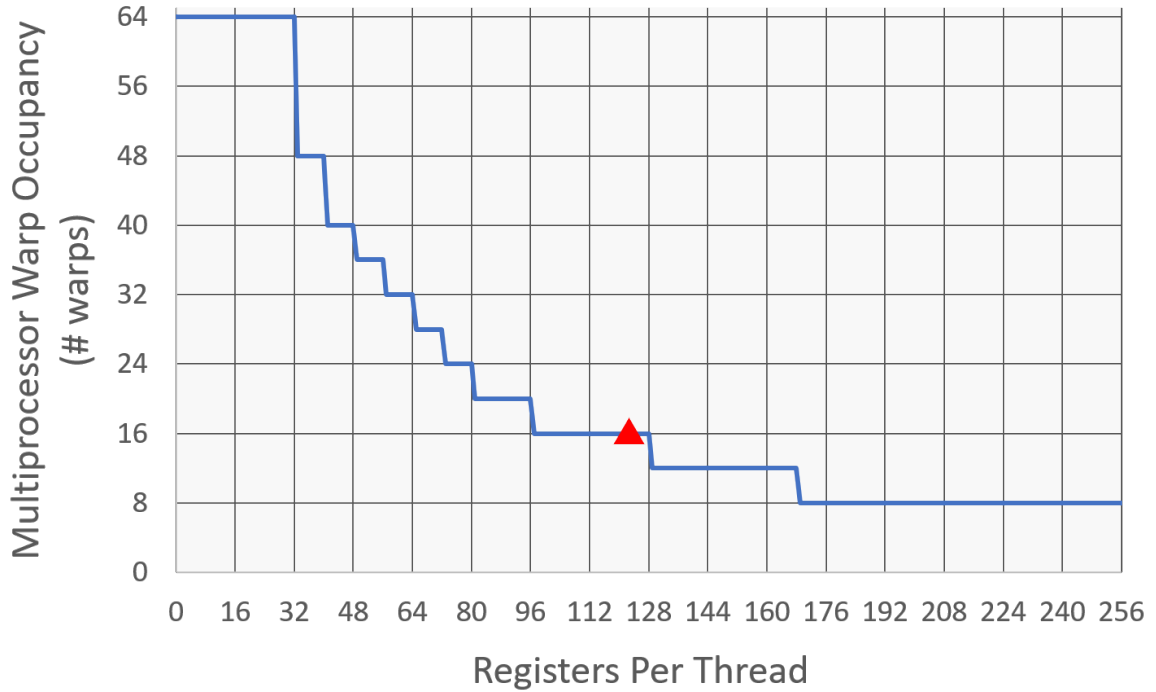


Figure 2.4: Effect of GPU register usage on GPU occupancy in the MHDRES kernel.

reference diagnostics generated by the CPU master code using a matching data set and processor layout. We defined an acceptance tolerance as the variance observed between MPI CPU runs with varying decompositions and core layouts, which is a maximum relative error margin of  $1e-05$ . However, but errors on the order of  $1e-7$  can cause large relative errors at points where the reference data is close to zero. Furthermore, the MURaM domain is heavily stratified in the vertical direction, leading to a density contrast from bottom to top of about  $1e9-1e10$ . To handle this issue, we compute first the relative error on each height layer by normalizing the absolute error between the diagnostic and reference data by the mean value of the reference data at that respective height. We accept a modification if the maximum of the relative errors from all heights is less than  $1e-5$ . Figure 2.5 shows an example error graph generated by our validation suite. These figures are often very helpful in debugging problems that occur at predictable areas of the domain, such as at processor boundaries.

For additional correctness checking we utilized a relatively new feature in the NVHPC compiler called PGI Compiler Assisted Software Testing (PCAST) [9]. This allows direct comparison of data from a reference run of the code to be compared to a current run. PCAST can be used in two modes: automatically running the CPU and GPU version of a kernel then comparing their outputs directly immediately after, or comparing the output to a previously generated PCAST output file. PCAST also has the option to generate all of these comparisons automatically with a compiler flag, however we have been unable to use this feature as it causes a code crash. Regardless of this issue we were still able to apply PCAST manually throughout the code.

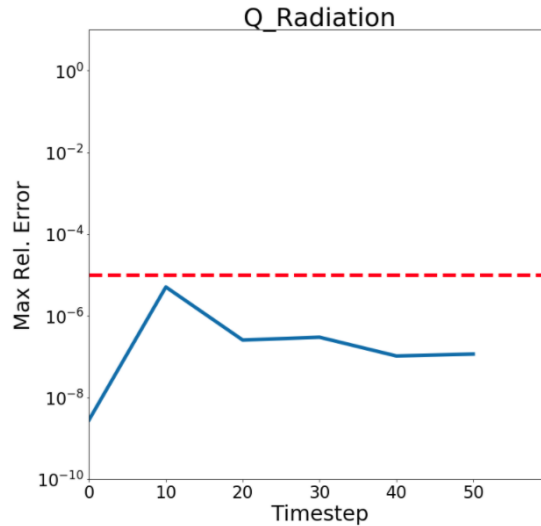


Figure 2.5: Maximum Relative Error in GPU calculated Q\_Radiation over 50 iterations. Red: max threshold and Blue: what was observed

To manually implement PCAST we have created a wrapper macro that could be ignored based on a toggle in the code compilation, or if using a non-PGI compiler. This macro was placed before each kernel and captured the data of all of the significant input variables, as well as after each kernel and captured all significant output variables. Then the code was compiled for CPU and ran for only two timesteps to avoid creating too large of a reference file. Any future GPU runs could then be built with PCAST and compared to the CPU reference pointing out any minor discrepancy between the reference run. Additionally, PCAST includes a feature called patching that will replace

any incorrect values with their reference, allowing us to see isolated errors and avoid error propagation to later tests.

In the incremental refactoring process, we initially target a single loop nest and apply OpenACC directives to have that loop run on the GPU, without focusing on any sort of optimization. Data management is also handled as locally as possible with all input variables copied to the GPU immediately before the loop and all output variables copied to the host immediately after the loop. This allows for a single isolated portion of the code to be parallelized without having any cascading effects on the rest of the code. The key benefit to this strategy is that we can ensure code correctness first and foremost before moving on to any optimizations, which can introduce new problems into the code. Once that portion of the code has been parallelized, we checked for two things. 1) ensure that the code still produces correct results, 2) verify with a profiler that the portion of code is running on the GPU and is behaving as expected.

Frequent code profiling during development gives a very important sanity check at every step of the process. It ensures that the most recent code changes are running properly on the GPU and contain any expected data movement associated with them. Some problems that this profiling can identify are kernels running significantly slower than expected, extra or unexpected data movement and low kernel performance from metrics such as occupancy or bandwidth-bound kernels. Many of these details become very important once we move past the initial parallelization and move on to optimizing the GPU code.

After every important loop was running on the GPU, we began the process of eliminating redundant data movement between the host (CPU) and device (GPU). These copies are expensive, but an artifact of the careful, incremental nature of our porting strategy.

Every core MURaM routine is ported to the GPU with optimized data movement. Additionally, the computational kernels within the radiation transport solver is further optimized. Fully optimizing the rest of the kernels within the code is a clear future direction for the MURaM project.

For the multi-GPU runs in this project we are using OpenMPI 4.0.5 that provides support for GPU-to-GPU MPI data transfers when installed on a machine with compatible hardware. To use this feature, a valid GPU address pointer is passed into the MPI function call. In a language, such as CUDA, this is very straightforward, as the programmer explicitly manages GPU memory allocations. In OpenACC however the GPU memory allocation is hidden from the programmer by the OpenACC runtime. To expose the GPU address pointer OpenACC provides the **host\_data** directive and **use\_device** clause that allows interoperability with GPU-based libraries. While using GPU-aware MPI with OpenACC is very simple, verifying that device-to-device data transfers are working as expected is a challenge. The only way that we currently know to verify this functionality is by profiling the MPI GPU application using a GPU profiler and checking explicitly that device-to-device (or DtoD) data transfers are occurring. Within our development system we have observed a significant performance difference between basic CUDA-aware MPI and GPU Direct RDMA. Through profiling, we could see that the GPUs were not performing DtoD transfers despite using CUDA-aware MPI. After rebuilding the OpenMPI 4.0.5 library with GPU Direct RDMA enabled, the profiler showed DtoD transfers were now being used.

## 2.4 Optimizing Radiation Transfer

It is well known that computing 3D radiation transport (3D-RT) is extraordinarily expensive, depending on two angular dimensions (i.e. the zenith and azimuthal angles) and three spatial dimensions. RT solvers are typically iterative, further adding to the cost. However 3D-RT is essential for a number applications in solar and stellar physics, including realistic 3D simulations of stellar photospheres [86, 87], and the accurate modelling of strongly scattering spectral lines, such as those in the solar chromosphere [77].

In the case of MURaM’s 288<sup>3</sup> reference test case with one band, the radiation transport solver (RTS) is the most expensive part of the calculation, accounting for nearly half the time on a single, dual-socket CPU node. More realistic, multi-node 3D

radiation transport, in which the RTS must be called once per frequency band, will drive the proportion of time spent in RTS even higher.

For this reason RTS has received the most optimization effort during our GPU port. However, RTS exhibits a wavefront data dependency pattern in its primary computational kernel as well as several blocking MPI communications. This section will focus on optimizing RTS while considering the complexities and limitations of the solver’s underlying algorithm.

The MURaM code uses short characteristics to solve for the radiation field [71]. This method involves integrating the radiation transfer equation along a ray using values of intensity and opacity interpolated from the neighbouring grid cells. To calculate the mean intensity and radiative fluxes at each grid point a set of 3 rays per octant are used to integrate over the unit sphere. The wavefront data dependency also provides a challenge for an efficient treatment in a decomposed domain, since it implies a serial dependence of MPI processes. In MURaM this challenge is overcome through an iterative approach. The solver uses 3D domain decomposition and computes all sub-domains in parallel and iterates until the intensities at the boundaries of each sub-domain are within a prescribed relative tolerance of typically  $10^{-4}$  to  $10^{-3}$ . Convergence is typically achieved after about 3 iterations, since a substantial fraction of the simulation domain is optically thick, i.e. the radiation transport is very local. However, this treatment limits strong scaling, since a large number of small sub-domains with a moderate optical depth increases the required iterations (including communication) to convergence. A multi-band opacity scheme is used in order to efficiently include the frequency dependence of the radiation transfer problem in the solar atmosphere. For feedback into the MHD code the radiative heating or cooling at each point in space is calculated. Figure 2.6 shows the structure of the core computational loop in RTS. Expanding on the routines described in Section 2.1, *writebuf()* and *readbuf()* pack and unpack a buffer that will be exchanged with neighbors in the *exchange()* routine.

The most important routine within RTS is called *integrate()*, which is executed 50-100 times per timestep. However, unlike the other routines, *integrate()* introduces

```

for( < x direction > )
  for( < y direction > )
    for( < z direction > )
      for( < angle > ) // Total 24 rays
      {
        interpol();
        while(g_maxerr > THRESHOLD)
        {
          readbuf();
          integrate();
          writebuf();
          exchange();
          double err = error();
          MPI_Allreduce(&err, &g_maxerr);
        }
        readbuf();
        flux();
      }
}

```

Figure 2.6: A simplified code example of the core computational loop in RTS.

a wavefront data dependency pattern that restricts the parallelism attainable within it. This means that even though we are working with a 3-dimensional domain, we can only achieve two dimensions of parallelism; one dimension of our problem will have to remain sequential as displayed in Figure 2.7.

This data dependency introduces several performance challenges that must be addressed: 1) GPU kernels with 2-D data parallelism are very small and under-utilize the compute processors of GPUs, 2) having one dimension of the domain remain sequential means we must launch possibly hundreds of small GPU kernels to compute the entire domain, introducing runtime-dominating kernel launch overhead, 3) since each ray sweeps in a different direction, some rays have better memory access striding than others, meaning that some rays take several times longer to compute than others.

Since *integrate()* has to deal with a data dependency, there are three possible scenarios that we refer to as an x, y or z dependency. These names depend on which of the three dimensions must remain sequential, as outlined in Figure 2.7. An x or y

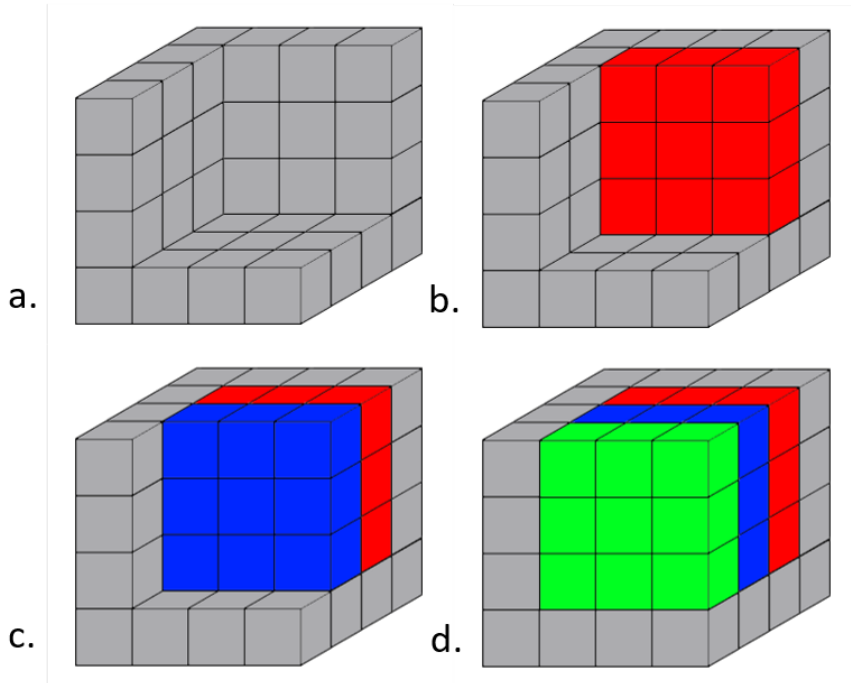


Figure 2.7: Representation of the wavefront dependency in RTS integration. a) shows the preloaded boundary conditions with no computation done. b) shows the first set of grid points in red to be computed in parallel. c) shows the next set of grid points to be computed in parallel, each blue grid point depends on up to four of the red grid points. d) shows the last grid points to be computed in parallel, each green grid point depends on up to four of the blue grid points.

dependency are parallelized in such a way that the  $z$  direction is handled by the inner most loop, which allows for vectorization on a perfectly strided array. However, for the  $z$  dependency, the  $x$  direction is the inner most loop which results in very bad memory striding. The  $z$  dependency is up to 10x slower than  $x$  or  $y$ , depending slightly on the dataset. The domain decomposition and dataset dimensions will determine how often the rays exhibit the  $z$  dependency, but in typical runs it happens often.

Our solution to this was to create a second version of all of the arrays used within *integrate()* to have an alternate transposed version. This includes the intensity which is calculated within *integrate()* and the coefficients which is calculated within *interpol()*. *Interpol()* uses three other arrays to calculate the coefficients, but they are read-only within *interpol()*. We transpose the three read-only arrays in *interpol()* once per timestep before the main computation happens, then we ensure that the coefficients

are written in the same order that *integrate()* will read them. Then we transpose the intensity array back before the flux values are calculated.

This results in the *integrate()* kernel being able to work in perfect stride even when encountering a z dependency pattern. The cost is the few extra transposes that have to be done, but in our test runs we are still seeing a significant improvement with *integrate()* running 5x faster overall, and RTS as a whole running 3x faster. This is ultimately a performance loss for the CPU, but is easily manageable using branching compilation with `ifdef` in C++. This “data transposition” version of RTS performed well for the GPU in every test case we have tried.

One strategy for addressing the 50% idle time of *integrate()* is to increase the amount of computations performed in each kernel. The two kernels that can be most easily merged are *interpol()* and *integrate()*. However, as seen in Figure 2.6 *integrate()* is inside of a convergence loop, meaning that it is typically called more times than *interpol()*. This is because the coefficients calculated in *interpol()* only need to be done once for each ray and the results are stored within an array to be read from in *integrate()*. Instead of storing the coefficients into an array we moved their calculation into the *integrate()* kernel and used them directly. This means that any ray that takes more than one iteration to converge will now have to compute these coefficients more often, but with the possible benefit that now with more work to do the *integrate()* kernel may be more efficient.

The results of this change varied in our test cases. For smaller datasets, such as  $64^3$ , where the GPU occupancy was at its lowest and the kernel had less work to do, the *interpol()* and *integrate()* merge resulted a speedup of around 30% for RTS overall when compared to a GPU run without this change. This optimization might be beneficial for future MURaM forecasting scenarios, where high throughput (capability) would be of paramount importance, requiring strong scaling to smaller per-GPU problem sizes to achieve. However, problem sizes more representative of the current use of MURaM simulations showed a significant performance decrease of up to 50%.

In the chromosphere the radiation field can be strongly scattering, meaning the radiation source function is strongly dependent on the intensity. For strong-scattering problems, the radiation transfer problem becomes more non-local, increasing iterations to convergence. In this case it is preferable to use a Gauss-Seidel convergence scheme [117]. As the intensity is integrated along the ray, this scheme will require updating the source function for each point along the ray, and then using the 'new' source function to integrate the intensity at the next point along the ray. This algorithm requires a combined treatment of interpolation and integration. This modification will therefore be of interest to apply the GPU short-characteristics scheme to broader range of stellar problems. For this reason, this variation of RTS may be an important direction for future work.

Another challenge that we addressed was the low GPU occupancy observed in *integrate()*. One technique tried was to combine the computation of several rays into a single kernel. Our first approach was to use OpenACC asynchronous programming to queue the work of all 24 rays simultaneously on the GPU. The hope was that the GPU would be able to overlap the computation of multiple rays since a single ray is only using about 10% of GPU resources.

In practice, we observed through the GPU profiler, nvprof, that this method did allow for a little bit of overlap between the computation of multiple rays, but far less than what we would assume to be theoretically possible. In our experimentation, this method only improved performance by 5%. Typically when using OpenACC asynchronous programming, GPU computation overlaps with either CPU computation or CPU/GPU data transfers, instead of overlapping multiple computational kernels. Additionally, there is no way to lock specific streaming multiprocessors to specific kernels within OpenACC; if this existed it could possibly be a viable solution to the problem we are facing.

Next, instead of relying on the OpenACC **async** directive to overlap computation, rays that exhibit a similar dependency pattern are combined with an outer parallelizable loop. We can identify 6 groups with 4 rays each that will exhibit a similar dependency

pattern with the group. This means that the arrays used within the affected RTS computation routines must be increased in size by a factor of 4. This would increase the work done per *integrate()* kernel by a factor of 4 as well. When observing only the time spent in *integrate()*, and using a predetermined number of iterations, these changes seem to provide a significantly better performance than the baseline code. Similarly to previous methodology, this RTS variation received a larger performance increase for smaller datasets, likely when the idle time between kernels and GPU occupancy is at its lowest.

Implementing these changes altered several different parts within RTS, which makes determining the exact performance benefit difficult. Since many rays would now be overlapped, instead of computed one-at-a-time, the way to determine convergence is changed to a global convergence instead of a per-ray convergence. This could have the added benefit of reducing the total number of iterations needed overall. Additionally, since 4 rays can be computed simultaneously, there will be 4 times fewer *exchange()* function calls and a factor of 24 times less *error()* function calls and associated MPI all-reduce routine calls which could greatly reduce the MPI communication within RTS. Lastly, since half of the rays are moving upward and half are moving downward, it is possible that we could use 3 groups of 8 rays instead of 6 groups of 4, as rays moving upward versus downward could still exhibit the same dependency pattern. Fully evaluating the effect of this optimization remains as future work for the project, and are not utilized when discussing final performance results.

## 2.5 Deconstructing the OpenACC runtime for performance

Anytime a GPU kernel is launched, a certain amount of overhead is incurred. When this overhead is very small compared to the time spent in the GPUs parallel computation, good performance is still achieved. However, in edge-cases such as the *integrate()* routine, this overhead can become a dominant factor in the overall execution time. The OpenACC **async** directive can be used to hide a large portion of this overhead by queuing many kernels on the GPU before they are run. While the earlier

kernels are being computed, the later kernels are being pre-loaded which allows overlap between the execution of the current kernel and the setup of the next one. However, by closely analyzing the GPU profiler, nvprof, it was clear that even with this optimization *integrate()* was still performing sub-optimally due to kernel launch overhead.

In our original port of *integrate()* we observed a  $35\mu\text{s}$  gap between each kernels' execution. Since each call of *integrate()* possibly requires hundreds of kernel launches (depending on the size of the dataset) and *integrate()* may be called several times for each ray, we expect that a gap of  $35\mu\text{s}$  to account for a total of 0.2 seconds of the GPU compute resources being idle per timestep. The profiler revealed that there is some amount of data movement happening before and after each kernel, which is a significant factor of this overhead. These data movements were caused by a pointer translation. This is likely related to the host address pointer needing to be translated with the device address pointer within the OpenACC runtime.

To address this, we changed how the arrays in *integrate()* were being allocated; instead of allocating them on the host first then the device, we allocated them only on the device. With this change the overhead has been reduced to  $26\mu\text{s}$  between kernels. This is an improvement, but the profiler showed that something is still happening between these kernel launches. We found that this has something to do with how C++ class members are handled. Since all of the arrays used within RTS are stored within a class, we created a pointer outside of the class that referenced the needed arrays, and used those new pointers in *integrate()*. After this change the overhead is further reduced to  $5\mu\text{s}$  between the kernels. Considering that a single execution of the *integrate()* kernel also takes  $5\mu\text{s}$ , the GPU processors are idle for about half of the time while computing *integrate()*. In order to address this issue, we would need to refactor the code, as discussed in the next section.

## 2.6 Scaling Results

In this section, we present the results of running MURaM on multiple CPUs and GPUs while demonstrating parallel efficiency.



Figure 2.8: Nvprof profiler highlights the process of reducing kernel launch overhead in the RTS *integrate()* kernel. (a) Overhead between 2 launches, (b) Reduced overhead between kernels launches, and (c) Further reduced overhead

### 2.6.1 Experimental Setup

The Cobra system consists of 3,424 compute nodes, each containing two Intel Xeon Gold 6148 Skylake (SKL) processors (20 cores at 2.4 GHz) and 100 Gb/s OmniPath interconnect. There are 64 GPU nodes with 2 NVIDIA Tesla V100-PCIE-32GB per node utilizing 32 GB HBM2 for a total of 7.9 TB HBM2 across all nodes. The following compilers and libraries are used for the performance results:

- CPU: Intel 19.1.3, Intel MPI 2019.9, MKL 2020.2 and FFTW-MPI 3.3.8
- GPU: NVHPC 20.9, CUDA 11, OpenMPI 4.0.5 with UCX 1.8.0 and FFTW-MPI 3.3.8 with multithreading

(Note: PGI compilers have been renamed since October 2020 to NVHPC [89])

## 2.6.2 Single GPU Performance

Our initial project focus was optimizing performance on a single GPU. In practice, the simulations that are run with MURaM will require multiple GPUs, but using a smaller dataset allows us to study GPU performance separately from the scalability of the MPI implementation. Using a single V100 GPU, the average time to simulate one single-band (or gray scale) timestep with the  $288^3$  dataset is 2.285 seconds. This is a 3.23x speedup over the 7.382 seconds for the same simulation on a single socket, 20-core Skylake CPU, and 1.73x speedup over 3.949 seconds on a dual-socket, 40-core Skylake CPU node. Using the Garsching measured/benchmark roofline values of memory bandwidth on Cobra for their Intel Gold 6148 SKL processors [109] and the V100 GPU, the benchmarked memory bandwidth ratio is 4.038x. Since we see 1.73x, and we are memory bandwidth limited, we are realizing 42.8% of the peak bandwidth for the full application and leaving 2.3x on the table.

## 2.6.3 Strong Scaling

Strong scaling is defined as how the solution time varies with the number of processors for a fixed total problem size. For our strong scaling experiment we use a  $288^3$  dataset divided across 8 GPUs and 8 fully subscribed CPU Skylake nodes (40-cores per node). The CPU runs use -O3 and AVX512 flags and the GPU runs use -O3. We are also comparing the strong scaling performance of the code with gray band (1 band) and colored band (4 band and 12 band). The colored band increases the workload in RTS and is proportional to the scaling of RTS. Figure 2.9 shows CPU and GPU strong scaling of MURaM with this configuration. CPU strong scaling for this case is not explored past 8 CPU nodes (320 total cores) because past this point the number of points per core becomes too low to offer meaningful data. The performance of the code is measured in millions of sites updates per second (Msite/s). Cobra contains 2 GPUs per node, so increasing to 4 GPUs requires inter-node communication. From 1 to 2 GPUs the code scales very well in both the single and 4-band case, roughly doubling the throughput. However, moving from 2 to 4 GPUs only increases throughput by 1.63,

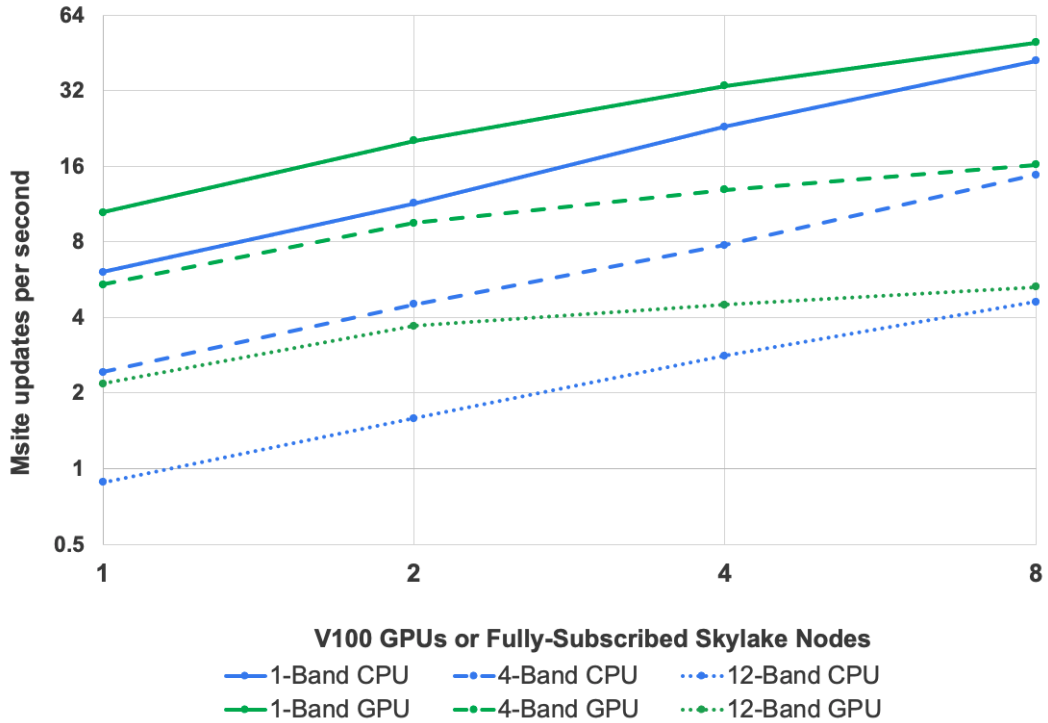


Figure 2.9: CPU-GPU strong scaling of a multi-band  $288^3$  dataset

which is possibly due to the higher cost of internode communication. Results shown for the 8 GPU runs indicate that a more optimal core decomposition for multi-node scaling may be chosen after further testing.

We also gathered strong scaling results using up to 96 GPUs for a  $288 * 576 * 576$  single-band dataset, as shown in Figure 2.10. These results show the strong scaling performance of the RTS routine as compared to the overall simulation: both seem to do reasonably well. The scaling efficiency of both RTS and the full model begin to decrease when the dataset is split over more than 32 GPUs, and the number of points per GPU falls below one million.

### 2.6.4 Weak Scaling

Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size per processor and gives a great deal of information about the MPI communication and overall scalability of the code. Figure 2.11 shows a

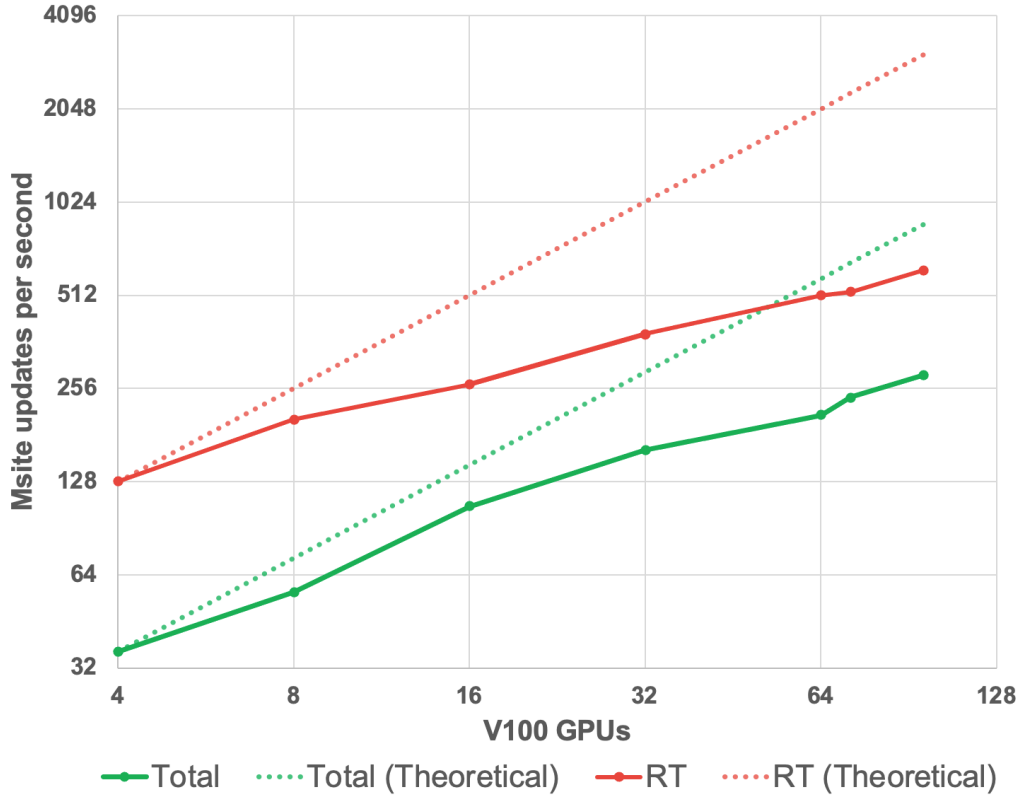


Figure 2.10: GPU Strong Scaling of a  $288 \times 576 \times 576$  dataset

breakdown of the GPU weak scaling with respect to the different routines of the code. This figure is also measured in terms of seconds per timestep. We use a  $288^3$  dataset per GPU for this experiment.

### 2.6.5 Results Summary

There is an increased computational cost moving the simulation from single-band to multi-band, as seen in Figure 2.9. For N-bands, the interpolation, integration, and flux calculation routines within RTS run N times more often. Additionally multi-band RTS requires more iterations to convergence for bands that capture the highly structured upper photosphere. Since the number of required iterations increases with the number of MPI processes and each iteration requires additional communication, this has a negative impact on the strong scaling results shown in Figure 2.9.

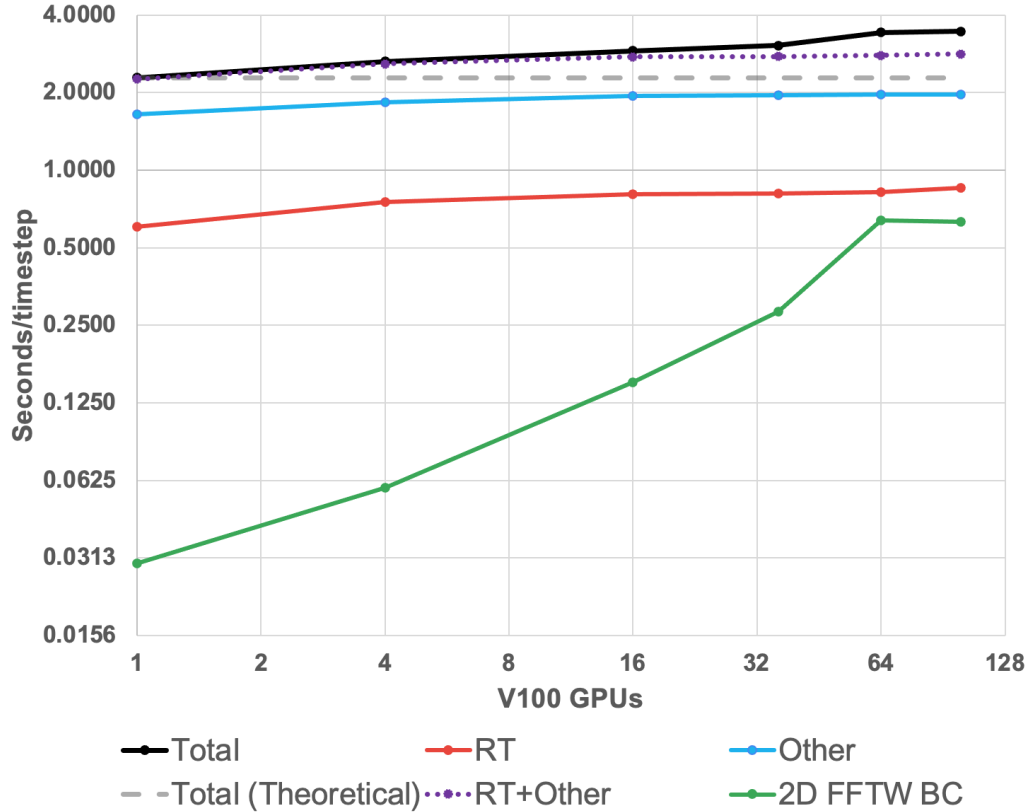


Figure 2.11: GPU weak scaling of a  $288^3$  dataset for RTS & other MURaM routines. Note that FFTW is a CPU only library

Figure 2.10 highlights a problem with the strong scalability of RTS on GPUs, where increasing the number of GPUs has diminishing improvements on performance. This is likely due to the increasing amount of inter-node communication as the GPU count rises, as well as the reduction of points per GPU. We believe that some of the future work outlined will have a positive impact on this strong scalability.

Lastly, from Figure 2.11 it is clear that between 1 to 64 GPUs the scalability suffers due to the FFTW library. Currently, we are using multi-threaded FFTW on the CPU with results being copied to the GPU after computation. As a future direction, we plan to explore a GPU-enabled FFT library.

## 2.7 Summary and Take-aways

Porting large applications such as MURaM highlights the key benefits and drawbacks of using directive-based models. Overall, we were successfully able to create a portable codebase that can target both CPU and GPU architectures using a mixture of MPI and OpenACC. The performance we achieve in many kernels is reasonable with our expectations of what other codes have achieved on the chosen GPU. We likely could have reached higher performance if we committed to rewriting the code entirely in the CUDA programming language, but we would completely sacrifice the codes portability to CPU architectures and have required far more development hours to accomplish such a task.

However, it is clear that by using OpenACC we lose some control over low-level optimizations of the GPU hardware. Namely, with large kernels (i.e more than 100 lines of code) we observe a large over-allocation of GPU registers which limits the possible number of concurrent threads. It is difficult to accurately identify possible ways to reduce the register usage, as some may be hidden by the runtime and others may be required to generate correct code. Additionally, in the case of radiation transport, OpenACC in many ways lacks the ability to optimally parallelize edge-case codes such as those with common, but complex data dependencies. CUDA offers several ways to optimize our integrate kernel, but none are available natively in the OpenACC standard, and no “work around” exists within any of the current OpenACC compilers.

## Chapter 3

# LIMITATIONS OF DIRECTIVE-BASED MODELS FOR REAL-WORLD APPLICATIONS

Large parts of the text of this chapter has been published in the Tenth Workshop on Accelerator Programming and Directives (WACCPD 2023), ACM <https://doi.org/10.1145/3624062.3624606>.

This chapter will extend the MURaM project and focus on recent developments and experimental features, including in-depth discussions of the limitations faced by our team that are inherent to such programming models, and various workarounds and suggestions to the OpenACC community to improve the programming model, as well as updated scaling results with various new code optimizations running on modern GPU hardware.

### 3.1 Introduction

Previous production runs use up to 20,000 CPU cores, however in it's current status it is not feasible for MURaM to reach the 100,000 to 1 million core range. Instead, our science goals require the use of exascale technologies. In the context of the US Department of Energy (DOE)-led exascale program [47], this means focusing on Graphics Processing Units (GPUs). There are several possible language/compiler pathways to exascale computing with GPUs: vendor-specific languages like CUDA [88], directive-based standards such as OpenMP [108] and OpenACC [64]; and domain-specific languages (DSL) such as Stella [60], LFRic [15] and its successor GridTools [112].

The GPU port of MURaM has three keys goals: (1) improve the throughput of the model to the max extent possible. (2) preserve portability of the application, at a minimum supporting both CPU and GPU execution. (3) achieve reasonable performance

portability between CPU and GPU performance, i.e maximize performance of each model without sacrificing the performance of the other. For this project the directive-based parallelization approach appeared to offer the best prospect of meeting the first two goals, while performance portability was an open question. Due to compiler availability and maturity at the start of this project, as well as experience within our team, we have selected to use OpenACC for porting MURaM to GPUs. OpenACC has been widely used in the past several years to migrate large scale applications demonstrating maturity and stability with the compiler implementations of the high level features. Some of the large (including production) applications that uses OpenACC include MPAS [124], ANSYS [102], Icosahedral non-hydrostatic (ICON) [103], LQCD Monte Carlo [25], and VASP [81].

Hardware portability is vitally important to the users of MURaM and is preferred to at least maintain a functioning CPU and GPU code. MURaM is also continuously under development implementing new science and optimizing the existing code in many ways, meaning that ideally we require only a singular source code so that changes do not need to be implemented multiple times for different hardware. OpenACC allows the programmer to augment a code written in a common, CPU-based language with special compiler comments called *directives*. These directives can provide a compiler with additional information such as available parallelism within the code or communication (i.e data transfers) with an accelerator device. A compiler could then convert the code with these additional directives to generate parallel code for a variety of accelerator devices. This allows for a single source code that can be built and run for many different architectures, limited by compiler support.

While the generality of OpenACC is very convenient for MURaM, it does present multiple downsides. Expressing low-level hardware features is often difficult or impossible. While some compilers may provide options for hardware-specific features (such as CUDA unified memory for NVIDIA GPUs), these features are not standard in OpenACC, and may not be available in other compilers and on other hardware. There simply are some advanced features present for these GPU hardwares that are inaccessible

to use in a purely OpenACC environment. While we greatly value the portability afforded by OpenACC, performance compromises must be carefully considered to feasibly achieve our computational goals. Fortunately, OpenACC does provide interoperability with other GPU languages, meaning that there are routes to achieve the desired results at the cost of losing portability in certain portions of the code.

The contributions of this chapter include:

- Enabling the study of scaling of MURaM on large scale machines
- Analyzing the impact of radiation transport (RT), a critical component of MURaM, on high resolution and large scale simulation studies in solar research
- Discussing steps to overcome performance limitations while using a directive-based programming model and how that affects portability
- Creating robust test cases for complex simulations using a checkpoint-based system with the PCAST debugging tool.
- Narrating the practical challenges while targeting the convergence between computer science and domain science

## **3.2 Methodology**

This section will focus on various designs and implementations included in MURaM to achieve a GPU-accelerated code. This includes discussions of our experiences with GPU-enabled libraries and tools, and provides insight of what developing a large, real-world application is like in modern GPU models.

### **3.2.1 High-performance FFT Libraries**

MURaM uses FFTs to formulate the magnetic top boundary condition of the simulation domain, which is a potential field extrapolation. While these routines are only called at the top layer of the simulation domain, we have previously observed the boundary routines become a bottleneck in large simulations due to poor scaling.

Prior versions of MURaM has used the FFTW3 library [55] with MPI enabled for calculation of the 2-dimensional upper boundary region. The GPU port of MURaM uses far fewer MPI ranks (equal to the total number of GPUs), so the use of FFTW was changed to run multi-threaded in addition to MPI enabled. Regardless of these changes, we observed poor weak-scaling in the boundary calculation, resulting in the 2nd largest limitation to MURaM weak-scaling (second to RT).

Due to this we have explored other FFT libraries and have settled on heFFTe [18], which is designed for high-performance FFT with exascale technologies. MURaM users now have four options for the FFT calculation: (1) MPI-enabled FFTW3, (2) MPI-enabled + multi-threaded FFTW3, (3) heFFTe (CPU) with FFTW3 backend, and (4) heFFTe (GPU) with cuFFT backend.

We observe that both of our heFFTe versions outperform and out-scale the old FFTW implementation. Currently, even for GPU runs of MURaM we opt to use heFFTe on the CPU as it performs slightly better in our testing. MURaM performs a 2D-FFT calculation along the upper boundary of the domain, meaning that the computation time is overall fairly small. Additionally, kernels needed for the field extrapolation are loaded in from a file, meaning that if running heFFTe on the GPU that data would also need to be transferred. Because of these two conditions heFFTe CPU slightly outperforms heFFTe GPU.

However, there is some early development for an experimental feature to perform the kernel calculation on-device by performing several additional FFT computations, instead of reading them in from a pre-generated file. From preliminary testing we still see heFFTe CPU outperforming our GPU implementation, but the implementation could potentially be optimized further.

### 3.2.2 Debugging and Test Cases with PCAST

Since MURaM is an iterative simulation where each computational component feeds into the next it is unrealistic to make isolated unit tests that fully and accurately encompasses each feature of the code. In the past we have created dedicated “test”

simulations, where we would run the simulation and carefully analyze the output by hand at various timestep checkpoints. Once we were confident that the simulation ran correctly we would save those results as the “golden run” and compare any future outputs to those existing results. These tests inform us when a simulation has failed, but does not accurately identify which specific portion of MURaM caused the ultimate result to be incorrect.

We are moving to a more robust approach using the PCAST [9] tool from the NVIDIA HPC compiler. PCAST allows the inclusion of detailed “checkpoints” throughout the code. This allows at-runtime comparisons of the state of the current simulation to a known-good run. PCAST allows for two key configurations: (1) compare data found in device memory to identically formatted data in host memory, and (2) compare data in host memory to identically formatted data from a previously recorded PCAST file.

*Configuration (1)* enables testing of individual parallel kernels to ensure that it produces correct results compared to the kernel running serially. With this one could ensure that the GPU execution of a code does not alter the results of the baseline CPU code. *Configuration (2)* does allow for a slightly less robust comparison of CPU vs. GPU execution, but also provides the option to compare CPU vs. CPU as well.

Our use of PCAST primarily focuses on configuration (2). Before each kernel, we define a set of inputs, which is all data used within the kernel. Some of these checks may be redundant to previous PCAST calls, but ensures that future code development on one kernel does not affect the PCAST check of another kernel. Then the kernel is executed on whatever device is currently being used. Finally, after the kernel we define a set of outputs, which is all data altered from within the kernel. To utilize PCAST we must first create a *golden run* which creates the initial PCAST comparison file. This golden run does still need to be verified by hand for correctness. Afterwards, any subsequent run of MURaM using the same dataset and decomposition can automatically be compared to that previous run. If any of the PCAST checkpoints are incorrect, we immediately know exactly where in the code the problem originates from.

We only want PCAST enabled when doing debug runs and PCAST is only available in NVIDIA compilers, meaning we have it default disabled if using any other compiler. Allowing for direct CPU to GPU comparison, such as outlined in configuration (1), is a very unique feature that only realistically functions when tied to a compiler. However, for our general use of PCAST we would be interested in a stand-alone library that includes the CPU comparison behavior described to enable PCAST functionality in all other compilers. In the past we have observed minor discrepancies between different compilers in the CPU code, and having such a library would help identify and understand them.

### 3.2.3 Optimizing Radiation Transport

It is well known that computing 3D radiation transport (3D-RT) is extraordinarily expensive, depending on two angular dimensions (i.e. the zenith and azimuthal angles) and three spatial dimensions. RT solvers are typically iterative, further adding to the cost. However 3D-RT is essential for a number applications in solar and stellar physics, including realistic 3D simulations of stellar photospheres [86, 87], and the accurate modelling of strongly scattering spectral lines, such as those in the solar chromosphere [77].

In the case of MURaM's 288<sup>3</sup> reference test case with one band, the radiation transport solver (RTS) is the most expensive part of the calculation, accounting for nearly half the time on a single, dual-socket CPU node. More realistically, multi-node 3D radiation transport, in which the RTS must be called once per frequency band, will drive the proportion of time spent in RTS even higher. For this reason RTS has received the most optimization effort during our GPU port. However, RTS exhibits a wavefront data dependency pattern in its primary computational kernel as well as several blocking MPI communications.

Several previous alterations have been made to RT [122] for GPU portability and performance. In summary, various multi-dimensional arrays (i.e pointer-to-pointer structures) have been flattened into contiguous arrays. Various arrays are allocated

only on the GPU to avoid the need for pointer translations, significantly reducing kernel launch overhead in certain circumstances. Read-only arrays are duplicated and transposed for better memory access patterns for certain radiation ray directions. CUDA-aware MPI used throughout RT.

We are now experimenting with additional GPU features in an attempt to further optimize the critical kernels within RT. Section 3.3 focuses on various optimizations to RT’s core integration kernel and includes discussions of the limitations we face further improving RT within the confines of OpenACC.

### 3.2.4 MURaM I/O

The MURaM code currently uses parallel MPI-IO for reading/writing 3D snapshots, and single process writes for smaller datasets (2D slices). For production runs on both CPUs and GPUs the typical I/O time accounts for less than 10% of the total simulation time. There are current efforts to change the I/O scheme to ADIOS, with integrated lossy data compression, though more development and analysis needs to be performed to comment on the effectiveness for MURaM.

### 3.2.5 Singularity Containers

Containers provide a robust solution for packaging the codes and offer various advantages like reproducibility, portability, and security. Containers can provide a massive advantage for MURaM, where you have multiple software dependencies and build/run configurations. With this idea, we have invested some time in developing containers for the MURaM code. Singularity is a type of container technology designed to use on a cluster where security is of primary concern. With most of MURaM’s science experiments running on closed compute clusters, we used singularity as our container technology. We wrote a singularity definition file to build a singularity container for MURaM. This definition file contains recipes to build various software stacks necessary for MURaM. We start with a basic Ubuntu system and install the essential software modules. Then we install the compiler for the MURaM OpenACC code, i.e., NVHPC

compiler suite with necessary NVIDIA drivers and CUDA toolkit. On top of the NVHPC compiler, we build the OpenMPI communication library with GPU direct capability, FFTW3 library, and heFFTe library with CUDA support. Using the above compiler and libraries ensures the MURaM support for multiple architectures and clusters. The container is currently in the testing phase, but will eventually become available for public use in MURaM’s repository.

### 3.2.6 Programming Struggles

During the integration of OpenACC in MURaM our team has encountered several issues relating to the various compilers and software stacks used. To give insight into the headaches encountered when using such models for real-world codes, the following are the three largest problems we have had to work around to bring MURaM to GPUs: (1) C+ multi-dimensional arrays (i.e pointer-to-pointer structures) are not clear how they are mapped to GPU memory, and have caused us random segmentation faults in some cases. We have since moved to using almost exclusively only flattened, 1-dimensional arrays. (2) “data partially present” errors inconsistently throughout the code. These errors typically mean that some array is not replicated to the GPU in full. However, we have seen this error in many places when using small, statically sized arrays. When an array size is clearly known by the compiler it is often able to handle the array automatically without explicit data copies, but in some portions of the code, and sometimes only with specific compiler versions, we instead receive the error and code crash. (3) performance degradation of certain kernels in more recent compiler versions. Some results in Section 3.4 use version 21.3 of the NVHPC SDK because it provides better performance in some kernels than the more recent 22.7 or 23.1 versions. Initial profiling suggests it is related to the heuristics for local memory, but it is difficult to understand the differences fully.

### 3.3 Exploring Performance Vs Portability

OpenACC provides a generalized parallel model for interacting with offloaded parallel hardware that often limits low-level user control and omits architecture-specific programming features. OpenACC allows for interoperability between itself and other programming models which could provide access to these architecture-specific features at the cost of code portability. Such portability is desired from MURaM users and we attempt to maintain a careful balance between performance and portability.

In this section we discuss performance limitations we are experiencing from OpenACC and discuss possible solutions within OpenACC. Lastly, we explore performance gained when using hand-tuned prototypes written in CUDA to understand the possible performance that could be seen if adopted by OpenACC, removing the compromise between performance and portability,

#### 3.3.1 Limitations of OpenACC

The *Integrate* kernel in radiation transport is restricted by a serial data dependency in one of the three Cartesian dimensions of our data. We define this dependency as either “forward” or “backward” depending on if the associated loop increment is positive or negative respectively, as well as which of the three dimensions (X, Y or Z) needs to be restricted. The remaining two dimensions can be parallelized in any manner without issue. Figure 3.1 gives an outline of the organization of the parallel loops within this kernel, assuming the Y direction must be restricted and is moving forward.

This implementation uses the two parallelizable inner loops as a single kernel that is called in sequence depending on the size of the serial dimension. This means in a typical simulation this kernel will be called hundreds of times per integration, and *Integrate* is called several times per radiation ray, with 24 rays total. This is further exasperated when running a multi-banded radiation simulation, resulting in *Integrate* being called thousands of times per simulation timestep, and hundreds of thousands of individual, small kernel launches.

```

for(int y = 1; y < Ny; y++) {
    #pragma acc parallel loop async
    for(int x = 1; x < Nx; x++) {
        #pragma acc loop
        for(int z = 1; z < Nz; z++) {
            <Integration Calculation>
        }
    }
}
#pragma acc wait

```

Figure 3.1: Basic loop structure of the Integrate RT kernel, assuming the dependency is in the Y dimension and moving forward.

We experience a lack of data parallelism within this kernel, as well as significant overhead in terms of kernel launching, synchronization and data movement. This is somewhat alleviated by queueing multiple kernel launches within an OpenACC queue by using the *async* directive. We have also found that the kernel launch overhead is greatly alleviated when manually converting all needed arrays to their associated device pointers, removing the need for the OpenACC runtime to perform a pointer translation for each array. By manually handling the data in this way, the overhead seen when launching a single *Integrate* kernel is reduced by  $\sim 80\%$ .

Even with these improvements *Integrate* still sees 25% of its total time taken by launch overhead and also experiences low GPU occupancy. Two possible optimizations paths are present for NVIDIA GPUs for handling this situation. Firstly, we may consolidate multiple launches of the *Integrate* kernel together, with the caveat that we need a grid-level synchronization after each inner 2D slice. In theory, in OpenACC this would potentially be a situation with the outer-most loop marked as a “sequential” loop, signifying that all gangs should synchronize after each iteration of that loop, such as the code prototype shown in Figure 3.2.

However, this code does not produce the desired result. Instead, each gang will execute the loop with no synchronization meaning that there is no guarantee for correct functionality. We want to prepare the GPU resources, i.e the grid of thread blocks,

```

#pragma acc parallel
{
  #pragma acc loop seq
  for(int y = 1; y < Ny; y++) {
    #pragma acc loop gang
    for(int x = 1; x < Nx; x++) {
      #pragma acc loop vector
      for(int z = 1; z < Nz; z++) {
        <Integration Calculation>
      }
    }
  }
}

```

Figure 3.2: A possible implementation for thread block synchronization in OpenACC.

with the invocation of the *parallel* directive. However, we want all thread blocks to obey the serial nature of the first loop marked by the *seq* directive.

A second method would be to utilize CUDA graphs. CUDA graphs allows the programmer to define a dependency graph where each node would be some CUDA operation such as a kernel launch or data transfer. We can create a graph that contains every launch we will need to do of the *Integrate* kernel which streamlines the kernel launch process and reduces overhead. However, there will be a cost of setting up the graph, but this cost is only accrued once at the start of a simulation. Since this utilizes a feature specific to NVIDIA GPUs it is likely unfit for CUDA graphs to be supported in the generalized OpenACC model, but compiler-specific features could be created for those that support OpenACC on NVIDIA GPUs.

### 3.3.2 Integration with Grid-Level Synchronization

The key focus of our first approach is that we want all thread blocks to synchronize after each step of the computation. With this mechanism only one step of the sequential loop is completed at a time, without the need of allocating and freeing GPU resources between each kernel invocation. We use a feature within CUDA called Cooperative Groups that allows for synchronizing across thread blocks. For grid-level synchronization to work we must allocate a number of thread blocks that will all simultaneously fit within

GPU hardware constraints (i.e the number of thread blocks is equal to the number of streaming multiprocessors), as well as use the `cudaLaunchCooperativeKernel` function in the CUDA kernel launching API. This consolidates the kernel launch overhead into a single launch at the cost of additional grid-level synchronization. Figure 3.3 shows this variation of the *Integrate* kernel.

```
namespace cg = cooperative_groups;

auto grid = cg::this_grid();

for(int i = 0; i < bound0; i++) {
    for(int j = blockIdx.x;
        j < bound1;
        j += gridDim.x)
    {
        for(int k = threadIdx.x;
            k < bound2;
            k += blockDim.x)
        {
            <Integration Calculation>
        }
    }
    grid.sync();
}
```

Figure 3.3: Integrate kernel with grid-level synchronization using CUDA Cooperative Groups.

### 3.3.3 Integration with CUDA Graphs

We also explore an approach to optimize the launch overhead by utilizing the CUDA Graphs feature. Our graphs are simple, linear graphs that includes a node for each kernel launch while obeying the serial dependency. Since the dependent dimension as well as the direction depends on the radiation ray, we create a separate graph for each of the 24 rays. This initialization is done once at the beginning of the simulation when RT is first called. The key benefit of this approach is to preemptively capture information about upcoming kernel launches, allowing for the enqueueing of *Integrate* kernels to happen with much lower overhead. Fig 3.4 shows this variation of *Integrate*.

```

cudaGraph_t graph[24];
cudaGraphExec_t instance[24];
void integrate_cuda_graph()
{
    if(!graphInitialized) {
        for(<Each ray configuration>) {
            cudaStream_t stream;
            cudaStreamCreate(&stream);
            cudaStreamBeginCapture(
                stream,
                cudaStreamCaptureModeGlobal
            );
            for(int b1 = 0; b1 < bound1; b1++){
                integrate_kernel<<<...>>>(args...);
            }

            cudaStreamEndCapture(
                stream, &(graph[ray]));
            cudaGraphInstantiate(
                &(instance[ray]), graph[ray],
                NULL, NULL, 0);
        }
    }

    cudaGraphLaunch(instance[ray], 0);
    cudaStreamSynchronize(0);
}

```

Figure 3.4: Integrate using CUDA Graphs to manage launching of individual 2D slice kernels.

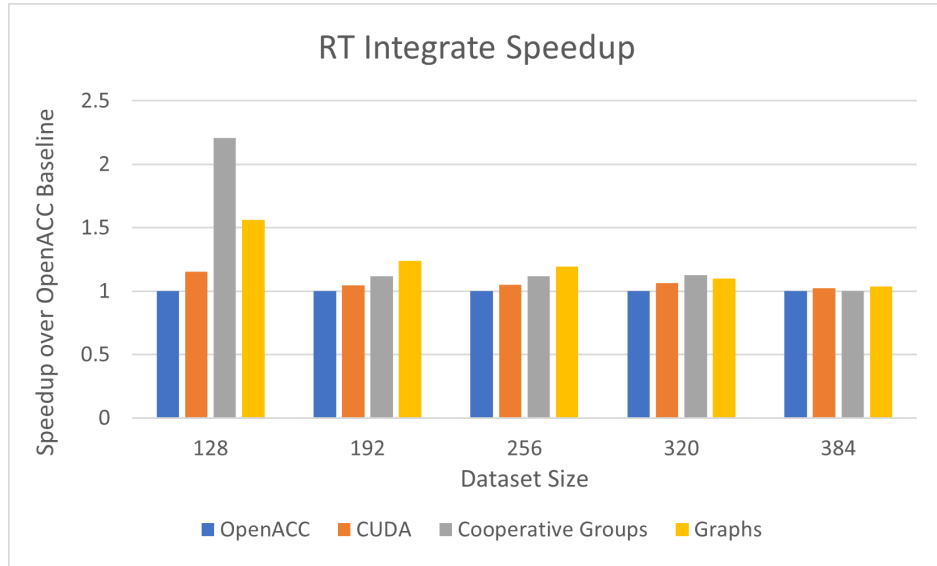


Figure 3.5: Relative speedup of the different variations of *Integrate* over MURaM’s OpenACC baseline implementation.

Code	Time per slice	Overhead	Total
OpenACC	$5.5\mu s$	$0.9\mu s$	$750\mu s$
CUDA	$4.5\mu s$	$0.9\mu s$	$665\mu s$
CUDA CG	$2.45\mu s$	N/A	$310\mu s$
CUDA Graphs	$4.7\mu s$	$0.7\mu s$	$472\mu s$

Figure 3.6: Breakdown of time spent in computation and overhead for the different *Integrate* variations, captured using the NSight Systems profiler. “Time per slice” in the computation time of a single 2D parallel slice of *Integrate*, “Overhead” is the time taken from the end of one kernel to the start of the next, and “Total” is the time taken for the entire integration, i.e over the entire 3D domain. All times are using the  $128^3$  dataset.

### 3.3.4 Integration Performance

Figure 3.5 shows the relative speedup difference of several *Integrate* implementations over MURaM’s OpenACC baseline. *OpenACC* uses only OpenACC, and is what is currently implemented in the full MURaM code. We have implemented several optimizations to reduce the OpenACC specific overhead, such as storing relevant data directly as device pointers to avoid needing any pointer translation.

The *CUDA* variation is a comparable implementation to the OpenACC version written in the CUDA programming language. Due to the overall simplicity of the *Integrate* kernel we do not expect that our CUDA code is inherently better optimized

by virtue of being “hand-written.” In all cases the CUDA version is slightly, and consistently, better than the OpenACC version. The most observable difference we see between these two versions is the number of registers used per thread. The OpenACC version requires 56 registers-per-thread, while the CUDA version uses only 26 register-per-thread. This causes the observed runtime GPU occupancy to be slightly better for the CUDA code, which is even more important considering that *Integrate* is a latency-bound kernel. We do not see this as a difference in the quality of the two codes, but rather inherent performance limiters we accrue when using a directive-based model such as OpenACC. We observe that the launch overhead between the OpenACC and CUDA implementation is similar, but the actual computational kernel is 10% faster in CUDA.

The *Cooperative Groups* version is written in CUDA using CUDA Cooperative Groups to facilitate multi-block, grid-level synchronization. This reduces the multiple kernel launches used in both the OpenACC and CUDA versions into a single kernel launch by launching only the exact number of thread blocks that fit simultaneously onto the GPU and obeying the sequential outer loop by using the grid-level synchronization so that all blocks will only move onto the next iteration once all have completed the current iteration. We observe higher register usage than the previous CUDA variation at 40 registers-per-thread. We see better relative performance of this version for smaller data sizes, at 2.2x faster than the OpenACC baseline for the  $128^3$  dataset, and 1.1x faster for the  $256^3$  dataset. Once we increase the dataset up to  $384^3$  we see the Cooperative Groups version is nearly identical to the OpenACC, and slightly slower than the other two CUDA implementations.

The key issue we are trying to solve is the kernel launch overhead. As the dataset gets larger, that overhead becomes less impactful. There would eventually be a data size where the kernel time is sufficiently large enough that the launch overhead would no longer significantly affect performance, since the next kernel could be prepared before the current kernel finishes. Due to this the *Cooperative Groups* variation becomes less

impactful as the dataset size increase, but could be a very important performance increase in strong-scaling scenarios where the grid points per GPU becomes smaller.

The *CUDA Graphs* variation is also written entirely in CUDA. For the results in Figure 3.5 we do not measure the overhead of one-time initialization of the graph. The underlying kernel used is completely identical to the kernel used in the *CUDA* variation, but uses the graph to facilitate kernel launching. Similar to the *Cooperative Groups* variation, *CUDA Graphs* sees less performance increase over the baseline as the dataset gets larger, and that launch overhead becomes less impactful. *CUDA Graphs* sees 1.2x speedup over the baseline for the  $256^3$  dataset. A 20% performance increase on the *Integrate* kernel would be extremely impactful for the overall MURaM performance, and especially for high-resolution simulations that depend on multi-band RT where *Integrate* is the most time-consuming kernel of the entire simulation.

Figure 3.6 shows the breakdown of timing for different aspects of these variations. We observe the  $128^3$  dataset since it best highlights the problem being solved and the benefit of these solutions. Since *CUDA Cooperative Groups* only produces a single kernel launch we can not gauge the time taken between the parallel 2D slices (i.e the time to synchronize), and instead divide the total time taken by the total number of slices in the dataset, making a direct comparison slightly difficult. For the other variations we observed the time taken per slice as seen through the NSight Systems profiler and averaged across all slices.

The two key takeaways we gain from this experimentation is as follows: (1) with the optimizations we have added to *Integrate* we have achieved close performance to an identical kernel written in CUDA. This is a far improvement from early, naive implementations of *Integrate* that introduced large amounts of overhead from OpenACC’s runtime library. (2) we can achieve a very significant performance increase of MURaM’s most important kernel by utilizing techniques found in CUDA when running on smaller data sets. This creates interesting opportunities for improved strong scaling of MURaM, and is discussed alongside other scaling results in Section 3.5.

### 3.4 Scaling Results

MURaM provides several use-cases in solar research and for this work we focus on two different applications:

(1) For high resolution simulations of the upper convection zone and photosphere (grid spacings comparable or smaller than 10 km) detailed RT is critical. It is computationally not feasible to capture the full frequency dependence of opacities in the solar atmosphere, which would require hundreds of thousands of individual frequency points. Following [86] spectral lines are combined in bands according to their strength (and sometimes also frequency) and average opacities are computed for each band (typically 4-12). However, even with 4-12 bands RT is still computationally expensive and accounts for 60-80% for the computing time. High resolution simulations of the photosphere are needed for in-depth comparisons with high resolution spectropolarimetric observations, such as those provided by the Daniel K. Inouye Solar Telescope (DKIST), to support a multitude of science investigations [94].

(2) For larger-scale simulations that aim at capturing entire solar active regions (scales on the order of 100 Mm) and extend from the upper convection zone into the lower solar corona numerical setups use lower resolution (grid spacings comparable to 100 km) and a detailed RT is less important. The cost of RT can be reduced by using only a single band (gray RT) and due to the small simulation time steps in setups that include the solar corona, RT does not have to be called at every simulation time step. In these setups RT accounts for less than 10% of the computing time and code scaling behavior is more dominated by the scaling of the MHD routines. These types of computational setups allow to simulate solar active regions over time-scales of hours to days and focus on studying processes related to the storage and release of energy in the solar corona, ultimately leading to the flares and coronal mass ejections [36, 35, 100]. More detailed studies of specific active regions will be enabled in the future through data-driven simulations [32].

Our results will utilize both of these use-cases to observe performance possibilities for the multiple uses of MURaM.

### 3.4.1 Experimental Setup: Raven and Derecho

Raven is an HPC system in the Max Planck Computing and Data Facility, installed in 2021, with a 23.4 PFlop/s theoretical peak performance. It consists of 1,592 compute nodes with Intel Xeon IceLake-SP processors with 72 cores and 256GB RAM per node. We used up to 128 out of 192 NVIDIA A100 GPU nodes (40GB HBM2 each) with a total of 512 GPUs for our weak and strong scaling runs. The GPU nodes are connected with 200 Gbps Mellanox HDR InfiniBand interconnects. For more, refer to [11]. We used NVHPC SDK 21.3, CUDA 11.4, OMPI 4.1.5 and FFTW 3.3.10, for our runs. Simulations were performed with more modern compilers, however a performance degradation was noticed, as discussed in Section 3.2.6

NCAR’s Derecho HPE Cray EX cluster, installed in 2023, is a 19.87-petaflops system. It consists of 2,488 compute nodes with 128 AMD Milan cores per node. We used up to 80 NVIDIA A100 GPU nodes (40GB HBM2 each) out of 82 for our runs with a total of 320 A100 GPUs. The GPU nodes have 600 GB/s NVIDIA NVLink GPU interconnect. For more, refer to [14]. We used NVHPC SDK 23.1, CUDA 11.7.1, Cray-MPICH 8.1.25, FFTW 3.3.10 for our runs.

All runs are using the heFFTe library on the CPU for the FFT calculation, as described in 3.2.1. For additional details of all of the experiments shown here, such as full performance breakdown and GPU layouts, please refer to raw data in Zenodo [13].

### 3.4.2 Weak Scaling

Weak scaling is the measure of performance when the number of processors increases at the same rate as the amount of data. This means that each processor keeps a constant amount of local data even though the total data size is increasing. Ideal weak scaling is when the time taken to compute the problem does not increase as the data size increases. Here we measure the weak scaling of MURaM on an increasing number of GPUs from 4 to 500. The run is a high resolution simulation of the upper convection zone and photosphere using 4-band RT every timestep. Figure 3.7 shows the total time per simulation timestep, as well as several individual points for a few

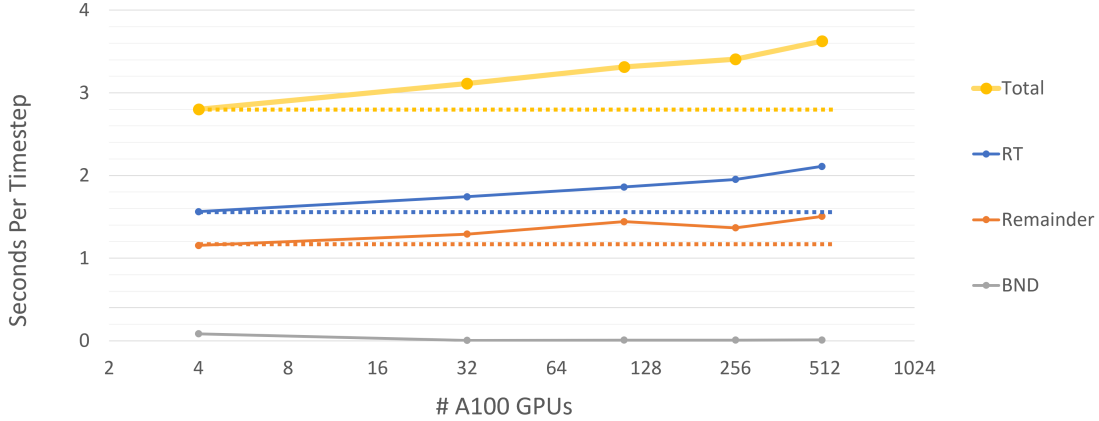


Figure 3.7: Weak scaling on Raven of a high-resolution simulation ranging from 4 to 500 A100 GPUs. Dotted lines represent ideal scaling.

different routines that are relevant to scaling. The data uses  $360^3$  (46,656,000) grid points per GPU.

The total time variance between 4 and 500 GPUs is 0.82 seconds, which is a 29% increase. RT accounts for 0.54 seconds of the total and sees a 35% increase overall. BND is the calculation of the boundary region and contains all of the FFT portions of the code. The “remainder” category accounts for everything besides RT and BND, and contributes to the remaining time increase. However, there is no single routine within the remainder that individually scales poorly, rather we see a minor, but steady increase in a few kernels as the number of GPUs increase. Additionally, BND accounts for a very small portion of the total runtime and does not see a notable increase from scaling. This is a significant improvement from previous versions of MURaM that used FFTW, which saw BND as the second worst scaling routine, behind RT.

### 3.4.3 Strong Scaling

Strong scaling is the performance measure of increasing the number of compute resources while keeping a consistent dataset size. The ideal scaling would be completely linear, i.e doubling the number of compute resources results in double the throughput. However, it is often the case in parallel programming that due to various factors a perfect strong scaling is improbable to achieve.

We have performed two different scaling experiments, each focusing on a different use-case of MURaM. Firstly we observe a high resolution simulation of the upper convection zone and photosphere, which uses 4-band radiation transport every timestep. These experiments were performed on the Raven supercomputer.

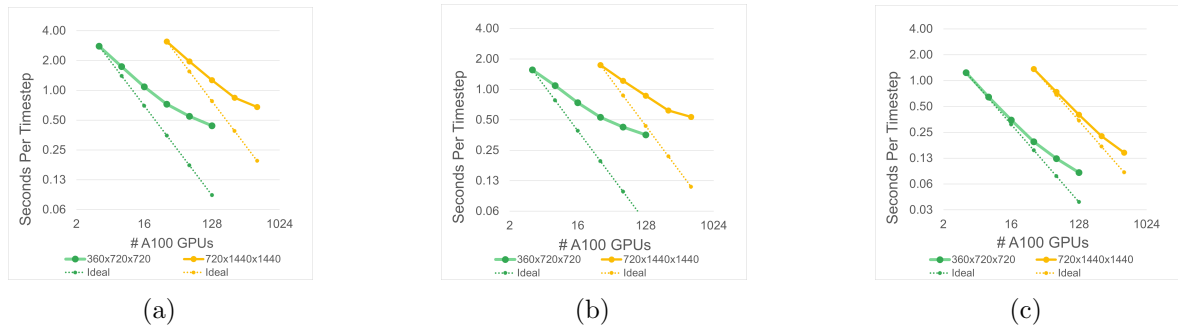


Figure 3.8: GPU strong scaling for a high-resolution simulation, 4-band RT, from 4 to 512 A100 GPUs on the Raven system. (a) shows total time per timestep, (b) show time of RT per timestep, (c) shows total time - RT per timestep.

Fig 3.8 shows these strong scaling results and has been divided into three plots to isolate the performance of radiation transport from the MHD portion of the code. As the number of GPUs increase, the number of grid points per GPU decreases, ranging from 46,656,000 to 1,458,000. GPUs typically perform best when sufficient data parallelism is available, and is especially important for MURaM since it is largely a memory latency-bound code. From this we expect to lose some efficiency as the grid points per GPU decreases.

We observe very promising scaling in the MHD portion of the code. The purely computational kernels scale near-linearly, and those with MPI communication are only affected somewhat, as our total MPI ranks is still relatively low. The scaling of radiation transport is not nearly as good, as expected. RT has significant communication, needing to communicate potentially dozens of times per timestep. Additionally, we observe several inefficiencies in various kernels in RT that are further exasperated by the reduced number of grid points.

We have also performed strong scaling experiments using a larger-scale simulation that aims at capturing the entire solar active region (scales on the order of 100 Mm) and

extends from the upper convection zone into the lower solar corona. This experiment puts a far lower focus on RT, only running the RT solver every fourth iteration and single band. This causes RT to run roughly 16x less than in the previous experiment. These results are captured from the Derecho compute cluster ranging from 48 to 320 A100 GPUs. We have also run this same experiment using CPU-only nodes to observe the scaling of MURaM for CPUs. This ranges from 32 to 512 AMD CPU nodes, for a total of 4,096 to 65,536 cores.

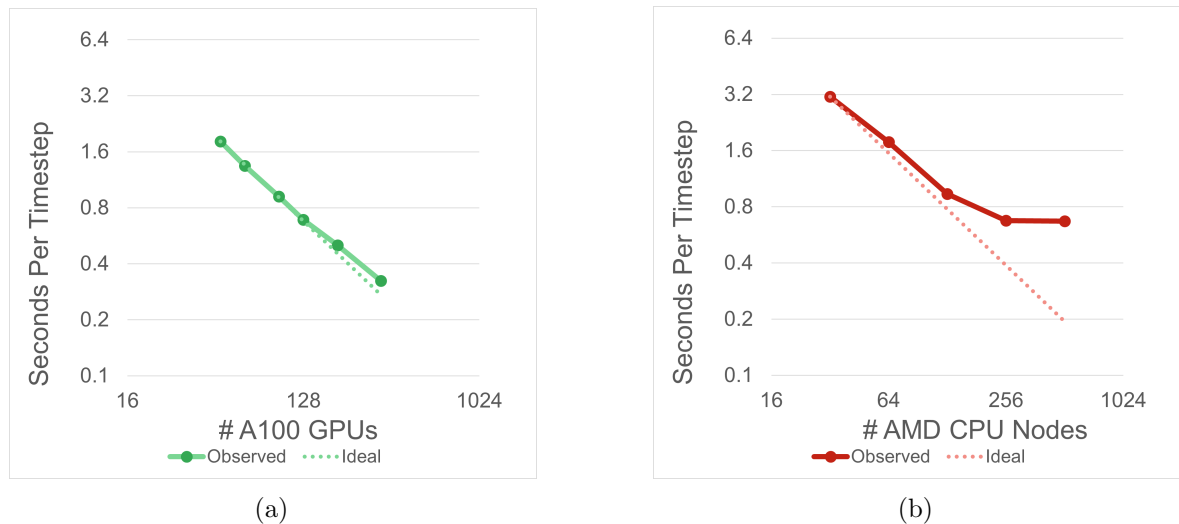


Figure 3.9: GPU strong scaling for a lower-resolution, larger-scale simulation from (a) 48 to 320 A100 GPUs and (b) 4,096 to 65,536 AMD CPU cores on the Derecho system.

Fig 3.9 shows these strong scaling results. In these results we observe better strong scaling than the MHD-only scaling from the previous experiment, which likely originates from the faster network connections on the Derecho system. Overall, this reinforces that the MHD portion of the code strong scales relatively well, and with less focus on RT we approach ideal scaling. We also begin to hit hard limits of CPU scalability. No matter how many CPU cores we use in the simulation we can not reach the GPU performance capability, while the GPU seems to be able to scale even further. While it is difficult to draw direct comparisons between CPU vs. GPU, we observe in terms of throughput for these runs that one A100 GPUs equates roughly to  $\tilde{90}$ -100 AMD CPU cores.

### 3.5 Analysis and Future Work

From all the above discussions and results, we are able to draw several analyses. The MHD portion of MURaM strong-scales fairly well. For simulations with a lower focus on RT it is feasible to reach higher throughput milestones by utilizing additional hardware. So far from the available GPUs we have scaled to we do not believe we have reached the point yet where MHD performances falls off. Additionally, with the improvements to the boundary routines using the heFFTe library, we see much better weak scaling than in previous versions of MURaM. There appear to be minor improvements that can be made to the MHD portion of the code, but the most significant impact to weak scaling is again RT.

RT falls off in our strong scaling experiments fairly quickly, making it somewhat infeasible to further improve RT’s performance with additional hardware resources. Introducing more GPUs compounds communication costs and reduces available data parallelism making individual kernels, such as *Integrate*, less efficient. If we wish to further improve the scalability of RT we must look towards code refactoring or algorithmic changes.

From our experiments in Section 3.3, we see potential in improving the strong scaling of RT by utilizing some GPU-specific features of CUDA. The two key components of RT that need to be improved for overall scaling is the *Integrate* routine and the communication scheme. Both the *Cooperative Groups* and *CUDA Graphs* version of *Integrate* see large improvements over the purely OpenACC baseline for smaller per-GPU dataset sizes, such as those seen in our strong scaling experiments. In the best case scenario for a  $128^3$  dataset we see a 120% improvement in our best prototype. Even in larger datasets, such as the  $256^3$  dataset, we see a 20% improvement, which is still extremely impactful in the case of high-resolution simulations with multi-band RT where the runtime of the *Integrate* kernel is very dominant of the total simulation time.

As a near term future work for this project, we aim to further optimize the codes shown in Section 3.3 and add them as optional inclusions to the full MURaM code. Then it would be interesting to recreate full strong scaling for these high-resolution,

RT-based simulations to see what sort of improvements have been made for the entire simulation time. Outside of the changes described in this chapter, it is likely worthwhile to experiment with algorithmic changes to improve both available data parallelism and reduced communication required.

## Chapter 4

### AUTOMATIC PARALLELIZATION OF SERIAL CODES WITH OPENACC ANNOTATIONS

This chapter will cover the work done during my preliminary exam using Clang and OpenACC to use both static and dynamic analysis to automatically generate OpenACC directives for parallelizing serial codes. This work highlights the feasibility of a framework for automatic parallelization and optimization of a code using a similar programming cycle as HPC application developers, and the difficulty of bridging the gap between high-level languages and low-level hardware features when using parallel programming directives.

#### 4.1 Related work

DawnCC [84] is a stand-alone tool for auto-annotating C code with OpenACC or OpenMP 4.0 directives. DawnCC uses LLVM/Clang [76] to convert a C source code to LLVM IR to take advantage of several LLVM-based static analyses. The authors offer several important and common factors that their tool overcomes in order to produce annotated parallel code.

Firstly, in order to properly manage data communication between the host processor and offloading device the lower and upper boundaries of dynamic memory must be known. LLVM provides a utility for analyzing LLVM IR called *symbolic range analysis* to estimate these boundaries, and is the primary reason that LLVM is being used for their work. By understanding the boundaries of used data in each kernel, DawnCC is able to generate proper data communication directives, and to apply several data communication optimizations across kernels reusing the same data.

Next, the authors address C/C++ pointer aliasing. This is when two pointers reference overlapping memory addresses. The authors use a technique called *pointer restrictification* to incorporate a pointer aliasing check before each computational kernel into the source code. In OpenACC, this utilizes an *if directive* to decide at runtime if a specific kernel should be run on the host or device. Then to parallelize the kernel, the OpenACC *kernels directive* is added, which asks the compiler to analyze a region of code and attempt to parallelize it. The *independent clause* is also added, which asserts to the compiler that a given code region is safe to parallelize, and to ignore any warnings including (but not limited to) pointer aliasing.

Lastly, the authors have developed a new technique they have called *copy fusion*. This is a method to use a dependency graph to eliminate redundant memory transfers. It is a common problem when doing GPU programming that host and device memory pools have to constantly be considered. To achieve optimal performance, the total number of transfers will need to be minimized, or hidden by overlapping computation and communication. The primary goal of copy fusion is to combine data regions for various kernels where their data use overlaps.

There are a few key limitations to this tool that significantly reduce its usefulness for real-world applications. DawnCC works to identify loop nests within the code and annotate them with some basic directives. For OpenACC, this means adding the OpenACC *kernels directive* to the outer-most loop. The *kernels directive* is very unique as it signals to the compiler that the given loop should be analyzed with an attempt to automatically parallelize it. In some simple cases, the compiler is able to successfully parallelize the loop, and may sometimes achieve higher performance than what a typical programmer could do manually. However, for the vast majority of codes, the *kernels directive* is significantly less effective.

While the authors solution does address the problem of pointer aliasing, it is far too aggressive. Firstly, falling back to running on the CPU if pointer aliasing is detected is inefficient. There would rarely be a situation where this would be a good compromise, it is likely that an expert programmer would have to step in and fix the

problem manually. Second, there are many reasons why some codes should not be parallelized, including pointer aliasing. Since the kernels are not being checked for other potential problems, overriding the compilers decision with the independent clause can produce catastrophic results. If this tool were to produce faulty code due to this, it would likely take a significant amount of effort from an expert programmer to debug it. It is also possible that even if pointer aliasing exists, a code could still be parallelized without problem, depending on the code.

Additionally, an issue that the authors do not address; if pointer aliasing does occur and causes a kernel to default to running on the host processor, then this can easily cause out-of-sync memory pools. For example, take a code that runs nearly all of its computation on the GPU, meaning that the data stored in GPU memory will contain all the meaningful results of their computations up to that point. Then, if the next kernel has to default to running on the host processor, then the host will not have access to any of those results unless explicitly copied from the device. None of the code examples the authors use for describing pointer restrictification contain a memory fail-safe for if this were to occur.

Since DawnCC does not do any sort of performance analysis, all loops are selected for parallelization. In real-world applications, it is unlikely that every loop should be run in parallel. For example, in the MURaM code, there is a very large loop nest that handle propagating radiation intensity through a three-dimension space. The radiation moves through the eight corner of the space, each of these with three different angles. That means that there is a  $2 \times 2 \times 2 \times 3$  loop nest, within it containing many different complex loops. If the authors solution were to be applied to a code like this then it is very likely to work, or perform well what-so-ever.

Lastly, DawnCC only annotates the outer-most loop in each detected kernel. Speaking from experience with OpenACC, compilers have a very difficult time parallelizing large loop nests with limited information. By only providing a directive to the outer-most loop will likely give poor performance or fail for large loop nests. One quirk with OpenACC is that OpenACC defines three usable levels of parallelism, but only

uses two by default. Any loop nests that are larger than two loops will likely have the inner-most loop running entirely sequentially across GPU threads. DawnCC may be useful for creating an “initial parallelization” of a source code, but will likely need to be heavily optimized by an expert programmer to achieve reasonable performance.

Kernel Tuner [119] is a python-based tool for auto-tuning CUDA and OpenCL code. The tool allows programmers to provide the source code of a GPU kernel and a set of tunable parameters, such as kernel dimensions, loop tiling and loop unrolling. The parameters will be edited and the kernel re-ran until an optimal solution is found. Several different techniques are employed to select values for each parameter, with the simplest and most time-consuming being brute force, but is guaranteed to eventually find the optimal solution.

The author concludes that brute force reliably performs the best, but take 10-100x more time to run than other strategies. Basin hopping and differential evolution are the next best two, however they do not always reliably find the optimal solution. In the authors test cases, the average performance of these two approaches were 25% worse than brute force.

The author also mentions using tools such as Kernel Tuner for test-driven software development. A programmer can create several test cases for each important kernel in their code and use Kernel Tuner to automatically run the tests, verify the kernel outputs are correct, and tune the kernel. Maintaining this process throughout the code development process may help with porting kernels to accelerators, and is an automated version of good coding practice.

The tool will also automatically handle data allocation and deallocation. However, it is left very vague, and not specified how this is actually resolved. It is likely that the programmer will also need to provide some sort of test configuration along with the kernel code otherwise the tool would not know what size the arrays should be, or what data they should be populated with. A code such as a dense matrix multiplication would surely be problematic if arrays containing valid indices were not provided. This

is important for AutoACC as the current auto-tuner implementation requires the programmer to provide at least one test configuration to run.

The tool requires the programmer to identify the tunable parameters in the kernel. This means that the tool is only optimizing parameters that are consistently present in codes, such as thread block dimensions, loop tiling, and loop unrolling. AutoACC is very unique in that it is generating the accelerated code, thus it is also generating the tunable parameters. However, one hope is that AutoACC will be able to identify different tunable parameters that most auto-tuners, including Kernel Tuner, would not usually operate on. Parameters such as the ordering of loops, ordering of memory read/writes, register usage and shared memory usage. Optimizing directive-based codes involves editing the source code just as much as editing the directives themselves.

## 4.2 Intermediate Representation with Clang’s Abstract Syntax Tree

There are three key types of AST nodes: declarations (*Decl*), statements (*Stmt*) and expressions (*Expr*). Declarations are either function or variable declarations (*FunctionDecl*, *VarDecl*). Statements cover every other node type as in C/C++ every expression, such as function calls (*CallExpr*), can also be used as a statement. The AST is an acyclic graph of nodes where the root node is a declaration (*TopDecl*) that contains all other declarations. An example of an AST is shown in figure 4.1.

AutoACC is built on the assumption any code the user is compiling is valid and compiles without error on their target OpenACC/OpenMP compiler. This means that AutoACC does not do rigorous code validation such as type checking. However, since the AST is mutable there must be strict requirements that the AST represents a valid C/C++ code. The constructors for each node type extensively checks that all children nodes are reasonable types. All children are stored as generic node types, but are only accessible through class getter methods that cast them to the correct value. Each node also keeps track of its parent node, as this is necessary to perform node replacement.

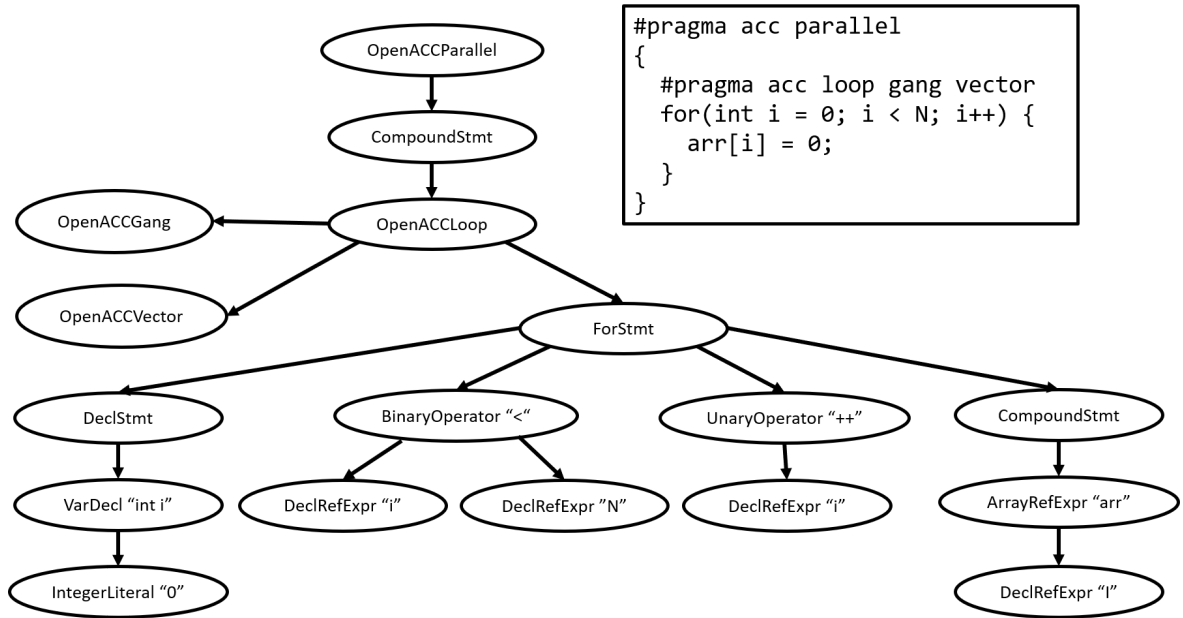


Figure 4.1: Visual representation of an AST for an OpenACC parallelized for loop.

### 4.3 Implementation of Automatic Annotation

OpenACC is designed to work for heterogenous computing machines; these are machines that have a host and one or more attached accelerators. The host is typically a general purpose processor, and the accelerators can be a variety of architectures, but commonly used are manycore CPUs, GPUs, and FPGAs. OpenACC provides the *parallel* and *kernels* directives for specifying regions of code that should be built and executed on a given accelerator. These regions typically contain for loops marked with the *loop* directive to specify work sharing.

**OpenACC Execution Model:** The *parallel* and *kernels* directives achieve the same goal but have drastically different implementations. They both are used to parallelize code, but the kernels directive requires significantly more analysis and work from the compiler. Regions of code marked with the kernels directive will be analyzed by the compiler, and the compiler can make an attempt to assign work sharing and specific optimizations. This means in a perfect scenario the programmer only needs to mark the parts of the code they wish to have parallelized and allow the compiler to do all of the heavy lifting.

Unfortunately, the *kernels* directive has several inherent flaws particularly for C/C++ code and is difficult to implement within compilers. In order for OpenACC to generate proper device code, it must know the size of the various arrays used in the kernel to ensure that equivalent device arrays exist. Also, the kernels directive is very sensitive to potential problem that inhibit parallel behaviors. The most common example in C/C++ is the existence of pointer aliasing. When a compiler detects potential aliased pointers (which is any kernel that contains more than one pointer) the compiler will refuse to parallelize. This decision can be overridden by the programmer if they decide that the loop is truly safe to parallelize.

**OpenACC Memory Model:** OpenACC's memory model is centered around linking and synchronizing between host and device memory with memory transferring runtime library calls. Most of this is done behind-the-scenes by the OpenACC runtime, but the programmer will still need to specify the sizes of arrays used in the various kernels, as well as transferring data between host and device as needed. The PGI compiler also supports the use of CUDA managed memory. This is a feature that exists for NVIDIA GPUs where both host and device memory exist within a virtual memory space. Data requests will go through the virtual memory, and data will be moved between host and device as needed. However, if using other compilers, or other accelerators, this feature is not supported.

**Three Levels of Parallelism:** OpenACC provides three levels of parallelism called *gang*, *worker* and *vector* threads. Gang can be thought of as coarse-grained parallelism, vector as fine-grained parallelism, and worker in between. Each gang contains one or more workers, each worker contains one or more vector threads. When a parallel or kernels region is encountered, a group of gangs, workers and vector threads are created on the device. Each gang will execute redundantly using a single worker and single vector thread until a loop is encountered and work is spread across the threads. Usually for work-sharing to occur, a for loop must be marked with the loop directive, however, when using the kernels directive the compiler may be able to auto-detect

loops for work sharing. The gang, worker and vector mechanism in OpenACC has very interesting implications for GPUs.

**Collapse:** OpenACC also provides a few clauses for the loop directive to restructure loop nests and create more optimal code. The *collapse* clause combines multiple tightly nested loops to make a larger loop and potentially increase the exposed parallelism. The *tile* clause portions the nested loops into tiles, potentially increasing data locality. These along with gang, worker and vector are the primary ways that a programmer to optimize their loops in OpenACC.

**Cache Directive:** The last OpenACC construct worth mentioning is the *cache* directive. This directive specifies arrays that should use some sort of cached memory. For GPUs, this will used shared memory, which significantly closer to the threads and significantly faster. Some compilers can also try to use shared memory optimizations in other places, but isn't entirely reliable without using the cache directive.

AutoACC is a source-to-source compiler for annotating C, C++ and Fortran code with OpenACC or OpenMP directives for offloading. AutoACC contains a framework for generating an abstract syntax tree (AST) that is modeled after the source language, and generating same-language source code from the AST. Different modules can exist on top of this framework such as annotation, optimization, correctness checking and auto-tuning. These modules are split into two separate parts called AutoACC Serial (AutoACC<sub>s</sub>) and AutoACC Parallel (AutoACC<sub>p</sub>).

Figure 4.3 shows the overall workflow of AutoACC. The tool has two separate entry points; at the beginning of AutoACC<sub>s</sub> and AutoACC<sub>p</sub>. If entering straight into AutoACC<sub>p</sub>, then it is assumed that the code has been already annotated with custom AutoACC directives by the programmer. Within AutoACC<sub>s</sub> and AutoACC<sub>p</sub> are different modules that the AST can be run through. The input to every module is an AutoACC AST, and the output should also be an AutoACC AST.

AutoACC<sub>s</sub> focuses on analyzing the serial behavior of the code. The primary goal is to understand overall code structure, and generate several “facts” about how the code runs in order to provide additional information to AutoACC<sub>p</sub>. These facts

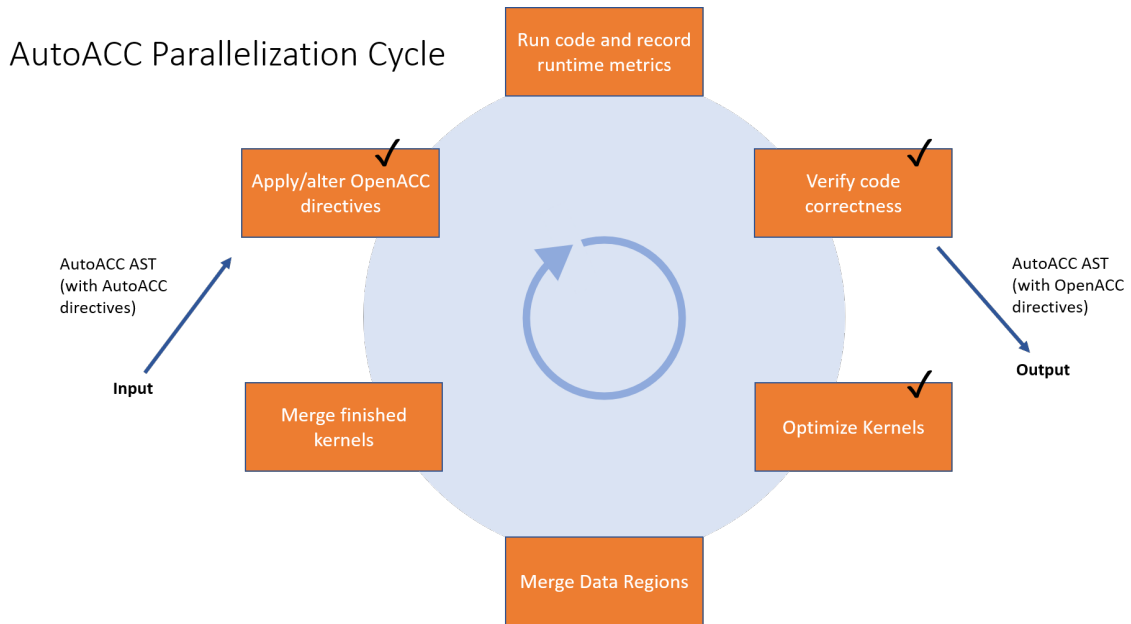


Figure 4.2: Looping development cycle of incrementally parallelizing sequential codes with directives.

include the sizes of loops, size of allocated memory, the striding of memory reads and writes and race conditions. These are all details that  $\text{AutoACC}_p$  will need in order to generate correct parallel code.

The OpenACC organization provides a best practices programming guide for OpenACC [75]. This guide highlights a few ideas that are important to how AutoACC generates OpenACC kernels:

- Compilers can make limited optimizations with limited information available. It is best to be verbose and include as much specifics as possible.
- Often times, code refactoring for better GPU performance offers better CPU performance as well.
- Leaving loops without a loop directive may cause the compiler to run that loop serially. It is best that all loops in a loop nest have some sort of loop directive so the compiler knows they are safe to parallelize.

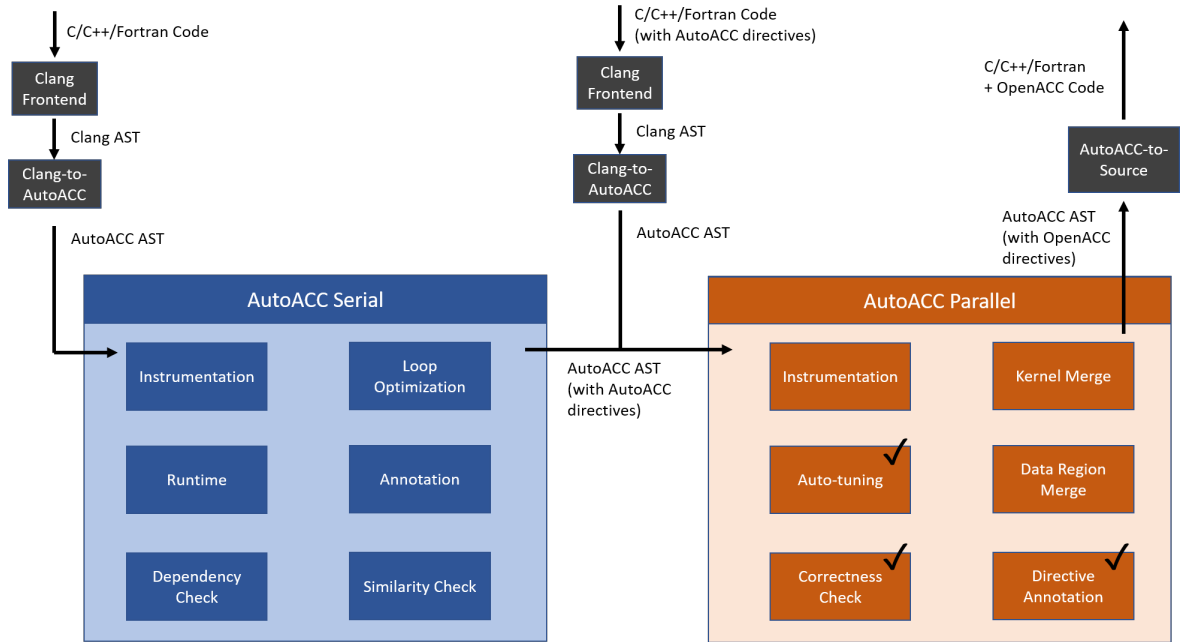


Figure 4.3: Workflow of the AutoACC tool from input sequential code to automatically generated parallel code.

- Parallelizing inefficient kernels is still likely better than doing a memory copy. It may be best to do computation on the GPU whenever possible, even if a single kernel will perform relatively poorly.
- When applying an optimization that is targeted for a specific architecture, it may be test to use the *device\_type* clause so those optimizations are not applied to other architectures.

AutoACC<sub>p</sub> implements a parallel program development cycle pictured in figure 4.2. This loop replicates the coding practice done by parallel software developers in order to generate a parallel code and optimize it for a given architecture. The first step is to use the information gathered in AutoACC<sub>s</sub> or from data provided by the programmer in the form of AutoACC directives to generate parallel code. Currently, the focus is on OpenACC and OpenMP code generation, however any directive-based model could be viable if correct modules are made. Also, a potential future direction could be generating other languages such as CUDA or HIP.

The second and third step is to run the new parallel code and validate that it gives correct output. Correctness is tricky to define, and many different validations techniques will need to be included. One technique that is commonly used for GPU program porting, and is implemented by other applications such as Kernel Tuner [119], is to check results on a per-kernel level. For each parallel kernel, check if all outputs of that kernel are within some accuracy threshold of the original serial code. There will also be support for users to provide their own test cases that will decide correctness or failure.

The fourth step optimizes the generated parallel code. This is done using auto-tuning techniques to tune kernel parameters to achieve optimal performance on a given architecture. There exists many different techniques for auto-tuning [119], brute force tends to give the best results. However, the time taken to do brute force auto-tuning will likely be too large for the tool to utilize in a reasonable amount of time. For this reason, AutoACC will likely have to use more than one auto-tuning technique, and may also allow the user to set a time limit on how long auto-tuning should go on for.

The fifth and sixth step are about combining the optimized code. In DawnCC [84], using algorithms to optimize data communication throughout a large source code is overall very important for total code performance, rather than just focusing on kernel performance. Also, sometimes the computation of multiple kernels can be combined to create larger kernels. Making a kernel larger often yields better results, until hardware resources are completely saturated. For this reason, step five and six will be very important for overall code performance.

The work described in this report is of early stages of the AutoACC tool. This will include the construction of the AutoACC AST, the custom AutoACC directives, OpenACC code generation and kernel auto-tuning. Currently, only a subset of the C language is supported, and only OpenACC code can be generated.

In OpenACC, the outer-most for loop annotated with directives is the entry point of the computation kernel. This loop technically does not need to be marked with any work-sharing directives, but results will end up unexpected. In MURaM, for

the RTS integration loop I have tried avoiding kernel launch overhead by making the first loop in the kernel sequential. However, the language here is misleading. Having a loop within an OpenACC kernel marked as sequential actually means that the loop will not have any work sharing. In the MURaM example, marking the outer-loop as sequential only made all gangs redundantly run all loop iterations simultaneously, instead of blocking like we may want.

Here I propose a kernel generation algorithm. This algorithm uses a depth first search approach. Nested loops in the AST are traversed until the first leaf node is found. In our loop nest, this will always be an inner-loop with no other nested loops. This loop is assigned the gang, worker and vector directives. Then the algorithm will backtrack to the parent loop (if one exists), and the parent loop will look at all of its children. Based on the parallelism available in its children, it will always take the gang directive from them, it may take the worker directive from them, and it will take the vector directive if the loops are going to be collapsed.

The following pseudo-code shows a representation of the kernel generation algorithm:

```
def createOpenACCLoop(kernel, children):
    if children is empty:
        return loop1D(kernel)
    else:
        return loop2D(kernel, children)

def loop1D(kernel):
    adjustWorkerVector(kernel.size, gang, worker, vector)
    return OpenACCLoop(gang, worker, vector)

def loop2D(kernel, children):
    gang, worker, vector, collapse = optimize2D(kernel,
        children)
    adjustWorkerVector(kernel.size, gang, worker, vector)
    return OpenACCLoop(gang, worker, vector, collapse)
```

If a given loop has no children, then it is assumed that it is an inner loop, so *loop1D* is called. If the given loop does have children, then it must either be an

outer-loop or somewhere in the middle. The given loop and its children loops are treated as two-dimensional; this means that the given loop will be optimized based on its children, and may also make the decision to edit its children. Some of the things that the kernel generation considers in the *loop2D* case:

- Since this loop is potentially an outer-loop, take the *gang* directive from the children, as the *gang* directive must be on the outer-most loop for the kernel to function correctly.
- If any children have the *worker* and *vector* clause, take the *worker*. The vector clause should stay on the inner-most loop, according to OpenACC best practices. When the *worker* clause is actually used, it generally does not belong on the same loop as *vector*.
- If the loops are tightly nested, collapse them; if the child loop is also a collapsed loop, take the collapsed clause and increase the number of loops by one.

The goal of this algorithm is to generate code that follows the OpenACC best practices to achieve good performance. In DawnCC [84], the tool chooses to only give bare minimum annotations, allowing the OpenACC compiler to do the significant amount of the work. However, this in many cases already mentioned would either create broken or poor performing code. On the other extreme, it is possible to do auto-tuning during the kernel generation. This would involve testing every possible legal OpenACC kernel, and choosing the permutation that gives the best performance. However, this is done by the auto-tuning module, and not all users of AutoACC would want to have to run a lengthy auto-tuner everytime they run AutoACC. A good middle-ground is to try and generate the best code using this heuristic and the best practices.

LLVM is an open-source, language non-specific intermediate representation (IR) employed by various compilers and research tools. Clang is a C/C++ compiler developed alongside LLVM to build C/C++ ASTs and produce LLVM IR [76]. Recent releases of Clang provide tools and libraries referred to as *libtooling* for utilizing Clang's AST,

visitors and matchers. There exists some work on using libtooling to create source-to-source compilers such as OP2-Clang which converts C source code to a domain-specific language called OP2 [20].

Using libtooling for AutoACC presented some key limitations. Clang’s AST is very straightforward and recognizable from the language that it supports, however, Clang’s philosophy is that once an AST node is created, it should never be changed. Clang does not perform any optimizations on the AST because it is meant to be as direct an interface as possible to LLVM, since LLVM has a wide variety of optimization tools. This means that libtooling is very useful for building LLVM IR, but does not provide easy access or customization to the AST.

Next, libtooling can parse OpenMP directives as Clang directly supports OpenMP, but does not support any kind of custom directives. However, custom directive support can be achieved in Clang by editing the Clang source code directly. AutoACC needs to support two kinds of new directives; a new suite of directives specific to the AutoACC tool and OpenACC directives. The only way to have libtooling recognize these directly is by creating a custom version of Clang with the necessary source code changes.

Lastly, libtooling does not allow for the creation of new node types. In Clang’s implementation of OpenMP [17] and Clacc’s support of OpenACC [45] new nodes are created to represent directives, and those nodes are integral into generate proper device LLVM IR. There is no way to create new nodes for AutoACC and OpenACC directives through libtooling; it would instead require a heavily edited version of Clang’s source code.

Clang does still provide a very straightforward C/C++ AST, and there is still much to gain from leveraging Clang’s frontend. For AutoACC, Clang is still used for parsing the initial C source code and generating an AST. That AST is then moved over to the AutoACC tool, and the nodes are converted to new classes written for AutoACC’s AST. Clang’s AST is very verbose; it provides a lot of extra details needed for correct LLVM IR generation (such as implicit casts on every expression node) that is trimmed in AutoACC’s AST. Since LLVM/Clang is a large compiler (in terms of

storage space and install requirements) it is also ideal to have a parsing mechanism that does not rely on Clang. In a future direction, a custom C/C++ parser would be implemented specifically for AutoACC but would disregard several common compilers features such as type checking as AutoACC is built on the assumption that the input source code already builds correctly with another compiler.

As stated previously, Clang's libtooling does not provide any support for custom directives parsers. The solution used was to have a custom flex/bison parser that would only identify known directives, ignoring all other code. The result of this is a list of isolated AST directive (*Directive*) and clause (*Clause*) nodes, and their location within the source code. Using the location of each directive, the AST is traversed until the node that would immediately follow the directive is found. Then, the directive node is injected into the AST, and replaces the child position of the original node's parent. Directive nodes are treated as statements, and the AST is designed to accept them as substitutes for compatible node types. For example, the OpenACC loop directive (*OpenACCLoop*) is meant to annotate a for loop (*ForStmt*). This means, anywhere in the AST that a *ForStmt* would be accepted, a *OpenACCLoop* can be substituted.

One issue found with parsing directives is that directives often reserve keywords that are normally not reserved in the programming language. For example, in OpenACC, the keyword *loop* means a very specific thing, but only when used in the context of a directive. However, the word *loop* has no such reservation in the C language. There is also a problem for AutoACC where the keyword *loop* represent an *OpenACCLoop*, or *AutoACCLoop*. This means that from a parser stand-point it would be difficult to parse directives and a programming language from the same parser class. The approach would be to identify directives (`#pragma`) and parse those lines with a different lexer/parser.

Whether or not a loop nest should be collapsed is difficult to decide, however it is a decision that the kernel generation algorithm will have to make. One will definitely want to avoid collapsing loops if it causes any sort of data dependency, which is something that AutoACC will look out for in the future. OpenACC provides several

clauses for combining tightly nested loops. This allows the programmer to make several key optimizations:

- Combining several small loops to create a single larger loop, potentially exposing more parallelism
- Make a loop nest smaller to better emphasize OpenACC's gang, worker and vector clauses
- Segment nested loops into tiles to improve data locality

To decide in which situations loop collapsing may yield better performance, a quick cast study was done. Using an custom version of a kernel called *saxpy*, several loop dimensions, worker and vector configurations were tested. It is in no way an exhaustive list of all possibilities, but this subset does highlight several key situations that are common to consider in OpenACC code.

For the cases of 2097152x128, 2097152x122, 2097152x134 and 2097152x96, these numbers are chosen very deliberately. For NVIDIA hardware specifically, threads must always execute in lockstep in groups of 32. This is called a *warp*. For this reason, when working with NVIDIA GPUs using CUDA, it is best practice to attempt to do everything in multiples of 32. It is my hypothesis that OpenACC for NVIDIA GPUs will follow this same rule, so some of the test cases pointed out intentionally break the warp rule to see if loop collapsing produces better results in this scenario.

Table 4.1 shows the runtime comparison of the *saxpy* kernel with different loop and kernel configurations. The kernel is broken into two tightly nested loops to test the affect of collapsing. When the two loops are evenly sized, the runtime difference between the collapsed and non-collapsed version is small, with the collapsed version slightly pulling ahead. When the size of these loops are increased to be very large, collapsing actually sees a significant performance decrease.

When the inner loop is significantly smaller, we start to see a clear difference. For the 2097152x128 loop configuration, the inner loop with a size of 128 perfectly fits

Table 4.1: Performance analysis of the OpenACC collapse clause

	1x128 NC	1x128 C	4x32 NC	4x32 C
8192x8192	0.000997	0.000977	0.001097	0.000979
16384x16384	0.004093	0.003941	0.004454	0.003970
32768x32768	0.016403	0.020638	0.017943	0.020723
2097152x128	0.003954	0.003940	0.003854	0.003969
2097152x122	0.004264	0.003784	0.004747	0.003771
2097152x134	0.007235	0.004185	0.005425	0.004173
2097152x96	0.002977	0.002992	0.002939	0.002977

Kernel configurations are: 1 worker with vector length 128 non-collapsed and collapsed (1x128 NC, 1x128 C) and 4 workers with vector length 32 non-collapsed and collapsed (4x32 NC, 4x32 C). Timings are using the *saxpy* kernel, a single NVIDIA V100 GPU, and the PGI 19.10 compiler. The times are in seconds and averaged over 100 runs of the kernel.

the 128 vector length, and the collapsed and non-collapsed versions perform nearly identical. However, when the loops size is adjusted slightly, and more importantly when the loop size does not fit evenly into CUDA warps, the collapsed version begins performing significantly better. When the loop does not evenly fit into a warp, resources are wasted, whereas with the collapsed loop, nearly every warp can be fully utilized. In the 2097152x96 we can see the results are again very close since the thread block size of 96 fits evenly into 3 warps, there are no wasted threads.

In real-world codes, there will likely be many situations where loops are not evenly divisible by 32, which seems to hint that in a general case the collapsed loops will likely perform better. With this information, the OpenACC kernel generator will collapse loops whenever possible, however, this decision can be undone in the auto-tuner phase if it ends up being non-optimal.

#### 4.4 Performance and Code Generation Results

The following code shows the input and output of AutoACC on the *matvec* kernel before auto-tuning.

The left code shows the *matvec* kernel annotated with the new AutoACC directives. These new directives provide AutoACC with a few bits of important

```

#pragma autoacc loop                                     \
  size(16974593) stride(1)                             \
  arrays(row_offsets(num_rows+1),                     \
         cols(nnz), Acoefs(nnz),                     \
         xcoefs(num_rows),                           \
         ycoefs(num_rows))
for(int i=0;i<num_rows;i++) {
  double sum=0;
  int start=row_offsets[i];
  int end=row_offsets[i+1];
  #pragma autoacc loop size(27) stride(1)
  for(int j=start;j<end;j++) {
    unsigned int Acol=cols[j];
    double Acoef=Acoefs[j];
    double xcoef=xcoefs[Acol];
    sum+=Acoef*xcoef;
  }
  ycoefs[i]=sum;
}

```

Figure 4.4: Matvec kernel annotated with AutoACC.

information that a compiler would normally not have; the outer-most loop is very large, and can express a lot of parallelism. The inner-most loop is very small, even slightly smaller than a CUDA warp, so parallelism is limited, and block dimensions will need to be adjusted. Also, these loops are not perfectly nested, so collapsing them or tiling them is not possible without rewriting the loop bodies.

The right code shows the generated OpenACC code. Since the inner loop is small, the vector length was reduced from the default value (128 to 32), and the number of workers was increased (1 to 4). This means that less threads within a warp are wasted, but the total threads per thread block stay at the default value of 128. Also, OpenACC clauses for handling GPU data management are added, as the GPU code could not compile or run without them.

The next code shows the changes that the auto-tuner has made to the laplace kernel.

```

#pragma acc parallel                                     \
  copyin(row_offsets[0:num_rows+1],                     \
         cols[0:nnz], Acoefs[0:nnz],                   \
         xcoefs[0:num_rows])                             \
  copyout(ycoefs[0:num_rows])                           \
  num_workers(4) vector_length(32)
{
  #pragma acc loop gang worker
  for(int i=0;i<num_rows;i++) {
    double sum=0;
    int start=row_offsets[i];
    int end=row_offsets[i+1];
    #pragma acc loop vector
    for(int j=start;j<end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
  }
}

```

Figure 4.5: Matvec automatic conversion to OpenACC.

On the left is the output of the code generation module. The code generation saw that the loops were tightly nested, and decided to collapse them. Since the inner-loop is adequately large, the vector length was not changed from the default value. On the right is the auto-tuned version. The auto-tuner found that the optimal configuration for this kernel is to tile the loops instead of collapse, and to greatly increase the threads per thread block. From having knowledge of the execution of this kernel, this is actually expected. The kernel reads memory from the matrix in both the x and y direction, so it was likely that a tiled loop would increase data locality and reduce the overall time the GPU is waiting for data to be fetched.

Saxpy is a single-precision combination of a scalar multiplication and vector addition shown in Equation 4.1. Saxpy is one of the most typical kernels found in linear

```

#pragma acc parallel \
  copyin(A[0:M*N]) copyout(B[0:M*N]) \
  num_workers(1) vector_length(128)
{
  #pragma acc loop gang worker vector \
    collapse(2)
  for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++) {
      int ind = i*N+j;
      B[ind] = (A[ind-off1] +
                A[ind-off2] +
                A[ind-off3] +
                A[ind-off4])/4;
    }
  }
}

```

Figure 4.6: Laplace2D

algebra (BLAS) libraries, and is commonly used as an introductory GPU programming code due to its simplicity and importance.

$$\mathbf{z} = \mathbf{ax} + \mathbf{y} \quad (4.1)$$

The laplace equation is a second-order partial differential equation. Equation 4.2 shows an implementation of the laplace equation for heat conduction across a 2-dimensional plate.

$$B_{i,j} = \frac{A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1}}{4} \quad (4.2)$$

The remaining kernels are from a a Poisson 3D conjugate gradient solver code. This code is often used for intermediate GPU programming lectures and presents some special cases that create interesting performance optimization possibilities that simpler codes do not have. The dot kernel in Equation 4.3 is a vector dot product, which presents a simple summation. The waxpby kernel in Equation 4.4 is a multi-precision double scalar multiply vector addition, very similar to saxpy. The matvec kernel in

```

#pragma acc parallel \
  copyin(A[0:M*N]) copyout(B[0:M*N]) \
  num_workers(1) vector_length(1024)
{
  #pragma acc loop gang worker vector \
    tile(32,32)
  for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++) {
      int ind = i*N+j;
      B[ind] = (A[ind-off1] +
                A[ind-off2] +
                A[ind-off3] +
                A[ind-off4])/4;
    }
  }
}

```

Figure 4.7: Laplace2D Tuned

Equation 4.5 is a sparse matrix-vector multiply, and is the main computational kernel in this conjugate gradient solver.

$$\mathbf{x} \cdot \mathbf{y} \tag{4.3}$$

$$\mathbf{z} = a\mathbf{x} + b\mathbf{y} \tag{4.4}$$

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{4.5}$$

These kernels were selected for a few different reasons. Saxpy is by far the most common introduction to GPU code, meaning that it was a good early goal for the AutoACC tool. Laplace is also a common tutorial code, but is significantly more complex computationally. Laplace also introduces new interesting concepts for performance and auto-tuning. Dot, waxpby and metvec all come from a GPU accelerated Poisson 3D solver. Dot and waspby are similar in complexity as saxpy, but are different enough that they have helped include some new features into AutoACC. Matvec is the most

complex of the kernels used, and provides several expected optimizations that present key victories for the auto-tuner.

The runtime results were collected for each kernel in several different configurations. The timing for each kernel only includes the time for computation; any data communication between the host and GPU is removed as it would be a constant number across all variations and make the results more difficult to interpret. Each run also executes a dummy GPU kernel before any timing starts, as the first kernel usually must initialize GPU drivers and suffers a significant overhead. The dummy kernel is used to deal with this overhead before any of the timed kernels run.

Table 4.2: **Results of AutoACC**

	Serial	Kernels	Basic	Generated	Auto-tuned
saxpy	0.04220	0.00146	0.00146	0.00146	0.00146
waxpby	0.03546	0.00197	0.00197	0.00197	0.00197
dot	0.00654	0.00190	0.00190	0.00191	0.00190
laplace	0.09892	0.01327	0.00392	0.00280	0.00223
matvec	0.36059	0.03187	0.03187	0.01099	0.01057

All runtime measurements are in seconds and averaged over 10 runs. Machine used contains AMD Ryzen Threadripper 1950X 16-core CPU and single NVIDIA Volta V100 GPU. Kernels *saxpy*, *waxpby* and *dot* use a data vector of 100,000,000 elements, *laplace* uses a 10,000x10,000 matrix, and *matvec* uses a 256x256x256 sparse matrix. Serial is running the code on a single core of the CPU. Kernels is parallelization only using an outer kernels directive like the work presented in DawnCC [84]. Basic is parallelization by naively annotating all loops with loop directives. Generated is the output of AutoACC’s code generation module. Auto-tuned is the output of AutoACC’s auto-tuner module.

The results for all configurations for the *saxpy*, *waxpby* and *dot* kernels are nearly identical in performance. This means that each of these kernels are simple enough that the PGI compiler is able to optimize them very efficiently already. However, with more complex codes we begin to see a large disparity.

In all of the runs, the basic parallelization is as good as or better than the pure kernels directive approach. This highlights that the compiler is not always able to properly identify parallel loops without some additional information from the

programmer. Specifically for the *laplace* code, the kernels directive was unable to parallelize the inner loop, and thus produced significantly worse results (nearly 600% slower than the best run).

Also, in all runs, the code generated by AutoACC without auto-tuning performed as good as or better than a hand-written basic parallelization. This is clear to see in both *laplace* and *matvec*. In *laplace*, the loop nest was collapsed by AutoACC, which resulted in better performance than keeping the loop separate. In *matvec*, AutoACC adjusted the vector length and number of workers to accommodate the small inner loop, which provided a significant performance increase.

For *laplace*, the auto-tuner changed the collapse clause for a tile instead, resulting in a 25% performance increase. For *matvec*, the auto-tuner increased the number of workers from 4 to 32, resulting in a 4% increase.

## Chapter 5

### THREE-LEVELED HIERARCHICAL GPU PARALLELISM IN THE OPENMP PROGRAMMING MODEL

Large parts of the text of this chapter has been published in the 52nd International Conference on Parallel Processing (ICPP 2023), ACM <https://dl.acm.org/doi/10.1145/3605573.3605640>.

This chapter focuses on the project to provide programmer control to all levels of GPU parallelism by designing and implementing OpenMP’s “simd” directive within LLVM’s OpenMP/GPU runtime. This project includes a new code generation scheme for OpenMP loops, a CPU-conforming execution model for “simd” loops, and an optimized GPU-centric model applicable to a subset of codes.

#### 5.1 Introduction

OpenMP has supported GPU offloading since the 4.0 standard with the inclusion of new `target` and `target data` directives, and has been extended and improved in subsequent OpenMP versions. User experiences of applying OpenMP target offloading can be found in some of the recent work including using the SPEChpc2021 benchmarking suite [28], other applications including HPGMG [39], miniMD [91], UK mini-apps [82], LULESH [68] among others. Work in [31, 92] discusses at length experiences gained and practices adopted from OpenMP hackathons when applying offloading features on HPC applications and mini-apps based on different computational motifs (BerkeleyGW, WDMApp/XGC, GAMESS, GESTS, and GridMini) targeting heterogeneous systems.

The OpenMP offloading support for GPUs in LLVM can be traced back to the two works discussed in [23, 22]. The (PGI) Fortran front-end, known as Flang, supported OpenMP offloading via the LLVM OpenMP runtime [90]. Since then, researchers have

been working on compiler and runtime optimization for LLVM OpenMP. The first front-end-based optimizations for NVIDIA GPUs that can avoid idle threads and reduce register usage was introduced in [16]. Work in [48] presented the TRegion interface which delays the discovery of SPMD regions to compiler middle end, contrary to the front-end based approach used before, which can support more kernels to execute in SPMD mode. Runtime support for concurrent execution of OpenMP target tasks was introduced in [115]. Results in [62] discusses OpenMP-aware program analyses and optimizations that allow efficient execution of the generic, CPU-centric parallelism model provided by OpenMP on GPUs. A co-design methodology is presented in [49] for optimizing applications using a specifically crafted OpenMP GPU runtime inducing near-zero overhead in most cases.

OpenMP offloading utilizes a host-device execution model where the host (CPU) schedules and synchronizes target tasks, in the form of *kernels*, and handles memory allocation and movement between the host and target devices (e.g. GPUs). Computational kernels are executed on the device by launching a league of *teams*. Each team has a *main thread* that will begin executing the code region contained by the `teams` directive. Additional *worker threads* can be spawned by using the `parallel` directive. There are three worksharing constructs: `distribute`, `for` and `simd`. `distribute` schedules loop iterations across the league of teams, `for` schedules loop iterations across threads within a team and `simd` uses single instruction multiple data (SIMD) parallelism for the loop.

GPU execution models utilize a similar structure with multiple streaming multi-processors (SM), each containing several parallel work units (warps for NVIDIA GPUs and wavefronts for AMD GPUs) that are able to execute simultaneously. A simple mapping of the OpenMP model to GPU hardware is a team per each SM, and threads within the team to hardware threads within the SM.

The `simd` construct specifies that the attached loop should be executed using SIMD parallelism. For GPUs this is typically done using single instruction, multiple thread (SIMT) parallelism, meaning that multiple threads within a computational unit execute the same instruction. This means that in terms of GPU offloading a `simd`

```

#pragma omp target teams distribute parallel for
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma omp simd
    for(int j=row_start;j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        #pragma omp atomic update
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}

```

Figure 5.1: Sparse matrix vector multiplication kernel with OpenMP offloading. The code highlights an small, variably sized inner-loop that benefits from the intermediary warp-level parallelism.

loop should be executed in parallel by a set of adjacent threads. We can achieve this by separating the threads in each team into distinct groups. These *simd groups* will contain a single main thread and multiple workers. The main thread executes `parallel` regions and all threads in the group execute `simd` loops.

To clarify the terminology used, we must differentiate between different uses of the same word. A common form of parallelism is data parallelism which is often referred to as Single Instruction Multiple Data (known commonly as SIMD). GPU architecture uses a very common concept called Single Instruction Multiple Thread (known commonly as SIMT), which is very similar too SIMD and is often reasonable to assume that these phrases can be used interchangeably. From here on, SIMD (when spelled in all capital letters) refers to this acronym.

Within the OpenMP standard there exists a directive called “`simd`”. OpenMP `simd` is a loop directive that specifies that the loop should be executed using SIMD parallelism. From hereonout any use of `simd` (when spelled in all lower case) refers to the OpenMP directive.

## 5.2 Parallel Loop Code Generation in LLVM/Clang

In this section we outline the the code generation for a new work-sharing loop API in Clang/LLVM.

### 5.2.1 Clang AST and Code Generation

We have altered Clang’s code generation for OpenMP `simd` loops to instead use our new function in the OpenMP IR Builder. These changes are all found within the Clang code generation in a file called “CGStmtOpenMP.cpp”. The two key requirements that need to be met to create a `simd` loop is a callback function to generate LLVM IR for both the trip count of the loop and the body of the loop. Clang’s *OpenMPSimdDirective* class is used to represent the OpenMP `simd` directive. Since this class is a loop directive it contains an *OMPCanonicalLoop* node as one of its children. The *OMPCanonicalLoop* has some built-in methods that are particularly useful for determining the trip count of the loop as well as resolving the loop variable.

The *OMPCanonicalLoop* is used to generate the LLVM IR for the loop trip count callback. Then, in the body callback a local allocation is created for the loop variable, and the *OMPCanonicalLoop* is used to initialize that loop variable based on the current loop iteration number. Lastly, Clang emits the *CompoundStmt* which includes the body of the loop.

Starting with the trip count callback, the *OMPCanonicalLoop* is used to generate a function that will calculate the trip count of the loop and return that value. This function requires the loop variable allocated and initialized, as well as other values such as the loops ending point and step value, if applicable. The type of the return value is based on the type that the loop variable uses, which may be cast to a different type later in the OpenMP IR Builder.

Next, the loop body callback is broken into two steps. First, code is generated that converts the iteration number into the relevant loop variable. The *OMPCanonicalLoop* has a method that generates a function in a similar fashion as used in the trip count calculation for determining the loop variable. Importantly, at this point the loop

variable has already been allocated from before the trip count calculation. However, because the loop body will be outlined later in the IR Builder, the loop variable needs to be changed to a local allocation in the loop body. The loop variable used within the trip count and loop variable calculation is replaced with a new value and a new loop variable is allocated locally in the loop body. Then the loop body is emitted from Clang as a generic Stmt node. All references to the loop variable within the body will now use the new local allocation.

While our work uses Clang as the front-end, the changes described would be applicable to any potential front-end wanting to use LLVM's GPU runtime. The front-end would have to provide code generation for the loop trip count and the loop body, similar to the methodology we have described. Then the OMP IR Builder would perform the loop task outlining and generate the appropriate runtime function calls. Loop scheduling is then performed from within the runtime.

### 5.2.2 LLVM's OpenMP IR Builder

LLVM's *OpenMP IR Builder* is used to generate OpenMP target code and to interface with the LLVM/OpenMP runtime library. This tool is designed to be front-end independent and allows for a generalized approach to creating parallelism with OpenMP without requiring a compiler to do extensive parallel code generation. A compiler may interface with the OpenMP IR Builder by creating callback functions, which handle certain parts of the code generation while the IR Builder will generate the code needed for OpenMP parallelism, including runtime library function calls.

We have added new functions to the OpenMP IR Builder to generate code for OpenMP worksharing loops. This requires two callback functions: 1) to generate the trip count of the loop, and 2) to generate the body of the loop. The generated loop body will later be isolated and moved into a separate function in a process called *outlining*, which allows the body to be passed into the OpenMP runtime by using the *outlined function* as a pointer. This function represents the task that a single thread would do

to execute a single iteration of the loop. Then the runtime will handle work distribution across threads and ensure that all iterations are executed.

Since the outlined function may reference variables that are no longer in the correct scope, these variables must be passed to the function as arguments. They are aggregated into a structure and passed as a singular payload to the outlined function. The payload is packed before the runtime function call and unpacked within the outlined function. Special consideration for these variables is needed for `simd` loops, since the generic execution mode requires the main thread to communicate these variables to the worker threads. In this case any variable used within the outlined region needs to exist in either shared or global memory such that it is accessible by all threads. During the outlining if any variables in the payload are local allocations from the encompassing parallel region, then those allocations are *globalized* [62], and the corresponding memory is deallocated at the end of the parallel region.

For `simd` loops executing in the CPU-centric generic mode (reg. Section 5.3.3) some variables will need to be shared among threads and will be globalized. When a `simd` loop is generated, any variables used within the body of the loop (which are the variables that will be passed to the outlined function) are checked to determine which memory they reside in. If the variable is a local allocation (i.e only visible to the current thread) it will be replaced with a shared memory allocation. If the variable is untraceable (such as the case if its allocation is in another translation unit) then it will be copied to shared memory just before the `simd` loop is executed.

### 5.2.3 Worksharing Loops in LLVM/OpenMP's GPU Runtime

Fig. 5.2 shows the implementation for executing `simd` loops within the runtime. This code specifically shows `simd` loops, but it generally applicable to all of OpenMP's loop constructs. The `WorkFn` variable is the outlined function which contains the body of the loop that each thread will execute. The `TripCount` variable is the total number of iterations that the loop should run for. Lastly, the `Args` variable is the payload

passed into the outlined function which may contain any number of pointers from global, shared, or local memory.

```
void __simd_loop(void *WorkFn, uint64_t TripCount, void **Args) {
    uint64_t omp_iv = getSimdGroupId();
    while(omp_iv < TripCount) {
        WorkFn(omp_iv, Args);
        omp_iv += getSimdGroupSize();
    }
    synchronizeWarp(simdmask());
}
```

Figure 5.2: Function for executing `simd` loops. Each thread will execute a portion of the total iterations.

The key difference in this function for other loop directives is in the initial iteration value and the stride. For `distribute` loops the team ID and the number of teams would be used for the initial value and stride, and for `for` loops the SIMD Group ID and number of groups is used for the initial value and stride.

Indirect calls using function pointers is normally costly. However, LLVM/Clang performs a front-end static analysis that creates an if/cascade to compare the function pointer against known outlined regions, a methodology defined in [23]. In the case that the region is not known and cannot be placed in this if/cascade, such as regions in functions defined in other translation units, an indirect call is emitted as a fallback option, following the functionality of Fig. 5.3.

### 5.3 SIMD in LLVM/OpenMP’s GPU Runtime

This section will describe the design and implementation of OpenMP’s “`simd`” directive in LLVM’s GPU runtime, including changes needed to LLVM’s original model to enable this addition.

#### 5.3.1 OpenMP/GPU Hardware Mapping

Fig. 5.4 shows an example of mapping a potential OpenMP target region onto a NVIDIA GPU. The `teams` directive spawns a league of teams and each team may

```

switch(WorkFn) {
  case outlined_1:
    outlined_1(Args);
    break;
  case outlined_2:
    outlined_2(Args);
    break;
  default:
    WorkFn(Args);
    break;
}

```

Figure 5.3: LLVM optimization for efficiently calling outlined functions.

contain many threads. This figure shows a single OpenMP team, but typically the league would contain several teams depending on the maximum number of concurrent threads the hardware allows. One thread within the team will be distinguished as the team main thread and will be in charge of running the code contained within the teams region.

The `parallel` directive spawns a team of threads to execute the parallel region. For this work, the team of threads is evenly divided into SIMD groups, where all threads within a group occupy the same warp. Our implementation does not allow for SIMD groups to encompass multiple warps as it extensively utilizes warp-level thread barriers. One thread in each group is designated as the *SIMD main thread* and will execute `parallel` regions while all other threads are *SIMD worker threads* and will execute `simd` loops.

The `simd` directive specifies that the attached loop should be executed using SIMD parallelism. For GPUs this is done by parallelizing across adjacent threads in a warp. For our implementation a `simd` loop distributes loop iterations across threads in the same SIMD group.

When a `teams` region is executing in generic mode an additional warp is assigned to act as the team main thread. This additional warp is needed for the purpose of thread synchronization as discussed in [65]. However, synchronization of threads within

a SIMD group is done using a warp-level synchronization, which does not have the same limitations.

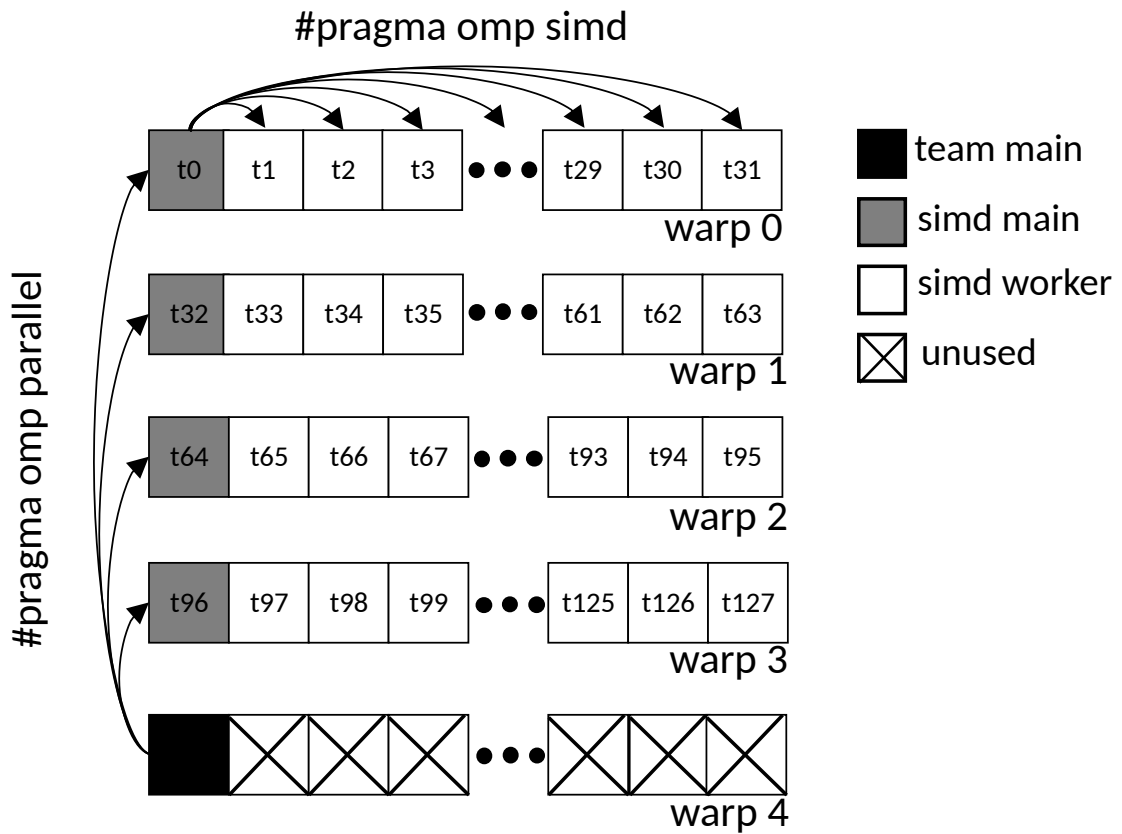


Figure 5.4: A possible mapping of a single OpenMP team on an NVIDIA GPU using four warps, totaling 128 threads for the computation. One SIMD group per warp, meaning one SIMD main and 31 SIMD workers per warp. One additional warp is included to act as the main thread in the team, which is required when the teams region is executing in generic mode.

The following functions have been created to handle the mapping of the SIMD groups within the runtime:

- `getSimdGroup` returns which group the thread belongs to.
- `getSimdGroupId` returns the thread's ID within its group. SIMD main threads always have an ID of 0.
- `getSimdGroupSize` returns the size of the SIMD group. All SIMD groups are the same size and the size could differ between parallel regions.

- `isSimdGroupLeader` returns true if the thread is a SIMD main thread for its group.
- `simdmask` returns a bit-mask that represents which threads in the warp share a SIMD group with the thread.

### 5.3.2 Overview of General Runtime Changes

At the start of an offloaded region all threads will begin by calling the `__target_init` function, which generally initializes the shared team state. It is also an important divergence point for the threads in the team. If the `teams` region executes in SPMD mode all threads will return from this function and immediately begin executing the user code. If the offloaded region will instead execute in generic mode only the team main thread will return to the user code while all the other threads will enter into a state machine where they will immediately encounter a thread barrier and remain idle until the main thread encounters an OpenMP `parallel` region.

`parallel` regions are handled through the runtime function `__parallel`. If running in SPMD mode all threads will reach the same call of `__parallel` and all threads will independently resolve the function pointer and handle the variable payload. In generic mode only the main thread will reach the `__parallel` function and the worker threads must be notified of the parallel region and any needed variables. When the main thread completes the thread barrier the worker threads will fetch the outlined function pointer and any variables used within the outlined function before executing it. Fig. 5.5 shows the `__parallel` function assuming the encompassing `teams` region is executing in SPMD mode.

Regardless of whether the `teams` region is SPMD or generic mode the runtime reaches another important divergence point in `__parallel` where each OpenMP `parallel` region can also be either SPMD or generic. In SPMD mode, all threads within the team will execute the `parallel` region, while in generic mode the main thread in each SIMD group will execute the `parallel` region and worker threads enter the SIMD state machine and immediately encounter a warp-level barrier and wait for

```

void __parallel(
    void *fn, void **args, int64_t nargs, int32_t SPMD
) {
    if(SPMD) {
        // All threads execute region in SPMD mode.
        invokeMicrotask(fn, args, nargs);
        return;
    }
    if(isSimdGroupLeader()) {
        // Only simd mains execute region in generic mode.
        invokeMicrotask(fn, args, nargs);
        // Send termination signal to simd workers
        setSimdFn(nullptr);
        synchronizeWarp(simdmask());
    } else {
        // Simd workers enter the state machine.
        simdStateMachine();
    }
}

```

Figure 5.5: A portion of the `__parallel` runtime function showing the two different execution modes that parallel regions can be. If the `parallel` region is SPMD mode then all threads within the SIMD group will execute it. If it is instead generic mode only the SIMD main thread will execute the parallel region while all SIMD workers enter into a separate SIMD state machine.

a `simd` loop to be encountered. A call to the new runtime function `__simd` signifies a worksharing loop for SIMD parallelization. Fig. 5.6 shows this function with the two different execution modes.

### 5.3.3 CPU-Centric SIMD-Generic Mode

Fig. 5.7 shows how each thread functions within the runtime. When the threads encounter an OpenMP `parallel` region that is executing in generic mode the threads will split into two possible paths, similar to `teams` generic mode. Threads that are designated as SIMD main will begin executing the `parallel` region user code. Fig. 5.4 shows a possible configuration of these SIMD mains with one main thread per warp, however it is possible to have multiple SIMD mains per warp. Threads that are designated as SIMD workers will enter the SIMD state machine and become idle while

```

void __simd(void *WorkFn, uint64_t TripCount,
            void **Args, uint32_t NumArgs) {
    if(isParallelSPMD()) {
        // In SPMD all threads in the SIMD group
        // execute the loop
        __workshare_loop_simd(WorkFn, TripCount, Args);
        synchronizeWarp(simdmask());
        return;
    }

    // In generic SIMD main thread sets up the
    // group state and signals the workers
    setSimdFn(WorkFn, TripCount);
    void **GlobalArgs;
    __begin_sharing_simd_args(&GlobalArgs);
    for(uint32_t i = 0; i < NumArgs; i++)
        GlobalArgs[i] = Args[i];
    synchronizeWarp(simdmask());
    __workshare_loop_simd(WorkFn, TripCount, GlobalArgs);
    synchronizeWarp(simdmask());
}

```

Figure 5.6: Runtime function for OpenMP `simd` loops in SPMD or generic mode. If the `parallel` region is generic mode then all variables needed within the `simd` loop must be shared from the SIMD main thread, and the SIMD workers must be notified of what loop should be executed and for how many iterations. If the `parallel` region is instead in SPMD mode then this information is already local to each thread and no communication needs to occur.

waiting for the main thread to encounter a `simd` loop. This is done through a warp-level barrier using a bit-mask to identify all threads within the same SIMD group. Fig. 5.8 shows the implementation for this state machine.

When the SIMD main thread encounters a call to `__simd`, it updates the SIMD group state with some information about the current `simd` loop, such as a function pointer that references the outlined function to be executed, the trip count of the loop and the addresses of all variables needed within the outlined function. When the SIMD main thread reaches the end of the current parallel region it sets the outlined function pointer in the SIMD group state as a `nullptr` which signifies a termination signal, and then notifies the workers through the warp synchronization. After this, all threads

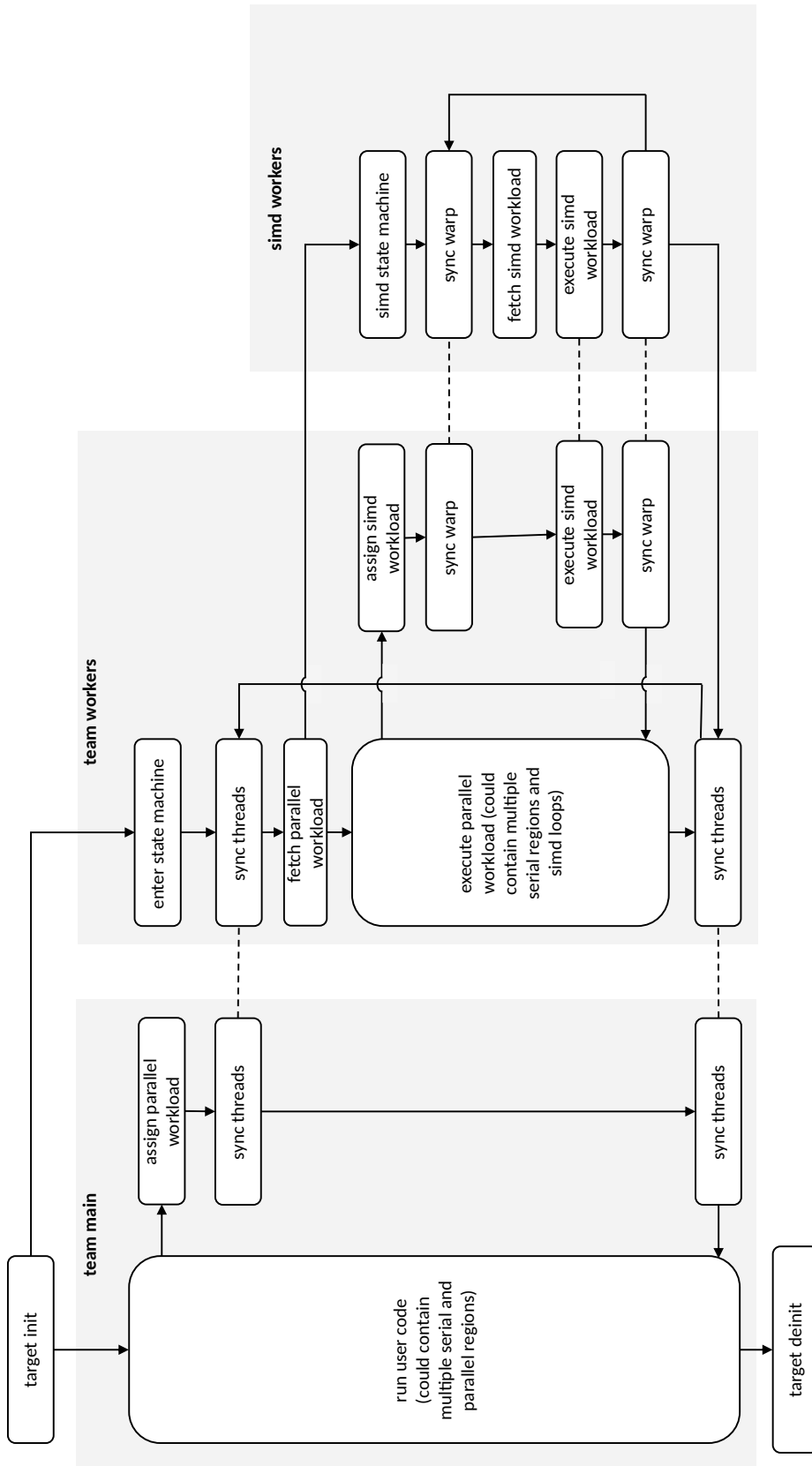


Figure 5.7: Program flow diagram of the GPU runtime assuming all regions are executed in generic mode.

```

void simdStateMachine() {
    do {
        void *WorkFn;
        void **FnArgs;
        uint64_t TripCount;

        // Wait for work
        synchronizeWarp(simdmask());
        getSimdFn(&WorkFn, &TripCount);
        if(!WorkFn) // Terminate at end of parallel region
            return;

        // Fetch shared variables and execute loop
        getSimdArgs(&FnArgs);
        __workshare_loop_simd(WorkFn, TripCount, FnArgs);
        synchronizeWarp(simdmask());
    } while(true);
}

```

Figure 5.8: New state machine for SIMD workers when the containing `parallel` region executes in generic mode. Workers immediately reach a warp-level barrier. Once the SIMD main thread completes the barrier all workers will fetch the function pointer of the current `simd` loop, as well as any variables shared from the SIMD main thread. If the function pointer is a null pointer then the worker threads exit the state machine, as this signifies the end of the current `parallel` region.

within that SIMD group will exit and run into a team-level barrier, where they will wait for all threads in all SIMD groups to finish the `parallel` region.

#### 5.3.4 Variable Sharing

When running in generic mode variables used within `parallel` regions and `simd` loops need to be shared from the main thread to all worker threads. These variables are always stored as pointers such that each variable is a consistent size. A static allocation of memory is reserved in GPU shared memory exclusively for these variables. Prior to our work the only thread that would write to this shared memory was the singular team main thread. If more variables needed to be shared than what the pre-allocated memory could hold, a global allocation is created to hold the variables instead, with that memory being deallocated at the end of the parallel region.

Now that this variable sharing space is written to by the team main thread and all SIMD main threads, the size of this space is increased and the available space is divided evenly among the SIMD groups. If a SIMD group needs more space than what is available a global memory allocation is created instead, which means that each SIMD group will have a pointer which correlates to where variables are stored (either in the shared memory or in a new global memory allocation).

Additionally, the size of a SIMD group can differ among different parallel regions. As an example using NVIDIA GPUs, if a target region is launched using 128 threads across 4 total warps, the number of total SIMD groups would be in the range of  $4 \leq NumGroups \leq 64$ , with the threads per group being  $2 \leq ThreadsPerGroup \leq 32$  (for 32 threads in a warp). If the group size is less than two then the parallel region would run on all threads in the team and all `simd` loops would execute sequentially. Different parallel regions can use a different number of threads per group which results in a varying number of groups. In a case where a large number of SIMD groups are used the variable sharing space is less likely to be able to fit all variables.

The most noteworthy change in hardware resource usage from our work comes in the form of these shared memory changes. Originally, 1,024 bytes of shared memory

were reserved as the variable sharing space. We have increased this to 2,048 bytes to help accommodate the new SIMD groups. This number is subject to change as more experimentation is done to select a size that is tailored for a typical code utilizing `simd`. Additionally, shared memory usage in general is increased for codes using our generic SIMD implementation as variables needed within the `simd` loops need to be moved to shared memory to be accessible by all threads within the SIMD group. This will vary by code, and will also be mitigated completely when using SPMD mode (reg. Section 5.3.5).

### 5.3.5 GPU-Centric SIMD-SPMD Mode

A `parallel` region using SPMD mode will be executed by all threads in the team. Unlike the generic mode, there is no difference between SIMD main and SIMD workers. All threads will allocate any variables local to the parallel region, determine the trip count of the loop, load the variable payload and call the `__simd` runtime function using the outlined function pointer. Since all of this information is now local to each thread there does not need to be any communication like in generic mode and variables local to the parallel region that are needed in a `simd` loop do not need to be moved to shared memory.

In the case where the `simd` directive is unused, `parallel` regions will always execute in SPMD mode with a SIMD group size of one. This signifies that only two levels of parallelism should be used and behaves identically to the current implementation of LLVM/Clang. Fig. 5.9 shows how SIMD worker threads handle both SPMD and generic modes.

Similar to `teams` regions, `parallel` regions executing in SPMD mode must not produce side-effects. In the case where all `simd` loops are tightly nested within the `parallel` region then no side-effects will occur. However, in any other scenario there may need to be some level of thread guarding and variable broadcasting to eliminate side-effects, such as [62] describes for `teams` regions.

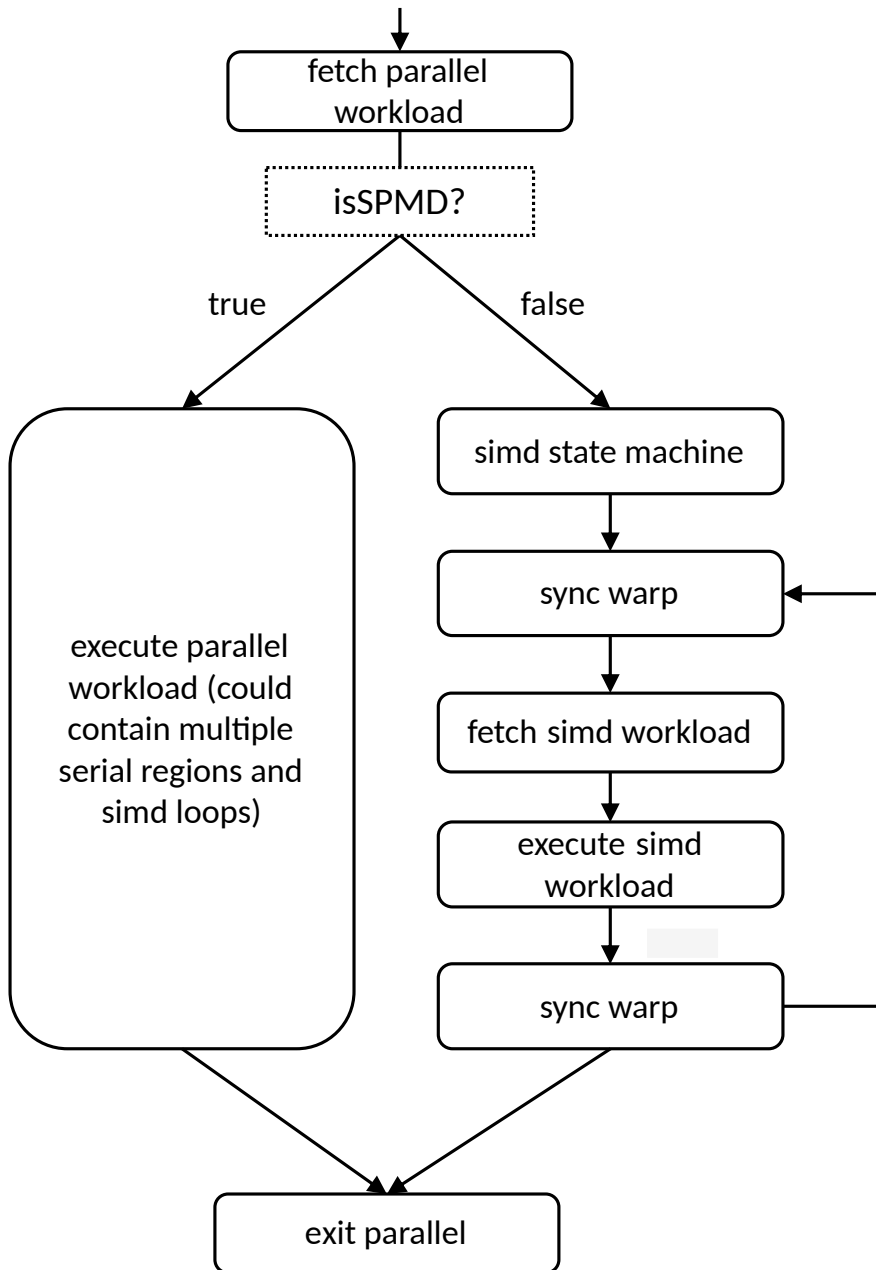


Figure 5.9: Flow diagram for SIMD worker threads upon encountering a parallel region. If the region should be executed in SPMD mode, worker threads will execute the entire region under the assumption that no side-effects will be produced. If the region should instead be executed in generic mode, worker threads will enter into the state machine and wait for a `simd` loop to be encountered.

### 5.3.6 Towards AMD GPU Support

AMD GPUs introduce some limitations to our execution model. LLVM/OpenMP does not provide an implementation for wavefront-level barriers, making our SIMD generic mode implementation incompatible with AMD GPUs. For this reason, our implementation only currently supports SPMD mode for AMD GPUs. If a parallel region would run in generic mode all `simd` loops will run sequentially. There may be some possibilities to implement generic mode on AMD GPUs using alternate methods for the thread barrier, however we do not yet know the viability of such approaches and will need to be explored as a future direction.

## 5.4 LLVM/OpenMP SIMD Results

These results will primarily focus on the performance of the `simd` implementation in the different execution modes. We are not able to fully address shortcomings in the OpenMP runtime [114] outside of the additions made in this work. Additionally, we cannot comment on the general performance of AMD GPUs in LLVM as it is not mature enough, and is not yet fully supported by our implementation. All of the experiments run were using A100 GPUs.

SIMD is not universally useful on all codes. The codes used in these results were selected specifically with the knowledge that they are compatible with three levels of parallelism and will benefit from this optimization. Since `simd` is not a well-supported feature among OpenMP offloading compilers there are few benchmarks that utilize the `simd` directive, so several of the codes used here either had the `simd` directive added for this work or were adapted from OpenACC which has a mature three-leveled parallel implementation.

We also observe the performance increase from several codes with known benefits from three-leveled parallelism. `sparse_matvec` is a sparse, matrix-vector product kernel adapted from an OpenACC code described in [10]. The inner-most loop of this kernel is relatively small, and varies based on the sparsity of the matrix. This kernel also originally used a data reduction on the product calculated in the inner-most loop,

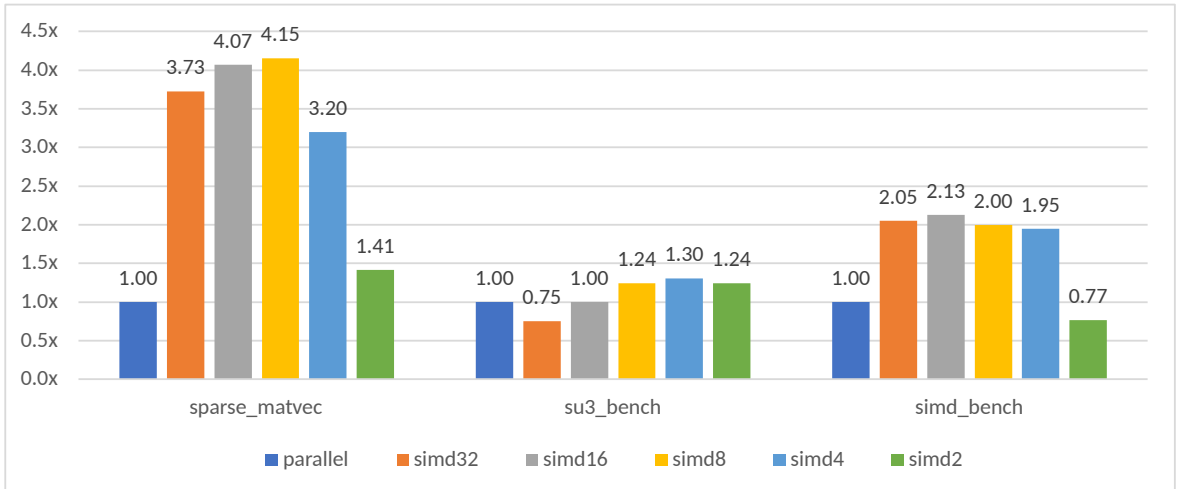


Figure 5.10: Results for various kernels comparing our `simd` implementation to the original two levels of parallelism. Experiments with all possible SIMD group sizes.

however reductions are not yet implemented for our new loop execution model, so instead we use a less efficient atomic update for the product.

To utilize the original two levels of parallelism we parallelize the outer loop with `teams distribute` and the inner loop with `parallel for`. With this structure the `teams` region will run in generic mode. For the three levels of parallelism we parallelize the outer loop with the combined `teams distribute parallel for` and the inner loop with `simd`, meaning the `teams` region will execute in SPMD mode and the `parallel` region in generic mode.

SU3\_bench [50] has a small inner-loop with 36 total iterations that was originally executed serially by each thread. We now apply `simd` to this loop to allow for SIMD parallelism on the GPU. In this code both `teams` and `parallel` regions are SPMD mode.

We have also created a new benchmarking kernel that very closely fits the three levels of parallelism to gauge the performance increase that these optimizations could potentially provide in an ideal scenario. This kernel example has a small inner loop that fits into a single warp, but is not collapsible with the outer-loop nest. We parallelize the outer-loop with `teams distribute parallel for` and the inner-loop with `simd` using SPMD mode for both `teams` and `parallel`.

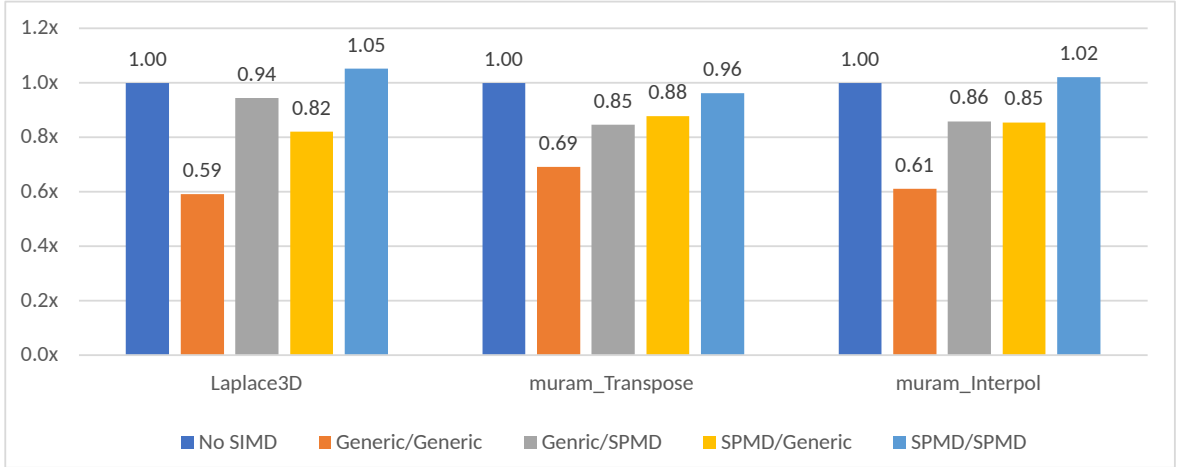


Figure 5.11: Relative speedup of the different combinations of execution modes. The modes are all combinations of `teams` Generic or SPMD / `parallel` Generic or SPMD. The performance similarity for SPMD/SPMD and “No SIMD” suggests low performance overhead in our SPMD implementation.

Fig. 5.10 shows the relative speedup over the two-level parallel baseline. For `sparse_matvec` we see a maximum speedup of 4.15x. This is partially due to the `teams` region now being SPMD mode which means that extra warps are not needed for the team main thread. We also see that a SIMD group size of 8 gives the best performance, likely due to it on average wasting fewer threads than other possible sizes due to the varying sparsity of the matrix, whereas the two-level parallel approach uses thread blocks of size 32, meaning that many threads may be idle.

SU3\_bench sees a maximum speedup of 1.3x using a SIMD group size of 4 threads, however this is only slightly better than 2 and 8 thread group sizes. These group sizes likely performed better than other options by reducing the number of idle threads given the size of the `simd` loop. Lastly, our benchmarking kernel sees a speedup of 2.13x with a SIMD group size of 16 threads.

These results highlight the possible improvement with SIMD parallelism. Not all codes will receive identical benefit from this optimization, but codes that cannot express efficient vector parallelism in a two-level parallel structure can see a speedup in the range of these example kernels.

To understand the performance difference of the different execution modes we analyze several kernels that include three parallelizable loops. The execution modes of

```

#pragma omp target teams distribute
for(int x = 1; x < nx-1; x++) {
    int x_ind = x*ny*nz;
    #pragma omp parallel for
    for(int y = 1; y < ny-1; y++) {
        int y_ind = y*stry;
        #pragma omp simd
        for(int z = 1; z < nz-1; z++) {
            int ind = x_ind + y_ind + z;
            B[ind] = (A[ind-strx] +
                    A[ind+strx] +
                    A[ind-stry] +
                    A[ind+stry] +
                    A[ind-strz] +
                    A[ind+strz])/6.0;
        }
    }
}

```

Figure 5.12: 3-dimensional heat diffusion OpenMP offloaded code.

these kernels can be adjusted between generic and SPMD mode by changing whether or not the loops are tightly-nested. This allows the kernels to be executed in all possible execution mode configurations, allowing us to analyze potential overheads from our implementation on near-identical kernels.

We have used three codes for this analysis: A simple three-dimensional heat diffusion kernel called `laplace3d`, and two kernels adapted from an OpenACC code called `muram_transpose` and `muram_interpol` [123]. We expect that these codes may see a small performance benefit from using a three-leveled parallel approach as it may slightly improve data-reuse, however, our main goal is to observe performance differences between generic and SPMD mode with our current implementation.

Fig. 5.11 shows the relative speedup of the different `simd` execution modes compared to the “No SIMD” version, which uses `teams` SPMD mode with two levels of parallelism. The number of teams and threads-per-team is kept consistent in these examples.

All regions executing in SPMD mode performs similarly to the “No SIMD” version, with `laplace3d` and `muram_interpol` seeing a marginal performance increase. Running in generic mode for either `teams` or `parallel` regions sees a  $\tilde{15}\%$  slowdown, which is the penalty for using the state machine and the extra synchronization it needs. Overall, from these results we conclude that our SIMD implementation produces little-to-no performance overhead when executing in SPMD mode. Additionally, the overhead accrued when executing in SIMD generic mode is comparable to the overhead of the `teams` generic mode.

## 5.5 Experimental Improvements to LLVM/OpenMP SIMD

This section will document the current status of the LLVM SIMD implementation and give insight of near-future goals.

### 5.5.1 Updated Code Generation for LLVM 17

With the full release of LLVM version 16 the development branch has moved to version 17, which brings a major change to the LLVM IR. Namely, pointers have been changed to no longer be typed, and instead all pointers are treated as generic pointers (i.e in terms of the C languages, these would be treated as “void” pointers.) From this there is no longer a distinction between different typed pointers, such as a floating pointer pointer vs. an integer pointer, or even a pointer to a complex struct. With these changes, and some other minor alterations made to the OpenMP IR Builder, our code branch has been rebased into LLVM 17, where it has begun the upstreaming process to be fully integrated into LLVM/Clang.

The upstream process for now includes the code generation portion which is outlined in Section 5.2. This also includes a suite of proper code generation test cases to verify the correctness of our “simd” code generation in conjuncture with other OpenMP directives and C/C++ constructs. After the completion of this step the full runtime will follow soon after, covering all content discussed in Section 5.3.

### 5.5.2 Loop Reductions

Loop reductions are a very common concept in directive-based programming models. Fig. 5.13 shows how a reduction is used; all threads alter the value of “sum”, which would normally cause problematic race conditions. The reduction allows for each thread to manage a local copy of “sum”, then at the end of the loop all threads will merge their values to create a singular global value. Fig. 5.14 shows one possible implementation of such a feature, where the “sum” variable is replaced by an array where each thread edits based on its thread ID. Then, the values are collapsed into a single value. In a more complex example, you would see the value initialization and collapsing also done in parallel using all threads.

```
#pragma omp target parallel for
for(.,) {
    float sum = 0;

    #pragma omp simd reduction(+:sum)
    for(int i = 0; i < N; i++) {
        sum += Ptr[i];
    }
}
```

Figure 5.13: An example of a summation reduction.

Alongside the new OpenMP loop API outlined in Section 5.2 comes a new API for handling reductions. For our work we will not create a new implementation for reduction, but instead will collaborate with the OpenMP community to ensure consistency between the model outlined here and the newly developed loop reductions.

### 5.5.3 Automatic conversion of SIMD-Generic code

As dicussed, SPMD mode is an important optimization for achieving high GPU performance using this OpenMP runtime. However, for SPMD mode to be applicable all threads must be able to safely execute the entire parallel region, assuring that all data needed is properly handled, and all threads choose the same branching paths. The work in [62] explains how generic mode code can be automatically converted to SPMD

```

#pragma omp target parallel for
for(..) {
    float globalSum = 0;
    float sum[omp_num_threads()];

    for(int i = 0; i < omp_num_threads(); i++)
        sum[i] = 0;

    #pragma omp simd
    for(int i = 0; i < N; i++) {
        sum[omp_get_thread()] += Ptr[i];
    }

    for(int i = 0; i < omp_num_threads(); i++)
        globalSum += sum[i];
}

```

Figure 5.14: An example of how a reduction could be implemented.

mode using LLVM optimization passes for OpenMP `teams` regions. In order for our SIMD work to allow for a similar automatic conversion, similar methods need to be applied.

```

#pragma omp target teams distribute
for(...) {
    #pragma omp parallel for
    for(...) {
        #pragma omp simd
        for(...) { }
    }
}

```

Figure 5.15: Tightly-nested parallel regions inherently allow for SPMD execution.

Fig. 5.15 shows an example of when SPMD mode can be used in our current model. Since the `simd` loop is tightly nested inside of the parallel region, ensuring that if all SIMD Workers execute the entire parallel region, there is no problematic code that disallows SPMD mode.

```

#pragma omp parallel for
for(int i = 0; i < N; i++) {
    float Val = some_computation();

    #pragma omp simd
    for(int j = 0; j < M; j++) {
        Ptr[i][j] = Val + some_other_computation();
    }
}

```

Figure 5.16: Extra code in parallel region causes side-effects and prevents SPMD mode.

Fig. 5.16 includes additional code within the parallel region. If we allow all threads to execute the region, we cannot guarantee that all threads receive the correct value of the “Val” variable. Only the value computed by the SIMD Main thread should be used, and thus the value of “Val” must be communicated in some way with the workers. In generic mode this process is handled by the variable sharing outlined in our implementation, but if we wish to instead use SPMD mode then some other mechanism must be used.

Similar to the implementation of [62], Fig. 5.17 shows a solution using new runtime functions for variable broadcasting. This allows the main thread to store the value of “Val” in a shared memory space before all worker threads would fetch it. Additionally, thread guarding is used to wrap the computation of “Val” so that only the main thread will execute that code. In a situation where many variables need to be broadcasted, to reduce communication all broadcasts could be delayed until just before they are needed, doing multiple broadcasts at once, instead of needing separate barriers for each.

Fig. 5.18 shows an alternative approach where all variables that would need to be shared are instead stored into shared memory. The main thread would handle all allocations at the start of the parallel region, and share those addresses with all of the workers. Then all threads will exclusively use those shared addresses during their computation. This allows for a single barrier to be done at the very start of a region,

```

#pragma omp parallel for
for(int i = 0; i < N; i++) {
    float Val;

    if (omp_get_thread_num() == 0) {
        Val = some_computation();
        // Store Val in shared memory buffer
        OMPRT_Broadcast(&Val, 4);
    }
    OMPRT_Barrier();
    // Read Val from the shared memory buffer
    OMPRT_FetchBroadcast(&Val, 4);
    OMPRT_Barrier();

    #pragma omp simd
    for(int j = 0; j < M; j++) {
        <Code that uses Val>
    }
}

```

Figure 5.17: Thread guarding and variable broadcasting to allow the code to execute in SPMD mode.

and would significantly reduce the amount of synchronization needed for more complex codes. The key drawback would be that replacing these variables with shared memory completely could negatively impact performance, depending on the code.

Fig. 5.19 shows one file implementation that has threads within the same SIMD group to share a shared memory stack. All threads call the “OMPRT\_SynchronizedSharedAlloca” function which will return the same shared memory pointer for each thread. However, right now, each thread has its own shared memory stack.

This feature is currently in development as a LLVM optimization pass that will introduce proper thread guarding and variable broadcasting to convert the SIMD-generic code into SIMD-SPMD. It is done as an optimization pass to alleviate extensive code generation from the front-end compiler to create a more independent and modular framework. These changes are expected to be upstreamed sometime after the initial runtime patch is completed.

```

#pragma omp parallel for
for(int i = 0; i < N; i++) {
    float *Val;

    if (omp_get_thread_num() == 0) {
        Val = (float*) OMPRT_AllocaShared(4);
        OMPRT_ShareVariableAddress(Val);
    }
    OMPRT_Barrier();
    OMPRT_GetSharedVariableAddress(&Val);
    OMPRT_Barrier();

    if (omp_get_thread_num() == 0) {
        *Val = some_computation();
    }

    OMPRT_Barrier();
    #pragma omp simd
    for(int j = 0; j < M; j++) {
        <Code that uses Val>
    }
}

```

Figure 5.18: An alternative version where instead of broadcasting variables a shared allocation is created and all threads will reference the same shared allocation.

#### 5.5.4 Towards AMD Support

LLVM supports AMD GPUs, though the implementation is generally considered as immature. With projects such as SPEC HPC [67] and SOLVVE [46, 63] and the upcoming Frontier supercomputer, we hope that a proper analysis of AMD performance in current LLVM will be available soon. This is to say that a full analysis of our implementation on AMD GPUs is currently difficult, but we can still focus on AMD compatibility. As a current status update, our implementation is being tested on the Frontier supercomputer and is currently in the stage of fully validating AMD code generation.

As mentioned, AMD does not allow for wavefront-level synchronization, making our SIMD-Generic model non-compatible, though SIMD-SPMD is supported. Once

```

#pragma omp parallel for
for(int i = 0; i < N; i++) {
    float *Val;

    Val = (float*) OMPRT_SynchronizedSharedAlloca(4);

    if (omp_get_thread_num() == 0) {
        *Val = some_computation();
    }

    OMPRT_Barrier();
    #pragma omp simd
    for(int j = 0; j < M; j++) {
        <Code that uses Val>
    }
}

```

Figure 5.19: Another alternative where a new runtime function that allows all threads to utilize the same shared memory stack. This reduces the need completely for additional synchronization.

automatic SPMDization is implemented, the range of possible codes in AMD would be greatly enhanced, though it will not cover all codes.

We have some ideas for circumventing the need for wavefront synchronization to allow our model to run on AMD in generic mode, but there is concern that these implementations would be very poor in performance. Instead we will focus on implementing more aggressive optimization passes to further increase the number of codes that can be run in SPMD mode and avoid generic mode entirely.

## Chapter 6

### CONCLUSION AND SUMMARY

Directive-based programming models are an attractive option for GPU programming due to its potential for architectural portability and familiarity to many programmers, utilizing the same languages that many modern HPC applications are written in. However, due to its nature, true performance portability will come at the cost of highly advanced compiler technologies to bring optimized code generation and access to advanced GPU features. Due to this complexity, compilers often lag behind in terms of feature completeness and performance, making the selection of viable compilers often limited for application developers.

The MURaM project showcases the difficulties of bringing real world applications to current generation HPC architectures, often requiring close collaboration with domain and HPC experts to bridge the gap between complex codes and algorithms and appropriate hardware usage and performance. In MURaM we have seen great benefit from the architecture portability provided by OpenACC bringing MURaM, and future improvements of MURaM, to both CPUs and GPUs with a singular code. We have also been able to integrate the optimized GPU FFT library, hEFFTe, utilizing the library for CPUs and GPUs seamlessly alongside our OpenACC code. Additionally, while we have showed our performance is not optimal, the GPU port is still achieving significant improvement over the original CPU version, with a single GPU computing over 2x faster than a full CPU node, and scaling beyond the limits of what was previously possible with the CPU only code.

These victories come at the cost of some very difficult to answer questions. (1): while the NVIDIA HPC compiler is mature enough to provide support for NVIDIA GPUs, there is currently not a great option for compilers supporting AMD or Intel

GPUs, which are being available in many recent supercomputers. While both GNU and Clang compilers may become viable options for AMD GPUs in the future, at this current moment MURaM is effectively locked to NVIDIA GPUs.

(2): since abstract, directive-based programming models heavily leverage advanced code generation and runtime libraries it is often difficult as an end user to control low-level details of a code's performance. In MURaM we face difficulties accurately predicting necessary GPU resource usage (i.e registers and shared memory) in various kernels, as the correlation between our high-level C++ code and the generated GPU output is obfuscated by the internal code generation of the compiler.

(3): GPU-specific features can be beneficial to many different kinds of codes, but are often inaccessible in purely directive-based models. In MURaM we have identified some advanced features for NVIDIA GPUs, which are incompatible with OpenACC, though OpenACC does provide interoperability options with CUDA and other GPU libraries, allowing us to explore these features in CUDA at the cost of those portions of code being locked to one architecture.

Fortunately, from the open-source nature of the LLVM compiler we have the opportunity to begin addressing some of these shortcomings of directive-based models. While the LLVM project focuses on the OpenMP programming model, various projects bringing OpenACC to LLVM aims to reuse much of the OpenMP implementation, drawing on the similarities of OpenMP and OpenACC. Thus, our contributions to LLVM are broadly applicable to a variety of models. Through these contributions we hope to improve the quality and feature completeness of the LLVM compiler infrastructure to impact the greater HPC ecosystem as a whole.

The primary contribution of the LLVM project is to fill a crucial gap in LLVM's GPU runtime by designing and implementing an execution model that gives finer control to the programmer over low-level GPU hardware mapping through the use of OpenMP's "simd" directive. This is an important directive which enables many important code optimizations, and increases LLVM/OpenMP's cross-comp ability with other models (i.e OpenACC, Kokkos, OpenMC). The design brings the CPU-centric

model of OpenMP onto GPU hardware through the use of the “generic” configuration, highlighting the difficulty of adapting these models to the distinct GPU architecture. Then, the GPU-centric “SPMD” configuration brings higher performance for a subset of codes.

The results show that the additional synchronization required for the “generic” model imposed a roughly  $\tilde{15}\%$  performance penalty for average kernels, while the optimized “SPMD” model shows no significant overhead. In kernels that benefit from this optimization, however, makes up for the performance loss and sees considerable performance improvement overall, with results ranging from 31%-215% total improvement depending on the code.

In conclusion, we have found that directive-based models are a viable means to parallel programming, utilizing common languages and defining parallelism in an abstract way without the need for specifying many hardware specifics. The performance achieved is very reasonable for  $\tilde{95}\%$  of the MURaM code, with the remaining 5% needing programmer intervention to achieve high performance. Through the development of real-world codes and close collaboration with these application developers, the models and compilers can be further improved to better suit those edge-cases and improve the HPC ecosystem as a whole.

## BIBLIOGRAPHY

- [1] <https://www.openmp.org/>.
- [2] <https://www.olcf.ornl.gov/summit/>.
- [3] <https://www.olcf.ornl.gov/frontier/>.
- [4] <https://www.openacc.org/>.
- [5] Hip : C heterogeneous-compute interface for portability. <https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/>.
- [6] Kripke. 2017. . <https://codesign.llnl.gov/kripke.php> [Online; accessed 15-August-2017].
- [7] Cuda zone. <https://developer.nvidia.com/cuda-zone>, journal=NVIDIA Developer, Sep 2019.
- [8] Intel® oneapi: Unified x-architecture programming model. <https://software.intel.com/en-us/oneapi>, Mar 2020.
- [9] PCAST — Pgi Compiler Assisted Software testing. <https://github.com/ORNL-CEES/Profugus>, 2020.
- [10] *OpenACC Programming and Best Practices Guide*. openacc-standard.org, May 2021.
- [11] Mps raven. <https://docs.mpcdf.mpg.de/doc/computing/raven-user-guide.html>, 2023.
- [12] MURaM, an open-source solar physics code. [https://github.com/NCAR/MURaM\\_main](https://github.com/NCAR/MURaM_main), 2023.
- [13] MURaM, availability of raw data. 10.5281/zenodo.8057228, 2023.
- [14] Ncar Derecho. [https://arc.ucar.edu/knowledge\\_base/74317833](https://arc.ucar.edu/knowledge_base/74317833), 2023.
- [15] Samantha V Adams, Rupert W Ford, M Hambley, JM Hobson, I Kavčič, Christopher M Maynard, Thomas Melvin, Eike Hermann Müller, S Mullerworth, AR Porter, et al. Lfric: Meeting the challenges of scalability and performance portability in weather and climate models. *Journal of Parallel and Distributed Computing*, 132:383–396, 2019.

- [16] Samuel F. Antão, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O’Brien. Offloading Support for OpenMP in Clang and LLVM. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–11, 2016.
- [17] Samuel F Antao, Alexey Bataev, Arpith C Jacob, Gheorghe-Teodor Bercea, Alexandre E Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, et al. Offloading support for openmp in clang and llvm. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–11. IEEE, 2016.
- [18] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. heffte: Highly efficient fft for exascale. In *ICCS 2020*, pages 262–275. Springer, 2020.
- [19] Randal S. Baker. Partisn on advanced/heterogeneous processing systems. 2014. <http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-20948> [Online; accessed 24-June-2014].
- [20] Gábor Dániel Balogh, Gihan R Mudalige, István Zoltán Reguly, SF Antao, and C Bertolli. Op2-clang: A source-to-source translator using clang/llvm libtooling. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 59–70. IEEE, 2018.
- [21] D Beckingsale. Umpire. <https://umpire.readthedocs.io/en/develop/>, 2017.
- [22] Carlo Bertolli, Samuel Antão, Gheorghe-Teodor Bercea, Arpith C. Jacob, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, and Kevin O’Brien. Integrating GPU support for OpenMP Offloading Directives into Clang. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 5:1–5:11, 2015.
- [23] Carlo Bertolli, Samuel Antão, Alexandre E. Eichenberger, Kevin O’Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. Coordinating GPU Threads for OpenMP 4.0 in LLVM. In *LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 12–21, 2014.
- [24] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM transactions on graphics (TOG)*, 22(3):917–924, 2003.
- [25] Claudio Bonati, Enrico Calore, Massimo D’Elia, Michele Mesiti, Francesco Negro, Francesco Sanfilippo, Sebastiano Fabio Schifano, Giorgio Silvi, and Raffaele Tripiccione. Portable multi-node lqcd monte carlo simulations using openacc. *International Journal of Modern Physics C*, 29(01):1850010, 2018.

- [26] J. P. Boris. A Physically Motivated Solution of the Alfvén Problem. *NRL Memorandum Report 2167*, November 1970.
- [27] C. Breu, H. Peter, R. Cameron, S. K. Solanki, D. Przybylski, M. Rempel, and L. P. Chitta. A solar coronal loop in a box: Energy generation and heating. *Astronomy and Astrophysics*, 658:A45, February 2022.
- [28] Holger Brunst, Sunita Chandrasekaran, Florina M Ciorba, Nick Hagerty, Robert Henschel, Guido Juckeland, Junjie Li, Verónica G Melesse Vergara, Sandra Wienke, and Miguel Zavala. First Experiences in Performance Benchmarking with the New SPEChpc 2021 Suites. In *International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 675–684, 2022.
- [29] Ronald M Caplan, Jon A Linker, Zoran Mikić, Cooper Downs, Tibor Török, and VS Titov. Gpu acceleration of an established solar mhd code using openacc. In *Journal of Physics: Conference Series*, volume 1225, page 012012. IOP Publishing, 2019.
- [30] B. G. Carlson. The numerical theory of neutron transport. In B. Alder and S. Fernbach, editors, *Methods in Computational Physics, Vol. 1*, page 1, 1963.
- [31] Barbara Chapman, Buu Pham, Charlene Yang, Christopher Daley, Colleen Bertoni, Dhruva Kulkarni, Dossay Oryspayev, Ed D’Azevedo, Johannes Doerfert, Keren Zhou, et al. Outcomes of OpenMP Hackathon: OpenMP Application Experiences with the Offloading Model (Part II). In *International Workshop on OpenMP (IWOMP)*, pages 81–95, 2021.
- [32] Feng Chen, Mark C. M. Cheung, Matthias Rempel, and Georgios Chintzoglou. Data-driven Radiative Magnetohydrodynamics Simulations with the MURaM Code. *Astrophysical Journal*, 949(2):118, June 2023.
- [33] Feng Chen, Matthias Rempel, and Yuhong Fan. Emergence of Magnetic Flux Generated in a Solar Convective Dynamo. I. The Formation of Sunspots and Active Regions, and The Origin of Their Asymmetries. *Astrophysical Journal*, 846(2):149, September 2017.
- [34] Feng Chen, Matthias Rempel, and Yuhong Fan. A Comprehensive Radiative Magnetohydrodynamics Simulation of Active Region Scale Flux Emergence from the Convection Zone to the Corona. *Astrophysical Journal*, 937(2):91, October 2022.
- [35] Feng Chen, Matthias Rempel, and Yuhong Fan. Eruption of a Magnetic Flux Rope in a Comprehensive Radiative Magnetohydrodynamic Simulation of flare-productive active regions. *arXiv e-prints*, page arXiv:2303.05405, March 2023.

- [36] M. C. M. Cheung, M. Rempel, G. Chintzoglou, F. Chen, P. Testa, J. Martínez-Sykora, A. Sainz Dalda, M. L. DeRosa, A. Malanushenko, V. Hansteen, B. De Pontieu, M. Carlsson, B. Gudiksen, and S. W. McIntosh. A comprehensive three-dimensional radiative magnetohydrodynamic simulation of a solar flare. *Nature Astronomy*, 3:160–166, November 2019.
- [37] M. C. M. Cheung, M. Rempel, A. M. Title, and M. Schüssler. Simulation of the Formation of a Solar Active Region. *Astrophysical Journal*, 720:233–244, September 2010.
- [38] Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. Experiences with sweep3d implementations in co-array fortran. *The Journal of Supercomputing*, 36(2):101–121, 2006.
- [39] Christopher Daley, Hadia Ahmed, Samuel Williams, and Nicholas Wright. A Case Study of Porting HPGMG from CUDA to OpenMP Target Offload. In *International Workshop on OpenMP (IWOMP)*, pages 37–51, 2020.
- [40] S. Danilovic, M. Rempel, M. van Noort, and R. Cameron. Observed and simulated power spectra of kinetic and magnetic energy retrieved with 2D inversions. *Astronomy and Astrophysics*, 594:A103, October 2016.
- [41] S. Danilovic, M. Schüssler, and S. K. Solanki. Probing quiet Sun magnetism using MURaM simulations and Hinode/SP results: support for a local dynamo. *Astronomy and Astrophysics*, 513:A1, April 2010.
- [42] Carlos A De Moura and Carlos S Kubrusly. The courant–friedrichs–lewy (cfl) condition. *AMC*, 10:12, 2013.
- [43] A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, and M. Wesenberg. Hyperbolic Divergence Cleaning for the MHD Equations. *Journal of Computational Physics*, 175:645–673, January 2002.
- [44] T. del Pino Alemán, J. Trujillo Bueno, J. Štěpán, and N. Shchukina. A Novel Investigation of the Small-scale Magnetic Activity of the Quiet Sun via the Hanle Effect in the Sr I 4607 Å Line. *Astrophysical Journal*, 863:164, August 2018.
- [45] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. Clacc: Translating openacc to openmp in clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 18–29. IEEE, 2018.
- [46] Jose Monsalve Diaz, Swaroop Pophale, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. Openmp 4.5 validation and verification suite for device offload. In *Evolving OpenMP for Evolving Architectures: 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26–28, 2018, Proceedings 14*, pages 82–95. Springer, 2018.

- [47] DOE. The compelling case for exascale computing. <https://www.exascaleproject.org/>, 2020.
- [48] Johannes Doerfert, Jose Manuel Monsalve Diaz, and Hal Finkel. The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In *International Workshop on OpenMP (IWOMP)*, volume 11718, pages 153–167, 2019.
- [49] Johannes Doerfert, Atemn Patel, Joseph Huber, Shilei Tian, Jose M Monsalve Diaz, Barbara Chapman, and Giorgis Georgakoudis. Co-Designing an OpenMP GPU Runtime and Optimizations for Near-Zero Overhead Execution. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022.
- [50] Douglas Doerfler, Christopher Daley, and USDOE. SU3\_bench: Lattice QCD SU(3) Matrix-Matrix Multiply Microbenchmark (SU3\_bench) v1.0, 4 2020.
- [51] H Carter Edwards and Christian R Trott. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Extreme Scaling Workshop (XSW)*, pages 18–24, 2013.
- [52] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.
- [53] Thomas M Evans, Wayne Joubert, Steven P Hamilton, Seth R Johnson, John A Turner, Gregory G Davidson, and Tara M Pandya. Three-dimensional discrete ordinates reactor assembly calculations on gpus. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States). Oak Ridge . . . , 2015.
- [54] G. H. Fisher, R. C. Canfield, and A. N. McClymont. Flare loop radiative hydrodynamics. V - Response to thick-target heating. VI - Chromospheric evaporation due to heating by nonthermal electrons. VII - Dynamics of the thick-target heated chromosphere. *Astrophysical Journal*, 289:414–441, February 1985.
- [55] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
- [56] T. I. Gombosi, J. R. Jokipii, J. Kota, K. Lorencz, and L. L. Williams. The telegraph equation in charged particle transport. *Astrophysical Journal*, 403:377–384, January 1993.
- [57] T. I. Gombosi, G. Tóth, D. L. De Zeeuw, K. C. Hansen, K. Kabin, and K. G. Powell. Semirelativistic Magnetohydrodynamics and Physics-Based Convergence Acceleration. *Journal of Computational Physics*, 177:176–205, March 2002.
- [58] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. Ieee, 2012.

- [59] B. V. Gudiksen, M. Carlsson, V. H. Hansteen, W. Hayek, J. Leenaarts, and J. Martínez-Sykora. The stellar atmosphere simulation code Bifrost. Code description and validation. *Astronomy and Astrophysics*, 531:A154, July 2011.
- [60] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C Schulthess. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.
- [61] Richard D Hornung and Jeffrey A Keasler. The raja portability layer: overview and status. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.
- [62] Joseph Huber, Melanie Cornelius, Giorgis Georgakoudis, Shilei Tian, Jose Manuel Monsalve Diaz, Kuter Dinel, Barbara M. Chapman, and Johannes Doerfert. Efficient Execution of OpenMP on GPUs. In *International Symposium on Code Generation and Optimization (CGO)*, pages 41–52, 2022.
- [63] Thomas Huber, Swaroop Pophale, Nolan Baker, Michael Carr, Nikhil Rao, Jaydon Reap, Kristina Holsapple, Joshua Hoke Davis, Tobias Burnus, Seyong Lee, et al. Ecp sollve: Validation and verification testsuite status update and compiler insight for openmp. *arXiv preprint arXiv:2208.13301*, 2022.
- [64] The OpenACC Application Programming Interface. OpenACC 3.1. <https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf>, 2020.
- [65] Arpith Chacko Jacob, Alexandre E Eichenberger, Hyojin Sung, Samuel F Antão, Gheorghe-Teodor Bercea, Carlo Bertolli, Alexey Bataev, Tian Jin, Tong Chen, Zehra Sura, et al. Efficient fork-join on gpus through warp specialization. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 358–367. IEEE, 2017.
- [66] H Jones, D Poliakkoff, and P Robinson. Chai. <https://github.com/LLNL/CHAI>, 2017.
- [67] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W Hwu, et al. Spec accel: A standard application suite for measuring hardware accelerator performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5*, pages 46–67. Springer, 2015.
- [68] Ian Karlin, Tom Scogland, Arpith C Jacob, Samuel F Antao, Gheorghe-Teodor Bercea, Carlo Bertolli, Bronis R de Supinski, Erik W Draeger, Alexandre E

- Eichenberger, Jim Glosli, et al. Early Experiences Porting Three Applications to OpenMP 4.5. In *International Workshop on OpenMP (IWOMP)*, pages 281–292, 2016.
- [69] E. Khomenko, N. Vitas, M. Collados, and A. de Vicente. Numerical simulations of quiet Sun magnetic fields seeded by the Biermann battery. *Astronomy and Astrophysics*, 604:A66, August 2017.
- [70] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*, pages 234–es. 2005.
- [71] P. B. Kunasz and L. Auer. Short characteristic integration of radiative transfer problems: formal solution in two-dimensional slabs. *J. Quant. Spectrosc. Radiat. Transfer*, 39:67, 1988.
- [72] Paul Kunasz and Lawrence H. Auer. Short characteristic integration of radiative transfer problems: Formal solution in two-dimensional slabs. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 39(1):67 – 79, 1988.
- [73] A Kunen, J Loffeld, A Black, R Chen, P Nowak, T Haut, T Bailey, P Brown, S Rennich, P Maginot, et al. Porting 3d discrete ordinates sweep algorithm in ardra to cuda. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2019.
- [74] E. Landi, G. Del Zanna, P. R. Young, K. P. Dere, and H. E. Mason. CHIANTIAN Atomic Database for Emission Lines. XII. Version 7 of the Database. *Astrophysical Journal*, 744:99, January 2012.
- [75] Jeff Larkin. Openacc best practices. [https://www.openacc.org/sites/default/files/inline-files/OpenACC\\_Programming\\_Guide\\_0.pdf](https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf), 2015.
- [76] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.
- [77] J. Leenaarts, M. Carlsson, and L. Rouppe van der Voort. The Formation of the H $\alpha$  Line in the Solar Chromosphere. *Astrophysical Journal*, 749(2):136, April 2012.
- [78] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligoeki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 129–148. Springer, 2014.

- [79] J. W. Lord, R. H. Cameron, M. P. Rast, M. Rempel, and T. Roudier. The Role of Subsurface Flows in Solar Surface Convection: Modeling the Spectrum of Supergranular and Larger Scale Flows. *Astrophysical Journal*, 793:24, September 2014.
- [80] Z. Magic, R. Collet, M. Asplund, R. Trampedach, W. Hayek, A. Chiavassa, R. F. Stein, and Å. Nordlund. The Stagger-grid: A grid of 3D stellar atmosphere models. I. Methods and general properties. *Astronomy and Astrophysics*, 557:A26, September 2013.
- [81] Stefan Maintz and Markus Wetzstein. Strategies to accelerate vasp with gpus using open acc. *Proceedings of the Cray User Group*, 2018.
- [82] Matt Martineau, Simon McIntosh-Smith, Carlo Bertolli, Arpith C. Jacob, Samuel F. Antao, Alexandre Eichenberger, Gheorghe-Teodor Bercea, Tong Chen, Tian Jin, Kevin O’Brien, Georgios Rokos, Hyojin Sung, and Zehra Sura. Performance Analysis and Optimization of Clang’s OpenMP 4.5 GPU Support. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 54–64, 2016.
- [83] Juan Martínez-Sykora, Jorrit Leenaarts, Bart De Pontieu, Daniel Nóbrega-Siverio, Viggo H. Hansteen, Mats Carlsson, and Mikolaj Szydlarski. Ion-neutral Interactions and Nonequilibrium Ionization in the Solar Chromosphere. *Astrophysical Journal*, 889(2):95, February 2020.
- [84] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: automatic annotation for data parallelism and offloading. volume 14, pages 1–25, 2017.
- [85] Andrea Mignone, G Bodo, S Massaglia, Titos Matsakos, O Tesileanu, C Zanni, and Anthony Ferrari. Pluto: a numerical code for computational astrophysics. *The Astrophysical Journal Supplement Series*, 170(1):228, 2007.
- [86] A. Nordlund. Numerical simulations of the solar granulation. I - Basic equations and methods. *Astronomy and Astrophysics*, 107:1–10, 1982.
- [87] Å. Nordlund, R. F. Stein, and M. Asplund. Solar Surface Convection. *Living Reviews in Solar Physics*, 6:2, April 2009.
- [88] NVIDIA. Cuda. <https://developer.nvidia.com/cuda-zone>, 2017.
- [89] NVIDIA. Nvidia hpc sdk. <https://www.pgroup.com/index.htm>, 2020.
- [90] Güray Özen, Simone Atzeni, Michael Wolfe, Annemarie Southwell, and Gary Klimowicz. OpenMP GPU Offload in Flang and LLVM. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–9, 2018.

- [91] Simon J Pennycook, Jason D Sewall, and Jeff R Hammond. Evaluating the Impact of Proposed OpenMP 5.0 Features on Performance, Portability and Productivity. In *International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 37–46, 2018.
- [92] Swaroop Pophale, Dossay Oryspayev, B Chapman, B Pham, C Yang, C Daley, C Bertoni, D Kulkarni, E D’Azevedo, H He, et al. Outcomes of OpenMP Hackathon: OpenMP Application Experiences with the Offloading Mode. Technical report, Brookhaven National Lab.(BNL), Upton, NY (United States), 2021.
- [93] Kenneth G Powell, Philip L Roe, Timur J Linde, Tamas I Gombosi, and Darren L De Zeeuw. A solution-adaptive upwind scheme for ideal magnetohydrodynamics. *Journal of Computational Physics*, 154(2):284–309, 1999.
- [94] Mark P Rast, Nazaret Bello González, Luis Bellot Rubio, Wenda Cao, Gianna Cauzzi, Edward DeLuca, Bart De Pontieu, Lyndsay Fletcher, Sarah E Gibson, Philip G Judge, et al. Critical science plan for the daniel k. inouye solar telescope (dkist). *Solar Physics*, 296(4):70, 2021.
- [95] M. Rempel. Numerical Simulations of Quiet Sun Magnetism: On the Contribution from a Small-scale Dynamo. *Astrophysical Journal*, 789:132, July 2014.
- [96] M. Rempel. Numerical Simulations of Sunspot Decay: On the Penumbra–Evershed Flow–Moat Flow Connection. *Astrophysical Journal*, 814:125, December 2015.
- [97] M. Rempel. Extension of the MURaM Radiative MHD Code for Coronal Simulations. *Astrophysical Journal*, 834:10, January 2017.
- [98] M. Rempel. Small-scale Dynamo Simulations: Magnetic Field Amplification in Exploding Granules and the Role of Deep and Shallow Recirculation. *Astrophysical Journal*, 859:161, June 2018.
- [99] M. Rempel, M. Schüssler, R. H. Cameron, and M. Knölker. Penumbral Structure and Outflows in Simulated Sunspots. *Science*, 325:171, July 2009.
- [100] Matthias Rempel, Georgios Chintzoglou, Mark C. M. Cheung, Yuhong Fan, and Lucia Kleint. Comprehensive radiative MHD simulations of flares above collisional polarity inversion lines. *arXiv e-prints*, page arXiv:2303.05299, March 2023.
- [101] Viacheslav M. Sadykov, Irina N. Kitiashvili, Alexander G. Kosovichev, and Alan A. Wray. Connecting Atmospheric Properties and Synthetic Emission of Shock Waves Using 3D RMHD Simulations of the Quiet Sun. *Astrophysical Journal*, 909(1):35, March 2021.
- [102] Sunil Sathe. Accelerating the ANSYS fluent r18.0 radiation solver with openacc. <https://tinyurl.com/yacxh5g7>, 2016.

- [103] Will Sawyer, Guenther Zaengl, and Leonidas Linardakis. Towards a multi-node openacc implementation of the icon model. In *EGU General Assembly Conference Abstracts*, volume 16, 2014.
- [104] Thomas Schad and Gabriel Dima. Forward Synthesis of Polarized Emission in Target DKIST Coronal Lines Applied to 3D MURaM Coronal Simulations. , 295(7):98, July 2020.
- [105] Robert Searles, Sunita Chandrasekaran, Wayne Joubert, and Oscar Hernandez. Abstractions and directives for adapting wavefront algorithms to future architectures. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, page 4. ACM, 2018.
- [106] Robert Searles, Sunita Chandrasekaran, Wayne Joubert, and Oscar Hernandez. Mpi+ openacc: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems. *Computer Physics Communications*, 236:176–187, 2019.
- [107] A. P. Snodin, A. Brandenburg, A. J. Mee, and A. Shukurov. Simulating field-aligned diffusion of a cosmic ray gas. , 373:643–652, December 2006.
- [108] The OpenMP API specification for parallel programming. OpenMP 5.1. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf/>, 2020.
- [109] Intel® Xeon® Gold 6148 Processor Product Specifications. Intel® specification. <https://ark.intel.com/content/www/us/en/ark/products/120489/intel-xeon-gold-6148-processor-27-5m-cache-2-40-ghz.html>, 2020.
- [110] L. Spitzer. *Physics of Fully Ionized Gases*, Interscience Publishers, New York. Interscience Publishers, New York, 1962.
- [111] O. Steiner, F. Calvo, R. Salhab, and G. Vigeesh. CO5BOLD for MHD: progresses and deficiencies . *Mem. Societa Astronomica Italiana*, 88:37, January 2017.
- [112] Felix Thaler, Stefan Moosbrugger, Carlos Osuna, Mauro Bianco, Hannes Vogt, Anton Afanasyev, Lukas Mosimann, Oliver Fuhrer, Thomas C Schulthess, and Torsten Hoefler. Porting the cosmo weather model to manycore cpus. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–11, 2019.
- [113] Aidan P Thompson, Laura P Swiler, Christian R Trott, Stephen M Foiles, and Garritt J Tucker. Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials. *Journal of Computational Physics*, 285:316–330, 2015.

- [114] Shilei Tian, Jon Chesterfield, Johannes Doerfert, and Barbara Chapman. Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1. In *International Workshop on OpenMP*, 2021.
- [115] Shilei Tian, Johannes Doerfert, and Barbara M. Chapman. Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 41–56, 2020.
- [116] E. Tiotto, B. Mahjour, W. Tsang, X. Xue, T. Islam, and W. Chen. OpenMP 4.5 Compiler Optimization for GPU Offloading. *IBM Journal of Research and Development*, 64(3/4):14:1–14:11, 2020.
- [117] J Trujillo Bueno and P Fabiani Bendicho. A novel iterative scheme for the very fast and accurate solution of non-lte radiative transfer problems. *The Astrophysical Journal*, 455:646, 1995.
- [118] Miikka S Väisälä, Johannes Pekkilä, Maarit J Käpylä, Matthias Rheinhardt, Hsien Shang, and Ruben Krasnopolsky. Interaction of large-and small-scale dynamos in isotropic turbulent flows from gpu-accelerated simulations. *The Astrophysical Journal*, 907(2):83, 2021.
- [119] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.
- [120] A. Vögler and M. Schüssler. A solar surface dynamo. *Astronomy and Astrophysics*, 465:L43–L46, April 2007.
- [121] A. Vögler, S. Shelyag, M. Schüssler, F. Cattaneo, T. Emonet, and T. Linde. Simulations of magneto-convection in the solar photosphere. Equations, methods, and results of the MURaM code. *A&A*, 429:335–351, January 2005.
- [122] Eric Wright, Damien Przybylski, Matthias Rempel, Cena Miller, Supreeth Suresh, Shiquan Su, Richard Loft, and Sunita Chandrasekaran. Refactoring the MP-S/university of chicago radiative mhd (muram) model for gpu/cpu performance portability using openacc directives. In *PASC*, pages 1–12, 2021.
- [123] Eric Wright, Damien Przybylski, Matthias Rempel, Cena Miller, Supreeth Suresh, Shiquan Su, Richard Loft, and Sunita Chandrasekaran. Refactoring the MPS/University of Chicago Radiative MHD (MURaM) model for GPU/CPU performance portability using OpenACC directives. In *Platform for Advanced Scientific Computing Conference (PASC)*, pages 1–12, 2021.
- [124] Zhifeng Yang, Milton Halem, Richard Loft, and Supreeth Suresh. Accelerating mps-a model radiation schemes on gpus using openacc. *AGUFM*, 2019:A11A–06, 2019.

- [125] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann. Alpaka – an abstraction library for parallel kernel acceleration. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 631–640, May 2016.