RICE UNIVERSITY

OKL: A Unified Language for Parallel Architectures

by

David Medina

A Thesis Submitted
in Partial Fulfillment of the
requirements for the degree

Doctor of Philosophy

Approved, Thesis Committee:

Timothy Warburton, Chair
Professor of Computational and Applied Mathematics

Beatrice Riviere
Professor of Computational and Applied Mathematics

Danny C. Sorensen
Noah Harding Professor of Computational and Applied Mathematics

---

Keith D. Cooper
L. John and Ann H. Doerr Professor of Computational Engineering

Houston, Texas

May, 2015

Abstract

OKL: A Unified Language for Parallel Architectures

David Medina

Rapid evolution of computer processor architectures has spawned multiple programming languages and standards. This thesis strives to address the challenges caused by fast and cyclical changes in programming models. The novel contribution of this thesis is the introduction of an abstract unified framework which addresses portability and performance for programming manycore devices. To test this concept, I developed a specific implementation of this framework called OCCA. OCCA provides evidence that it is possible to achieve high performance across multiple platforms.

The programming model investigated in this thesis abstracts a hierarchical representation of modern manycore devices. The model at its lowest level adopts native programming languages for these manycore devices, including serial code, OpenMP, OpenCL, NVIDIA's CUDA, and Intel's COI. At its highest level, the ultimate goal is a high level language that is agnostic about the underlying architecture. I developed a multiply layered approach to bridge the gap between expert "close to the metal" low-level programming and novice-level programming. Each layer requires varying degrees of programmer intervention to access low-level features in device architectures.

I begin by introducing an approach for encapsulating programming language features, delivering a single intermediate representation (OCCA IR). Built above the OCCA

IR are two kernel languages extending the prominent programming languages C and Fortran, the OCCA kernel language (OKL) and the OCCA Fortran language (OFL). Additionally, I contribute two automated approaches for facilitating data movement and automating translations from serial code to OKL kernels.

To validate OCCA as a unified framework implementation, I compare performance results across a variety of applications and benchmarks. A spectrum of applications have been ported to utilize OCCA, showing no performance loss compared to their native programming language counterparts. In addition, a majority of the discussed applications show comparable results with a single OCCA kernel.

Acknowledgements

I'm grateful to my advisor Professor Tim Warburton for his guidance and providing me a great working environment during my time at Rice University. With his supervision, teachings, and collaboration, I was able to gain the valuable knowledge and experience required for this thesis work. Through him, I've had the opportunity to meet many colleagues from our research group and collaborators in academia, national labs, and industry. Likewise, I'm thankful for the faculty at CAAM for teaching the fundamentals required for my work. I would like to thank my committee, Prof. Riviere, Prof. Symes, Prof. Sorensen, and Prof. Cooper, for their advice, feedback, and support. I also want to mention my great thanks to Dr. Amik St-Cyr for his mentoring and suggestions through my internships.

This would also not have been made possible without important people outside my academic life. In particular, I would like to thank my wife Xiong for her constant help, understanding, and coffee. Many colleagues are also very good friends outside of school, helping me out when I need it. I'm lucky to have good friends, notably Rajesh Gandham for being my awesome officemate and collaborator through many projects and Jesse Chan for his constant help.

Table of Contents

Nomenclature

1 Abbreviations

CPU	Central Processing Unit
FPGA	Field Programmable Gate Arrays
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HPC	High Performance Computing
IR	Intermediate Representation
NUMA	Non-Uniform Memory Access
RAM	Random Access Memory
SIMD	Single-Instruction Multiple-Data

2 Languages and Standards

Heterogeneous Computing	Combined use of CPUs and accelerators for computational purposes
OpenMP	Open standard for programming multithreaded systems through directives.

OpenMP 4.0	Open standard for programming heterogeneous systems through directives.
OpenACC	Open standard for programming GPUs through directives.
OpenCL	Open standard for programming heterogeneous systems by unifying parallel device architectures such as CPUs, GPUs, and FPGAs.
CUDA	NVIDIA's proprietary language for programming NVIDIA GPUs
COI	Intel's Coprocessor Offload Infrastructure used to program the Xeon Phi
OKL	OCCA Kernel Language (C-based)
OFL	OCCA Fortran Language (Fortran-based)

3 GPU Terminology

GPGPU	General Purpose Graphics Processing Unit, GPUs used for general purpose computing
Host	Processor(s) running an application, usually by a CPU
Device	Hardware targeted for offload computations to such as a multicore CPU, GPU, or other accelerator
Kernel	GPU Function

(CUDA) Thread	Manages work in the GPU at the finest granularity level
(CUDA) Block	Group of CUDA threads executing concurrently
(OpenCL) Work-item	Manages work in the GPU at the finest granularity level. Synonymous with threads in CUDA
(OpenCL) Work-group	Group of OpenCL work-items executing concurrently. Synonymous with blocks in CUDA
Global memory	Memory located on the device with the longest latency in the device memory hierarchy
Shared memory	Memory co-located with the processor's cache, accessible by all FPUs on a compute unit
Register memory	Processing unit's register file, smallest latency in the device memory hierarchy

4 OCCA Terminology

Mode	Target language/specification for a device such as Serial, OpenMP, OpenCL, and CUDA
Loop Tag	Fourth clause in a for-loop indicating the type of parallelism to be taken by the for-loop
Outer Loops	For-loops with an `outer` tag, parallelized by threads (CPU) or work-groups (GPU)
Inner Loops	For-loops with an `inner` tag, able to be vectorized (CPU) or work concurrently (GPU)

Shared memory	Shared memory when using a GPU architecture, otherwise a regular buffer
Exclusive memory	Register memory unique to a inner-loop iteration, value can jump inner-loop scopes

1 Introduction

With the collapse of single-processor computing in the high performance computing (HPC) field, a wave of parallel-computing has emerged. Alongside the paradigm shift of multiprocessor computing, several hardware architectures tailored for parallel computing have risen and fallen. Keeping confidence in numerical methods and their implementations while engaging in this undetermined and changing environment became a motivation for the work provided in this thesis.

The novelty presented in this thesis is the programming model used to abstract modern manycore devices. Multiple layers in this model were developed, including the representation of low-level features present in manycore architectures towards a high-level model automating. These concepts were implemented throughout the OCCA project to unify commonalities found across prominent architectures in the HPC community. We start with an introduction on the past, present and projected future architectures and their programming models to set the background for the thesis work. Although the work shown in this thesis generalizes current many-core models, knowledge in architecture advancements provides insight on identifying recurring features across generations of hardware designs.

1.1 Overview of Computational Architectures

We begin by discussing the general-purpose central processing unit (CPU) and its architectural improvements through the last few decades. Following is an overview

on graphics processing units (GPU) and strides to the general-purpose programming model we see today in CUDA and OpenCL.

1.1.1 Central Processing Units

Floating point units (FPUs) began as co-processors, replacements for software-designed floating-point operations. Central processing units would offload floating-precision work to external FPUs until the FPUs became integrated with the processor. Transistors per area grew exponentially (modeled by Moore's Law), increasing peak FLOPS exponentially (Schaller 1997). Once power limitations became apparent, voltage and currents were scaled proportionally to transistor sizes (Dennard scaling) to control power utilization and maintain the transistor density rate noted by Moore's Law (Dennard et al. 1974). Although Dennard scaling made the use of the transistor count feasible, performance scaling was reduced around the year 2000, requiring architectural improvements to continue scaling (Dally 2009).

The use of Beowulf clusters (computers networked together) gained traction as a method to sidestep the limitations set by a single processing unit. A similar philosophy was then applied to central processing units, resulting in multicore processor designs with each core containing an independent instruction scheduler. Processors with 2-8 cores and motherboards containing 2-4 processor sockets became the industry standard. Together with increasing core-count, single-instruction multiple-data (SIMD) cores became common on general-purpose and performance-based processing units. This idea of reusing an instruction on multiple data entries is not new and has been implemented since the CRAY-1 in 1976 (Russell 1978). Current SIMD vectorization units not only allow for instruction reuse but decrease overall fetching latency due to vector instructions operating on a contiguous data segment, usually requiring page alignment of memory accesses.

While many aspects contributed to gains in computation performance, bandwidth and latency has improved at different rates. The processor clock speeds increased at a rate to which memory systems were incapable of continuously providing data for, hence a computational bottleneck stemming from the memory bandwidth. Apart from pure bandwidth increases, a hierarchy of cache levels were introduced to predict and prefetch data directly for processing elements to avoid excessive data movement. Transfers between memory and cache are automated to fetch fixed-sized memory segment cache lines into cache following a non-cached memory access (cache miss). Likewise, cache line evictions are also automated while maintaining coherence across cores.

Alongside with CPU architecture improvements, co-processors tailored for specific computational tasks were also developed. Graphics-processing and embarrassingly-parallel computations are currently common uses for co-processors while the central processor is used for general-purpose computations. Well known co-processors include graphics processing units (GPUs), field programmable gate arrays (FPGAs), Intel's Xeon Phi, and IBM's Cell architecture. However, there are also architectures aiming for performance with low-power consumption, such as massively parallel processor arrays (MPPAs).

1.1.2 Graphics Processing Units

Graphics cards were developed due to the increasing demand for improved graphics in video games. The architectural design for graphics cards was based on the linear algebra operations used in 3D rendering. Rather than relying on the general-purpose CPU, computations for rendering were offloaded onto graphics cards. Briefly stated, programming practices for graphics rendering queue batches of identical instructions which make use of an embarrassingly parallel architecture.

Programming graphics cards made use of a fixed pipeline, splitting work into

different stages for updating vertices and managing individual fragments. With the potential of using graphics cards for general purpose computing, the Brook language was developed to reduce the complexity of using the fixed pipeline for general purpose computations (Buck et al. 2004). In 2007, Brook inspired CUDA, NVIDIA's proprietary language, for using their graphics cards as general purpose GPUs (GPGPUs). However, CUDA vendor-locks application development to NVIDIA GPUs for hardware acceleration (Bodin and Bihan 2009).

To address this issue, the Khronos group, an open standards consortium, released specifications for the OpenCL standard for programming heterogeneous platforms including: CPUs, GPUs, Intel's Xeon Phi and field-programmable gate arrays (FPGAs). In 2008, a year after CUDA's initial release, NVIDIA and Advanced Micro Devices (AMD) presented their first functional OpenCL demo in Siggraph Asia 2008 (Munshi 2008). Presently, programming for these accelerators requires either CUDA or OpenCL, or approaches which make use of these languages.

1.2 Programming Next-Generation Architectures

With the rise of accelerator models, it is not known which architectures will endure. For example, the IBM's Cell architecture, that powered the Playstation 3 and in the first petaflop-capable machine, was put on hiatus less than a decade after development (Campbell 2009). The uncertainty in lasting architectures could cause applications to become defunct in the future. This systemic risk for software development presents a challenge for code longevity. I will briefly discuss programming differences between CUDA and OpenCL together with their respective advantages and disadvantages.

1.2.1 Programming with CUDA and OpenCL

The similarities between CUDA and OpenCL become evident in their programming model but their popularity in use differ. NVIDIA releases their own compiler wrapper, nvcc, to allow CUDA kernels to be embedded in the application code, while OpenCL separates host code (application code) with the device code (kernels), which steepens the learning curve. The simplicity of combining host and device code is one of the reasons CUDA is somewhat more popular than OpenCL which separates device kernels (functions written for the GPU) from application code. Despite this, OpenCL is favorable largely due to its open standard which has been implemented for a large range of architectures. Fortunately, kernels written in CUDA can be translated to OpenCL without much effort as seen in AMD's "Porting into OpenCL" (Advanced Micro Devices 2013) site; however, the same could not be said about the host-device interaction. As seen in code listing 1.1, CUDA allows mixing host and device code; whereas OpenCL requires separate files.

```
// (1) CUDA: Embedded file holding type
//           definitions and functions

class vector3 {
public:
  float x, y, z;

  __host__ __device__ float normalize(){
    const float norm = sqrt((x*x) +
                            (y*y) +
                            (z*z));

    x /= norm;
    y /= norm;
    z /= norm;
  }
};


__global__
void normalizeArray(vector3 *array,
                    const int entries){

  const int n = threadIdx.x +
                (blockIdx.x * blockDim.x);

  if(n < entries)
    array[n].normalize();
}
```

```
// (2) OpenCL: External file holding type
//             definitions and functions

struct vector3 {
  float x, y, z;
};

float normalize(vector3 *v){
  const float x2 = (v->x)*(v->x);
  const float y2 = (v->y)*(v->y);
  const float z2 = (v->z)*(v->z);

  const float norm = sqrt(x2 + y2 + z2);

  v->x /= norm;
  v->y /= norm;
  v->z /= norm;
}

__kernel
void normalizeArray(vector3 *array,
                    const int entries){

  const int n = get_global_id(0);


  if(n < entries)
    normalize( &(array[n]) );
}
```

Listing 1.1: Examples of object-oriented programming in (1) CUDA and (2) OpenCL

1.2.2 Performance Differences Between CUDA and OpenCL

There are striking differences between NVIDIA's proprietary language and runtime API compared with OpenCL's low-level API (with flexibility on platform-choice). Hence, approaches have been taken to combine advantages from each language. The benefits for developing multithreaded GPGPU applications in CUDA inspired the question whether a translation between CUDA and OpenCL would be possible and useful. Various papers have benchmarked well-known suites on both CUDA and OpenCL, summarizing that comparable performance can be achieved in both, but depend on the hardware and optimizations (Karimi et al. 2010, Fang et al. 2011, Du et al. 2012, Wang et al. 2014). For example, fluctuations in performance can be seen across NVIDIA's

CUDA, NVIDIA's OpenCL and AMD's OpenCL platforms running the same kernel, with one-to-one translations, due to varying optimizations found on the different platforms. A similar occurrence is found when comparing with performance across compilers on the same application code.

Different approaches have been taken to address portability and performance when uniting these two frameworks. Some approaches try to minimize adjustments to legacy codes to prevent refactoring on the host-device model by using directives, inspiring standards such as OpenMP and OpenACC (Dolbeau et al. 2007). Other approaches accept the host-device programming model and strive to combine CUDA's language features with OpenCL's flexibility, motivating source-to-source solutions. Lastly, we discuss libraries which hide the underlying languages by providing specialized routines, such as BLAS (Basic Linear Algebra Subprograms).

1.3 Literature Review

The next chapter describes in detail the initial development of OCCA (Medina 2014), including the OCCA IR kernel language. This section provides a spectrum of approaches for achieving similar code portability throughout a range of hardware. Analyzing these present and past approaches motive the choices taken in the development of the OCCA IR. These approaches are grouped into three categories: the use of directives which allow the compiler to provide code transformations; source-to-source transformations between programming languages; and lastly, providing a set of tailored operations, hand-coded for each provided backend.

1.3.1 Directive Approach

The first approach we will discuss makes use of directives in the form of `#pragma`'s or comment regions to shift code manipulation to the compiler. OpenMP is a promi-

nent example of a specification which uses directives. Rather than using a low-level management of threads, OpenMP eases multithreading development by automating thread management.

With the introduction of GPUs used in high performance computing (HPC), it was of interest to achieve the simplification seen in multithreading for these new accelerators. In 2009, *hi*Cuda was one of the first directive-based project to address this issue, handling device tasks such as memory management and kernel generation (Han and Abdelrahman 2009, Han and Abdelrahman 2011). In the same year, a Star Superscalar (StarSs) programming model extension was proposed, anticipating an extension of OpenMP to introduce multithreaded programming on GPGPUs (Ayguadé et al. 2009). These projects were followed by a large number of standards and standard proposals, resulting in OpenACC and OpenMP 4.0. OpenACC was the first commercial release, supported at the time by compilers from Cray, NVIDIA, CAPS and PGI. Open-source projects such as openMPC (Lee and Eigenmann 2010) and IPMACC (Lashgar et al. 2014) gave open-source alternatives to using OpenACC in application codes. Meanwhile, OpenMP received a few proposals for its 4.0 specification (Ferrer et al. 2011, Duran et al. 2011). There are currently few compilers supporting OpenMP 4.0 due to the specification being relatively new.

While there have been promising results, it should be noted that several benchmarks were embarrassingly parallel and could be described with simple access patterns (Han and Abdelrahman 2009, Wienke et al. 2012, Herdman et al. 2012, Reyes et al. 2012). It was of interest to not only note the performance, but the amount of labor required to enable GPGPU support on traditional CPU applications through directives as opposed to refactorization with OpenCL and/or CUDA (Wienke et al. 2012, Wang et al. 2014). The paper by (Wienke et al. 2012) showed a drop of 90% performance when using an OpenACC implementation of their non-linear optimization algorithm

when using naive placements of `#pragmas` compared to OpenCL, and a 60%-80% drop when restructuring the parallel loops. Similarly, (Wang et al. 2014) reported a 20%-40% performance drop when comparing on a Fermi GPU, Kepler GPU, and the Xeon Phi for their finite difference implementation. However, the restructuring required for the naive OpenACC implementations in both papers required 1%-2% line changes when compared to their OpenCL and CUDA implementation counterparts. The tailored OpenACC kernels which achieved 40%-60% relative performance, line changes were still less than 45% when comparing with OpenCL and CUDA. The fact that there are missing or immature elements required for GPU optimization cannot be hidden, including:

- Access to manually manipulate shared memory
- Manual memory retainment between the host and device
- Forced global synchronization across certain parallel regions
- Light asynchronous support
- Limited use of functions (required to be inline but not guaranteed to work)

Nevertheless, while some algorithms could be deemed inefficient with this approach, others generate comparable CUDA/OpenCL kernels as to their hand-coded counterparts; an important factor when balancing developer time with performance. There are tools to attempt different optimizations with code transformations; for example, some machine-learning-based methods automatically test for performance improvements, but their use in practice still needs investigation (Grewe et al. 2013, Lee and Vetter 2014).

To summarize, the use of directives puts greater value on the learning curve and development time as opposed to computational performance. Preliminary results have been positive for simple kernel examples, but the specification appears to be lacking

low-level features for leveraging the full targeted architecture. In contrast with the directive approach, the following discussed approach makes use of the advantages in CUDA, for it's simpler programming language, and OpenCL, for it's portability across multiple architectures.

1.3.2 Source-to-source Approach

The second approach discussed is a source-to-source approach, where code transformation occurs prior to compilation or at the compilation stage. Although not as simple as the directive approach, this approach allows users to take advantage of CUDA's language extensions which provide many features to ease GPGPU programming. For example, NVIDIA has released toolkits for several optimized linear algebra routines, such as cuBLAS and MAGMA, which facilitate many dense and sparse linear operations; something difficult to enable with the directive approach.

However, as previously mentioned, using CUDA limits the user to use NVIDIA GPUs for accelerating codes. Two projects addressed this issue by creating source-to-source translators from CUDA to OpenCL to achieve platform flexibility: CU2CL (Martinez et al. 2011, Gardner et al. 2013) and SWAN (Harvey and De Fabritiis 2011). The SWAN project came from industry and apparently stopped updating after the introduction of NVIDIA's Fermi architecture. Meanwhile CU2CL, an academic prototype, currently supports a core portion of later versions of CUDA which have added many useful features.

A second approach to source-to-source conversion is based on analyzing the intermediate assembly produced by CUDA and converting it to assembly supported by other platforms. GPU Ocelot (Diamos et al. 2010, Farooqui et al. 2011) and INSIEME (Jordan et al. 2013) have taken this assembly approach towards platform flexibility. A possible disadvantage arises when porting assembly to non-NVIDIA platforms or even

distinct hardware due to architecture-dependent optimizations applied on distinct architectures.

Aside from pure CUDA-to-OpenCL and assembly transformations, a few other translation projects have been released. Par4All is a project that was developed to automatically detect loops from C and Fortran and transform them at compile-time into OpenMP, OpenCL and CUDA (Amini et al. 2012, Ventroux et al. 2012). Similarly, the project discussed in (Anderson 2014) is a prototype to convert Python code into OpenCL kernels in a more controlled environment. These approaches resemble a black-box due to automatic conversion, which may be overly intrusive and could create issues interfacing with other libraries.

To summarize, the source-to-source approaches encounters difficulties due to the transformation scope and the rapidly changing specifications. Similar to requirements of directive-based approaches, which require access to the complete project source code, source-to-source projects could require a full application transformation which may be impractical in industrial codes. The last approach tries to prevent the global-scope requirement and is robust to changes in specifications. By creating a wrapper around CUDA and OpenCL, the next approach creates tailored frameworks which prioritize in easy-to-use routines without sacrificing high performance.

1.3.3 Wrapper Approach

The "wrapper approach" focuses on creating libraries and objects wrappers. By developing and implementing tailored algorithms for supported architectures, high performance can be achieved while maintaining platform flexibility. While CUDA has several exclusive mathematical libraries, such as cuBLAS and MTL4, there are also libraries available for OpenCL, such as VexCL/ViennaCL (Demidov 2012, Demidov et al. 2013). AMD has also released the open-source OpenCL counterpart, clBlas, as

well as Bolt, the standard template library (STL) styled library for executing common algorithms on vector structures (Rogers and FELLOW 2013). Other industrial approaches include Intel's Thread Building Blocks (TBB) and Microsoft's specification for C++ AMP, a collection of highly templated libraries for executing parallel tasks (Pheatt 2008, Gregory and Miller 2012).

SkePU is another C++ library which focuses on the map-reduce model for kernel generation (Enmyren and Kessler 2010, Dastgeer et al. 2011). By providing "skeleton" code, users have limited templates with which to create kernels. While SkePU can generate OpenCL and CUDA kernels, the code skeletons offered to generated kernels with require simple access patterns.

The motivation for platform flexibility has increased with the growing number of architectures, and so have the available libraries. Several national labs have investigated possible portable solutions for next generation codes. The Sandia National Laboratory has developed Kokkos, a C++ library which supports Pthreads, OpenMP and CUDA (Edwards and Trott 2013). Kokkos is based on multidimensional arrays which cover device memory and textures for linear algebra routines. Kokkos uses C++ and supports CUDA and thus is able to implement API calls which take in functors as inputs for generating code at compile-time. Likewise, Lawrence Livermore National Laboratory developed RAJA, a C++ library focused on portability for simulation codes (Hornung and Keasler 2013). Similar to Kokkos, RAJA focuses on kernel generation through functors but relies in C++ 11 and lambdas. The Los Alamos National Laboratory has instead focused on improving Thrust which support CUDA, OpenMP and Intel's TBB (Bell and Hoberock 2011). And lastly, the Oak Ridge National Laboratory been working with directive approaches, such as OpenACC.

Relying on libraries to provide optimized subroutines can improve developer time, but can also limit the scope of an application and cause excess data movement. These

tailored libraries, although allowing for minimal kernel generation, do not expose enough flexibility for customizing complex kernels. Because our goal is to present a portable solution for developing in heterogeneous environments, releasing a set number of specialized routines is not sufficient. However, we acknowledge and adopt salient features such as the simplification in between host and device interaction seen in libraries like Thrust, Kokkos and RAJA.

1.4 Publications

During the thesis work, I had the opportunity to collaborate with various colleagues from Rice University, industry, and national laboratories. I would like to include the co-authored publications that have been accepted in peer-reviewed journals, in submission, or in progress, together with a list of talk invitations and poster presentations.

1.4.1 Published Journal Articles

- Gandham, R., Medina, D. and Warburton, T. 2014, GPU Accelerated discontinuous Galerkin methods for shallow water equations, *Communications in Computational Physics*. This paper discusses a high-order discontinuous Galerkin method for the shallow water equations, implementing algorithms implemented used OCCA.

- Medina, D. S., St-Cyr, A. and Warburton, T. 2015, High-Order Finite-differences on multi-threaded architectures using OCCA, *in* 'ICOSAHOM 2015', Springer. We discuss a high-order finite difference implementation for a seismic imaging. An efficient algorithm implemented in OCCA is described for the wave equation using a vertical transverse isotropy model.

1.4.2 Journal Articles in Progress

- Medina, D. S., St-Cyr, A. and Warburton, T. 2014, OCCA: A unified approach to multi-threading languages, arXiv preprint arXiv:1403.0968. The original paper describing OCCA, it's programming model and the macro-based OCCA IR (intermediate representation) discussed in chapter 2.
- Rahaman, R., Medina, D., Lund, A., Tramm, J., Warburton, T. and Seigel, A. 2015, Portability and Performance of Nuclear Reactor Simulations on Many-Core Architectures, *in* '77th EAGE Conference and Exhibition 2015'. This article came from a collaboration with a group at the Center for the Exascale Simulation of Advanced Reactors (CESAR) at the Argonne National Laboratory. We compare a neutronics Monte Carlo algorithm on various architectures and programming approaches, such as OCCA and OpenACC.
- Fahrenholtz, S. J., Moon, T., Franco, M., Medina, D., Danish, S., Gowda, A., Shetty, A., Maier, F., Hazle, J., Stafford, R. J., Warburton, T. and Fuentes, D. n.d., A model evaluation study for treatment planning of laser induced thermal therapy. This article discusses the use of a spectral element method for simulating heat transfer from a laser induced thermal therapy. The efficient implementation uses OCCA to utilize modern architectures.

1.4.3 Conferences and Seminar Invitations

- OKL*: A unified kernel language for parallel architectures*, *Sandia National Laboratory, May 2015*
- OKL*: A unified kernel language for parallel architectures*, *SIAM CSE '15, March 2015*

- OKL: *A unified kernel language for parallel architectures*, *Rice Oil & Gas '15, March 2015*
- OKL: *A unified kernel language for parallel architectures*, *Chevron, February 2015*
- *Tutorial on* OCCA, *CESAR Sound-off meeting at Argonne National Lab, December 2014*
- OCCA: *A unified approach to multi-threading languages*, *Computation-Institute for Scientific Computing Research at Lawrence Livermore National Lab, September 2014*
- OCCA: *A unified approach to multi-threading languages*, *ICOSAHOM 2014, June 2014*
- *High-order Numerical Methods for High-Contrast Seismic Imaging*, *Rice Oil & Gas '13, February 2013*

1.4.4 Poster Presentations

- OKL: *A unified kernel language for parallel architectures*, *SIAM CSE '15, March 2015*
- OCCA: *A unified approach to multi-threading languages*, *Rice Oil & Gas '14, March 2014*

1.5 Outline

The goals for this thesis include a portable solution for current and future architectures, an ease for the development of parallel codes, and the ability to expose as much parallelism as possible for achieving optimal performance; in other words, uniting the

advantages from the approaches previously mentioned. A portable solution is developed using the OCCA intermediate representation discussed in chapter 2, a macro-based approach using the preprocessor for source-to-source transformation. Facilitating the development for heterogeneous platforms is addressed by introducing OKL and OFL, minor extensions to C and Fortran (discussed in chapter 3) which replaces `#pragma`'s with more familiar loop structures. Lastly, by requiring users to expose parallelism, the OKL and OFL languages avoid unintentionally removing features seen in multicore and many-core architectures.

The thesis content is split into the different components that make up the OCCA project. Chapter 2 details the host API and the OCCA intermediate representation which describes how portability is achieved. Also included in chapter 2 are updates to OCCA since the original papers (Medina 2014, Medina et al. 2014) such as: current and future language support, methods which make OCCA non-intrusive, and HPC-related features. Chapter 3 outlines the tools, specifications and features for the proposed kernel languages, the OCCA kernel language (OKL) and the OCCA Fortran language (OFL). The minimal extensions to C and Fortran seen in OKL and OFL respectively, which allows OCCA to embed kernels into the application code in the future, similar to CUDA. With the compiler tools developed and discussed in chapter 3, it is possible to give an option which assimilates the kernel languages without much change to the application host. Chapter 4 and chapter 5 finish the OCCA specifications by describing two additional layers of assistance towards developers. Chapter 4 introduces unified virtual addressing, its focus being on the implemented automatic data movement for obscuring data movement between the *host* and *device*. Chapter 5 introduces a method for automatically detecting loop-carried dependencies on serial code, auto-tagging for-loops to generate OKL kernel instances, and run them for detecting efficient kernels generated. Chapter 6 details current applications based on OCCA with preliminary results jointly with benchmarks for performance and portability validation. The thesis

concludes with a synopsis of the thesis work and future work in chapter 7.

2 OCCA: Unified Approach To Multithreading Languages

The novel contribution detailed in this chapter comes from the abstracted programming model used to encapsulate native programming languages for manycore devices. An implementation of this model resulted in the OCCA *intermediate representation (IR) which supports serial code, Pthreads, OpenMP, CUDA, OpenCL, and COI. Constructing the* OCCA IR *used a generalization of current parallel architectures to unify the different languages and standards for heterogeneous computing. Furthermore, I discuss an abstract offload model demonstrating a runtime compilation design with the ability to pick a target architecture at runtime and utilize native compilers for their respective language standard.*

In this chapter, I discuss an abstract programming model for manycore devices. To further motivate the use of a unified programming model, I first demonstrate the architecture similarities between traditional multicore processors and current graphics processing units (GPUs). The similarities in manycore devices motivate features in the proposed unified programming model. An implementation of this concept was developed, producing the OCCA intermediate representation (IR) which generalizes current parallel architectures to unify the different languages and standards for heterogeneous computing, including serial code, Pthreads, OpenMP, CUDA, OpenCL, and COI.

Accompanying the OCCA IR is an abstracted offload model for the supported backends and a unified interface to manage it, where the offload model defines the *host* as the computational processing unit running an application, offloading computations

to the *device*, the targeted hardware used for additional computations. We conclude the chapter by discussing shortcomings in the macro-based language and transition towards the kernel language discussed in chapter 3.

2.1 OCCA Background

Prior to the OCCA project, I developed and translated industrial and academic numerical simulation codes between OpenMP, OpenCL, and CUDA. Through this experience, it became apparent that optimizations between parallel languages did not differ much due to the commonality across their programming models as inferred by the approaches mentioned in section 1.3. It was of interest to investigate a method for leveraging the common aspects found in OpenMP, OpenCL, and CUDA. Thus I developed a prototype for OCCA to offer a unified frontend for those languages and standards through an offload model. The offload model retains the programming model seen in OpenCL and CUDA, where the *host* uses the developed frontend to communicate with the *device*. However, because the coding standards for each device still differed, I created the OCCA intermediate representation to also unify the OpenMP, OpenCL and CUDA compute kernels. I discuss current parallel architecture similarities, a motivation for the development of a single kernel language which can maintain performance across distinct platforms.

Chapter 1 gave a brief introduction to the development of current architectures. In this section, I elaborate on CPU and GPU architectures and their programming models. By describing architectural similarities, we motivate the development of the OCCA IR and OCCA application programming interface (API) discussed in the next sections. We start by outlining the traditional CPU architecture, the most common processor currently used in HPC, followed by analogous descriptions for the GPU

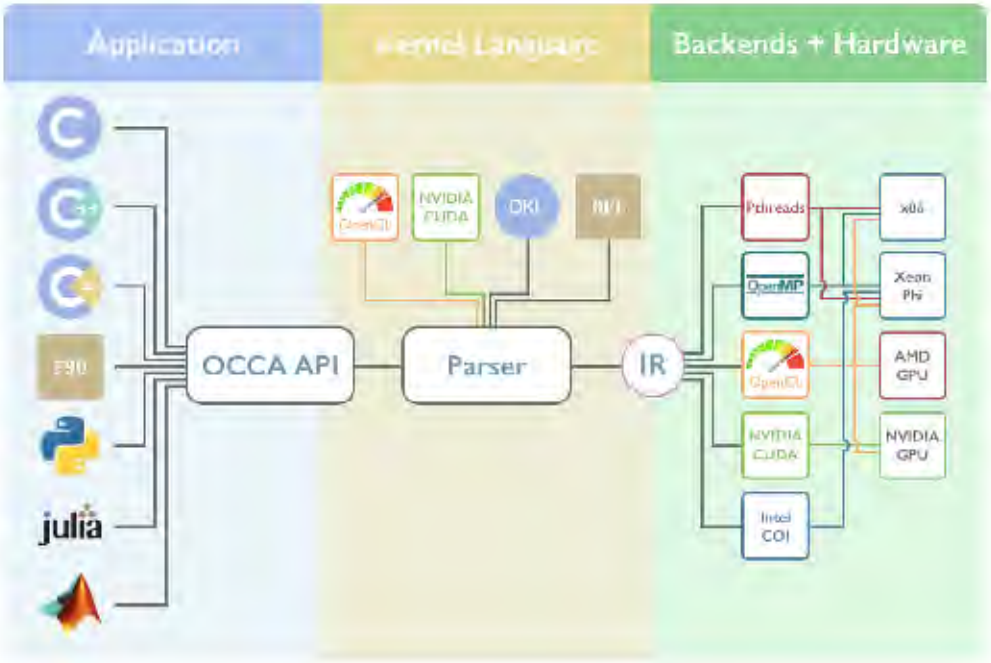

Figure 2.1: Current relationship between supported frontends, the OCCA languages (OKL, OFL, OCCA IR), and supported backends

architecture.

2.1.1 CPU Architecture

As mentioned in subsection 1.1.1, it has become common for architectures to feature a hierarchy of cache. When data is queried by the processor, a segment of data is moved to cache if not already in cache, as seen in figure 2.2. Common optimization algorithms tailored for modern CPUs try to make use of data locality to maintain needed data in cache, otherwise the segment (cache line) could be evicted.

An increase in clock frequency was slowly followed by an increase in data loading capabilities (Wulf and McKee 1995). With the plateau in clock speed, different approaches to increasing computational speed were examined, such as vectorization. Vectorization is a computational complement to local data fetching in caching policies which applies one instruction to consecutive data as seen in figure 2.3

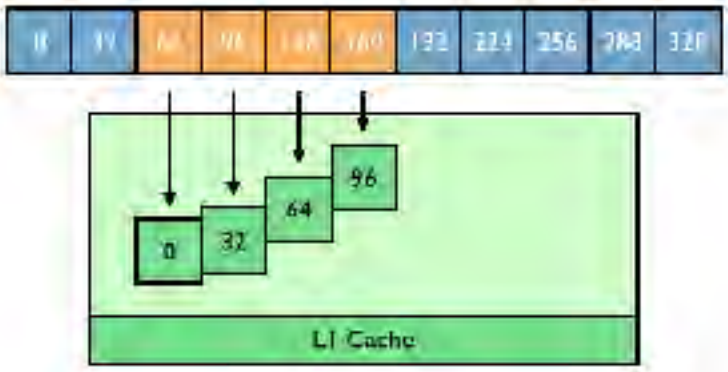

Figure 2.2: When data is accessed from DRAM, neighboring data is sent and stored through the different cache levels.

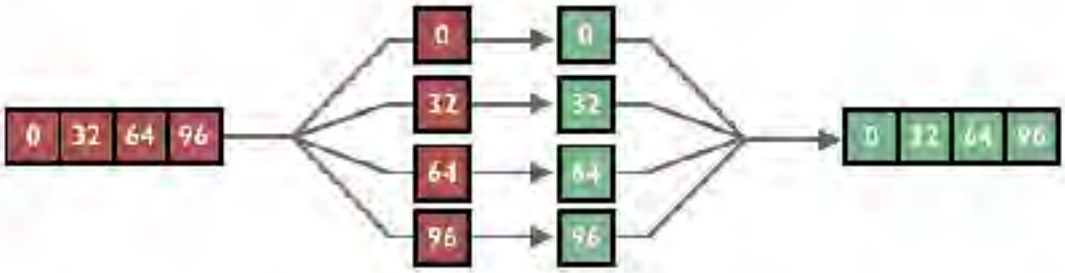

Figure 2.3: With large registers and vectorization instructions, multiple consecutive data entries can be updated in parallel.

Intel's MMX vectorization instruction set introduced single-instruction multiple-data (SIMD) operations for the x86 architectures, allowing for operations on 64-bit registers to occur concurrently. MMX was followed by the streaming SIMD extension (SSE) instruction set family (SSE, SSE2, SSE3, SSE4) which operated on 128-bit registers. The most current vectorization instruction set by Intel is the advanced vector extensions (AVX) supporting 256-bit registers and 512-bit registers in the case of Intel's Xeon Phi.

Multithreading is the last CPU architecture feature that will be discussed. A large advancement has been made since the first dual-core processor in 2001, to consumer-level processors possessing up to eight cores. Akin to creating a network of computing units (such as those found in Beowulf clusters and current supercomputers) multi-threading integrates multiple independent compute units in a processor. By incorpo-

rating multithreading, it is possible to scale algorithms by the number of cores while still making use of the architecture optimizations mentioned above. This next section will go over major optimizations used in the GPU architecture which are then compared with the architecture features discussed in this section.

2.1.2 GPU Architecture

Graphics processing units were originally developed to manage computations needed in 3D rendering procedures. By using programmable graphics shaders, such as OpenGL's shading language (GLSL) and DirectX's high-level shader language (HLSL), processes were able to offload data and processing to graphics cards. The graphics pipeline can be generalized into three parts: primitive processing, primitive culling, and fragment processing. While the first two steps detect the primitives (such as triangles and quadrilaterals) which will be rendered, the fragment processing step is usually the most computational intensive processes in the graphics pipeline. Each fragment represents the smallest unit in the rendered buffer, analogous to a pixel and it's display, and whose value is calculated through a graphics shader. All fragments run through the same computational shader, an embarrassingly parallel task.

Figure 2.4 shows the GPU architecture layout and memory hierarchy. The compute units in the GPU are denoted by work-items, or threads in CUDA, which are grouped into work-groups, blocks in CUDA, using a SIMD operation approach, relabed as SIMT (single-instruction multiple-threads) by NVIDIA. A synchronization between all work-groups can only occur when a kernel finishes executing since the standard allows for more work-groups than available hardware to execute them; in other words, there is an implicit loop over work-groups that need to be launched for every kernel. In order to expose enough parallelism for GPUs, the user must lay out instructions to be executed as a work-item. Work-items which reside on specific hard-

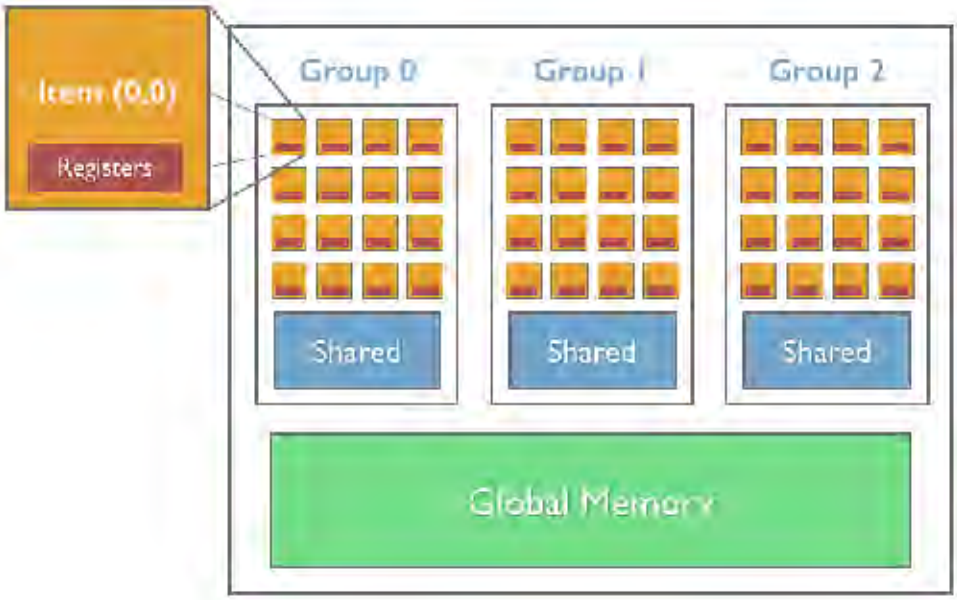

Figure 2.4: GPU Hierarchy showing work-group and work-item relation as well as the different memory types and their scopes.

ware groups (as wavefronts in AMD terminology and warps in NVIDIA terminology) are issued the same instruction to execute with the option of calling a NOOP, no operation, instruction when branching occurs. Figure 2.5 shows kernel execution with branching.

Figure 2.5: Work-items on the same hardware groups (half-warp or wavefront) follow the same instruction or call a NOOP (No Operation) instruction.

In the memory hierarchy seen in figure 2.4, each work-item is associated with its own private register set, using the memory layer with the fastest fetching times. Shared memory is the second-fastest memory type found on GPU architectures, which uses the same physical hardware as cache. Apart from latency, shared memory is useful for

synchronizing and communicating data across work-items. Global memory refers to the DRAM found in GPUs, having the longest latency in the GPU memory hierarchy. Data resident in global memory can be fetched from all work-items with support for atomic operations across work-groups.

Aside from GPU shared memory, the existence of registers, cache and DRAM are also seen in CPU architectures. Similarly, the global bandwidth is distributed to the memory managers between the large number of compute units found on modern GPUs; hence, bandwidth is also a common bottleneck in most numerical applications.

Figure 2.6: Having consecutive work-items load from global memory consecutive data, otherwise known as coalescing reads, allows full use of the bus which vectorizes loads.

If we compare memory access speeds on an NVIDIA K40, NVIDIA's newest single-processor workstation card at this time, the bandwidth for global memory accesses is 288 GB/s while shared memory bandwidth is around 1.2 - 1.8 TB/s using NVIDIA's profiler. The large gap in bandwidth often forces optimizations and best-practices to avoid global memory fetches. The cost of global memory fetches can be reduced through consecutive work-items loading consecutive data (coalesced reads) by making use of the GPU's large bus (384 and 512 bits on current NVIDIA and AMD cards respectively), as seen in figure 2.6. Likewise, shared and register data can be used as intermediate storage for computations.

2.1.3 CPU and GPU Similarities

By looking at the optimizations discussed for both CPU and GPU architectures, we identify shared features in their programming practices. The key features include: memory accesses, vectorization and efficient multithreading.

Efficient GPU memory accesses make use of the large bus width by using consecutive strides in memory. In comparison, the CPU would make use of resident cache lines by also loading consecutive strides in cached memory. As for vectorization, hardware mapped to consecutive work-items, such as a half-warp or wavefront, could be thought of entities in a vector unit. Being forced to apply the same instruction or call a NOOP is analogous to a CPU's vector instruction operating on a large register. Lastly, the GPU programming model makes an assumption that there is no interdependence between work-groups. If we make the same assumption on a CPU multithreaded program that work is split in the work-group/work-item paradigm, each thread would become independent aside from atomic operations. Forks and joins are no longer present through the assumption that threads operate on independent work-groups.

Similarities in optimization procedures between the CPU and GPU create the foundation on which OCCA is developed. Making use of the work-group/work-item model while exposing shared memory unifies most optimizations found in the CPU and GPU architectures. Unfortunately, as is discussed in chapter 6, performance is not always portable between architectures. However, language has enough flexibility to include architecture-specific kernel optimizations to achieve near-optimal performance.

2.2 OCCA Intermediate Representation (IR)

This section discusses the approach taken to integrate different heterogeneous parallel programming languages and standards. While the focus for the unified programming model arose from the CPU and GPU, as described in section 2.1, the OCCA

IR targets a variety of devices. Given the support for the initial backends (OpenMP, CUDA, and OpenCL), OCCA is usable on traditional CPUs, GPUs, Intel's Xeon Phi, and FPGAs. However, the number of available backends was extended during the thesis work to include Pthreads, COI, and a prototype for HSA (Heterogeneous System Architecture).

We start by motivating the derivation of the OCCA IR by comparing with prior approaches, identifying features used (or improved upon) together with disadvantages which were overcome. Next, we describe the programming model specification for the OCCA IR kernel specification. Additionally, we describe the device memory hierarchies found in the OCCA IR and their use in HPC applications.

2.2.1 Derivation

Key features found in GPU architectures, and later in the Intel's Knights Landing containing large on-chip memory storage, must be made accessible for the high performance focus of the project. Our programming model must be flexible in order to incorporate future architectures and language standards. Recalling section 1.3, portability approaches across multiple backends were categorized into directive, source-to-source, and wrapper approaches. Here we give short summaries of their advantages, which we adopt into the OCCA IR, as well as their disadvantages which we address through the OCCA project.

The directive approach, as seen in OpenMP and OpenACC, relies on a proper mapping between user code and its mapped compiler code transformation. Unfortunately, aside from trivial code snippets, directing proper code transformations tailored for the GPU requires the user to properly manage code transformations through additional `#pragma`'s. In addition, interactions between the *host* and *device* such as data allocation, data movement and data freeing requires user management if performance is

of importance. This user management turns the directive approach usage of `#pragma`'s to be equivalent to a library `API` combined with `__attribute__` language extensions, in order to give compiler information about variables and statements. The adoption and improvements of automatic data movement is discussed in chapter 4 and automatic code transformation in chapter 5; however, the aim of the OCCA IR is that of creating a single programming format towards multiple devices.

The source-to-source approach, at least in the context of CUDA to OpenCL, has the advantage of assuming the user is writing in the GPU programming model. CUDA and OpenCL have options for low-level optimizations and multiple memory hierarchies, permitting the use of a higher descriptive language compared to traditional languages, such as C and Fortran. However, available source-to-source translators handle only a subset of CUDA or it's assembly language (PTX) as mentioned in section 1.3. Rather than relying on a proprietary programming language with no standard, a custom programming language became an appealing solution.

Lastly, the wrapper approach was not an option due to the limitation on their routines. However, being able to develop libraries and wrappers using OCCA was a requirement due to their use in application development.

The first step towards creating a custom programming language was to develop an intermediate representation. Developing the OCCA programming model started with a macro-based approach, covering major keywords used in supported modes and unraveling macros based on the chosen backend. By developing the OCCA IR as a macro-based language, there exists the option to further append additional backends. We introduce the OCCA IR through its programming model and specification.

2.2.2 Programming Model

Before covering the set of macros making up the OCCA IR, we introduce the programming model For a comprehensive list of the OCCA IR macros, the reader can refer to appendix A. While the OCCA IR programming model is based on the GPU programming model, the major difference is the inclusion of explicit loops to denote parallelism. As discussed in subsection 2.1.3, modern CPU and GPU architectures contain similar aspects in their programming models. Visually shown in figure 2.7 are the similarities between GPU work-group independence and CPU thread-independence, as well as the work granularity between GPU work-items operations and SIMD vector-lane operations. However, rather than imposing implicit iterations over work-groups and work-items as seen in OpenCL and CUDA, we expose work distribution with the for-loop-like macros seen in code listing 2.1 and code listing 2.2. Code listing 2.1 compares the OCCA IR mapping to CUDA and OpenCL while code listing 2.2 compares its mapping to OpenMP.

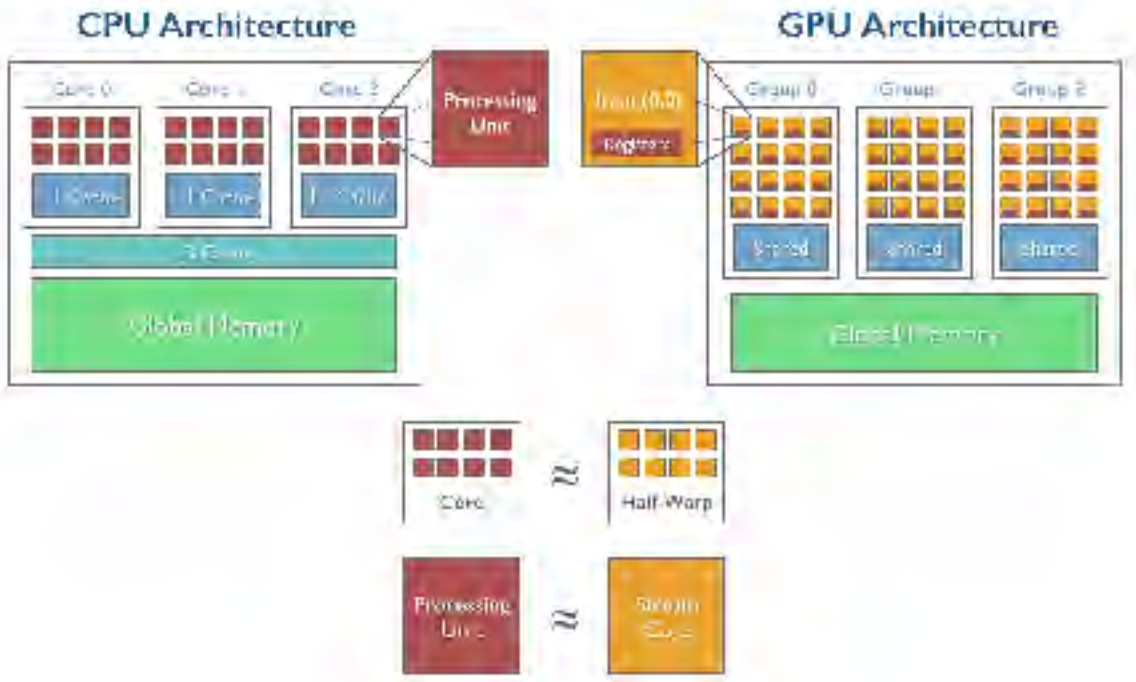

Figure 2.7: Aside from memory hierarchy similarities, current multicore processors equipped with vector instructions are similar to graphics processing units

```
// (1) CUDA:
//     Example kernel
//     with a thread's
//     scope of work


__global__ void kern(){



  // Thread Work


}
```

```
// (2) OpenCL:
//     Example kernel
//     with a work-item's
//     scope of work


__kernel void kern(){



  // Work-item Scope


}
```

```
// (3) OCCA:
//     Example kernel
//     showing different
//     scopes of
//     parallelization

occaKernel void kern(){
  occaOuterFor0{
    // Work-group Scope
    occaInnerFor0{
      // Work-item Scope
    }
  }
}
```

Listing 2.1: Difference between implicit loops in CUDA and explicit loops in the OCCA IR.

```
// (1) OpenMP:
//     Example kernel
//     with OpenMP
//     #pragmas


extern ''C'' void kern(){
  #pragma omp for
  for(int t = 0; t < 8; ++t){
    // Thread scope
    #pragma simd 8
    for(int v = 0; v < 8; ++v){
      // Vectorization Operations
    }
  }
}
```

```
// (2) OCCA:
//     Example kernel
//     showing different
//     scopes of
//     parallelization

occaKernel void kern(){

  occaOuterFor0{
    // Thread scope

    occaInnerFor0{
      // Vectorization Operations
    }
  }
}
```

Listing 2.2: Difference between implicit loops in CUDA and explicit loops in the OCCA IR.

We first discuss the OCCA IR for-loop keywords shown in code listing 2.1 and code listing 2.2. Similar to the current GPU programming models, the `occaOuterForX` loops iterate over independent groupings of compute units where `X` $\in$ {`0, 1, 2`} allowing groupings of work-groups in up to three dimensions. Likewise, work-groups can be composed as 1, 2, or 3 dimensional groupings of work-items dictated by `occaInnerForX` where `X` $\in$ {`0, 1, 2`}. If OpenCL and CUDA took this approach for their program-

ming models, the outcome would look similar to that seen in code listing 2.3 and code listing 2.4. The complete macro expansions for the OpenMP, CUDA, and OpenCL modes in OCCA IR for-loops are seen in code listing 2.5.

```
__kernel void kern(){
  for(int wgZ = 0; wgZ < get_num_groups(2); ++wgZ){          // Loop Grouping (1)
    for(int wgY = 0; wgY < get_num_groups(1); ++wgY){
      for(int wgX = 0; wgX < get_num_groups(0); ++wgX){
        // Work-group Scope

        for(int wiZ = 0; wiZ < get_local_size(2); ++wiZ){    // Loop Grouping (2)
          for(int wiY = 0; wiY < get_local_size(1); ++wiY){
            for(int wiX = 0; wiX < get_local_size(0); ++wiX){
              // Work-item Scope
        }}}
  }}}
}
```

Listing 2.3: The code listing expands the implicit for-loops found in OpenCL kernels. Loop grouping (1) expands multidimensional work-groups and loop grouping (2) expands multidimensional work-items.

```
__global__ void kern(){
  for(int wgZ = 0; wgZ < gridDim.z; ++wgZ){              // Loop Grouping (1)
    for(int wgY = 0; wgY < gridDim.y; ++wgY){
      for(int wgX = 0; wgX < gridDim.x; ++wgX){
        // Block Scope

        for(int wiZ = 0; wiZ < blockDim.z; ++wiZ){       // Loop Grouping (2)
          for(int wiY = 0; wiY < blockDim.y; ++wiY){
            for(int wiX = 0; wiX < blockDim.x; ++wiX){
              // Thread Scope
        }}}
  }}}
}
```

Listing 2.4: The code listing expands the implicit for-loops found in CUDA kernels. Loop grouping (1) expands multidimensional work-groups and loop grouping (2) expands multidimensional work-items.

```
//---[ (A) OpenMP Mode ]-----------------------------------------------------

extern ''C'' void kern(){
  #pragma omp parallel for
  for(int wgZ = 0; wgZ < occaOuterDim2; ++wgZ){            // Loop Grouping (1)
    for(int wgY = 0; wgY < occaOuterDim1; ++wgY){
      for(int wgX = 0; wgX < occaOuterDim0; ++wgX){
        // Block Scope

        for(int wiZ = 0; wiZ < occaInnerDim2; ++wiZ){      // Loop Grouping (2)
          for(int wiY = 0; wiY < occaInnerDim1; ++wiY){
            for(int wiX = 0; wiX < occaInnerDim0; ++wiX){
              // Thread Scope
        }}}
  }}}
}

//---[ (B) CUDA Mode ]-------------------------------------------------------

__global__ void kern(){
  {{{
     // Block Scope
     {{{
        // Thread Scope
     }}}
  }}}
}

//---[ (C) OpenCL Mode ]-----------------------------------------------------

__kernel void kern(){
  {{{
     // Work-group Scope
     {{{
        // Work-item Scope
     }}}
  }}}
}
```

Listing 2.5: The code listing shows simplified expansions of the OCCA-for loops for (A) OpenMP-mode, (B) CUDA-mode, and (C) OpenCL-mode.

To truly create a unified programming model that will maintain correctness across the supported backends, we introduce the OCCA IR specification. We begin by stating what is permitted and not permitted between scopes in an OCCA IR kernel as seen in code listing 2.6. More specifically, allowed statements in the following 6 specialized scopes:

A) Kernel arguments

B) Outside `occaOuterFor` loops

C) Between `occaOuterFor` loops

D) Between an `occaOuterFor` and `occaInnerFor` loops

E) Between `occaInnerFor`

F) Inside `occaInnerFor`.

```
occaKernel void kernelName(occaKernelInfoArgs, ... /* (A) */){
  // (B)
  occaOuterFor1{
    // (C)
    occaOuterFor0{
      //(D)
      occaInnerFor1{
        //(E)
        occaInnerFor2{
          // (F)
        }
      }
    }
  }
}
```

Listing 2.6: The code listing shows the specialized scopes inside an OCCA IR kernel.

2.2.2.1 Kernel Arguments

The scope covered in this subsection is denoted by (A) in the code listing 2.6.

We aim to enable any type as a kernel argument, thus provided multiple keywords for describing kernel arguments. The CUDA specification and OpenCL standard contain routines for additional languages to utilize them and thereby allowing OCCA to generate and launch kernels with ease. With the CPU-based backends (serial, OpenMP, and Pthreads), however, we faced restrictions due to the `C/C++` language. Two constraints appear: calling kernels with arbitrary arguments and seamlessly passing the `occaOuterFor` and `occaInnerFor` bounds.

```
//---[ 1. Regular C ]------------------
void kernelName(int * A,
                int & B){}
//---[ 2. OCCA IR   ]------------------
void kernelName(occaKernelInfoArgs,
                occaPointer int * A,
                int occaVariable  B){}
```

Listing 2.7: The code listing shows the distinct kernel argument types, where `A` corresponds to arrays and `B` to arguments passed by value.

Similarly to Fortran, passing all arguments by reference allowed for a non-templated approach for launching kernels with varying arguments dynamically. While there exists branching to pick the correct function pointer which will launch the OCCA IR kernel, the portable method can be used with multiple frontend languages. Code listing 2.7 shows additional keywords (`occaPointer` and `occaVariable`) used to decorate kernel arguments, abstracting data types to control how they're treated by the compiler. `occaPointer` is used to declare the scope of the memory object in GPU-based modes, while it is mere decoration for CPU-based modes. On the other hand, `occaVariable` is used as decoration in the GPU-based modes, but allows arguments to be passed by their address in CPU-based modes.

The second requirement relevant to the kernel arguments is the addition of `occaKernelInfoArgs`. CUDA and OpenCL kernels require loop dimensions prior to launching, a trait lacking on CPU-based modes. In the OCCA IR expansions for CPU-modes, the `occaOuterFor` and `occaInnerFor` loops expand to for-loops whose bounds are passed through the `occaKernelInfoArgs` macro.

```
const int *occaKernelArgs, int occaInnerId0, int occaInnerId1, int occaInnerId2
```

where `occaKernelArgs` stores the loop bound information, and the variables `occaInnerIdX` are the inner loop iterators.

2.2.2.2 Outside `occaOuterFor`

The scope covered in this subsection is denoted by (B) in the code listing 2.6.

Following the kernel arguments, the kernel usually begins with an `occaOuterFor` statement. Statements inside the OCCA IR kernel scope but outside of an `occaOuterFor` are restricted to constant variable definitions. Because the outer and inner scopes are emulated in GPU-modes, statements are executed once per work-item/thread while statements are only executed once for CPU-modes.

2.2.2.3 Between `occaOuterFors`

The scope covered in this subsection is denoted by (C) in the code listing 2.6.

Between the `occaOuterFor` scopes, non-constant Variable definitions are supported with restrictions. Variables decorated with `occaShared` and `occaPrivate` can be defined in this scope but not initialized; definitions of these qualifiers are found in subsection 2.2.3. However, variables can only be initialized and not updated due to the emulated scopes in the GPU-modes.

Lastly, `occaOuterFor` loops are not guaranteed to be executed the given nesting in order. Hence, the iterator for the `occaOuterForX` loop which is denoted by `occaOuterIdX` where `X` $\in$ {`0, 1, 2`} could execute in any permutation.

2.2.2.4 Between `occaOuterFor` and `occaInnerFor`

The scope covered in this subsection is denoted by (D) in the code listing 2.6.

`occaInnerFor` loops are located inside `occaOuterFor` loop scopes, but more statements are allowed inside `occaOuterFor` loops. Declared variables are still required to be constant or decorated with `occaShared` or `occaPrivate`; however, conditional statements and loops are supported. The only limitation on conditional statements and loops is that each must contain an `occaInnerFor`.

2.2.2.5 Between `occaInnerFors`

The scope covered in this subsection is denoted by (E) in the code listing 2.6.

Between `occaInnerFor` loops, variable declarations must be decorated with `const`, `occaShared`, or `occaPrivate`. No other statements are supported in this scope.

2.2.2.6 Inside `occaInnerFors`

The scope covered in this subsection is denoted by (F) in the code listing 2.6.

Inside the inner-most `occaInnerFor` scope, anything supported by C is supported in the OCCA IR. Similar to `occaOuterFor` loops, `occaInnerFor` are not guaranteed to be executed in order. Hence, the iterator for the `occaInnerForX` loop which is denoted by `occaInnerIdX` where `X` $\in$ {`0, 1, 2`} could execute in any permutation.

2.2.3 Device Memory Hierarchy

With the exposure of loop-structures already described, we begin to describe the memory hierarchy exposed by the OCCA IR. Recalling that memory has become one central issue in optimizing for performance, we emphasize memory hierarchy exposure in the OCCA programming model. There are three types of memory layers: global memory, shared memory and register memory; all which are exposed in the OCCA IR. The simplest level of memory to explain is global memory which represents data residing in RAM, be it CPU or GPU DRAM.

Because the concept of shared memory differs between CPU and GPU architectures, the mention of shared memory throughout the thesis proposal will correspond with the GPU shared memory terminology. Shared memory resides in the same physical location as cache for GPUs and can be described as a memory scratchpad found in fast-memory. Important features of shared memory include fast data fetching as well as a synchronization method between work-items in a work-group. By encompassing

work-items in a work-group in the shared memory scope, algorithms can make use of parallel data fetching and storage with visibility throughout the work-group. In the OCCA IR, we represent shared memory with the qualifier `occaShared` as seen in code listing 2.8. With shared memory being a supported feature in OCCA, we also include local and global barriers to synchronize across work-items in a work-group. An example is given in code listing 2.8.

```
occaKernel void swap16(occaKernelInfoArgs,
                       occaPointer float *ptr){
  occaOuterFor0{
    occaShared float s_ptr[16];

    occaInnerFor0{
      if(occaInnerId0 < 16)
        s_ptr[occaInnerId0] = ptr[occaInnerId0];
    }

    occaBarrier(occaLocalMemFence);

    occaInnerFor0{
      if(occaInnerId0 < 16)
        ptr[occaInnerId0] = s_ptr[16 - occaInnerId0];
    }
  }
}
```

Listing 2.8: Shown is the barrier implementation to synchronize across a work-group as splitting inner-loops, explicitly stating that all work-items have finished the instructions prior to the barrier.

Lastly, we will discuss registers and some issues surrounding their use. Data stored in registers utilize the fastest memory layer in the memory hierarchy for both CPU and GPU architectures. Hence programmers and compilers try to find a balance between data storage on registers and the amount of registers available to the hardware (for example, prefetching onto registers and reusing them as opposed to fetching from higher latency memory layers). However, the use of barriers causes an issue for OCCA-modes which treat inner-loops serially, such as OpenMP, Pthreads and COI. For example, in code listing 2.9, the variables `reg` and `regArray` would normally be overwritten in each loop iteration. In order to keep the code from overwriting `reg` and `regArray`, we implement a wrapper for types through the `occaPrivate` call. For OCCA IR, the

variable holds an array with an entry for each work-item, allowing mutual exclusion of variables through behaving as one in the kernel code.

```
occaKernel void kern(occaKernelInfoArgs){
  occaOuterFor0{
    occaPrivate(int, reg);              // Register      : int reg;
    occaPrivateArray(int, regArray, 2); // Register Array: int regArray[2];

    // . . .

    occaInnerFor0{
      reg = occaGlobalId0; // reg would normally be overwritten by the loop
      regArray[0] = 0;     // regArray would also normally be overwritten
      regArray[1] = 1;
      // . . .
    }

    occaBarrier(occaLocalMemFence);

    occaInnerFor0{
      int i = reg;         // Allocating registers normally for only one scope
      int d = regArray[0];
      // . . .
    }
  }
}
```

Listing 2.9: Shown is a simple example of private variable use to prevent overwriting during each loop iteration.

2.3 Application Programming Interface

Apart from the OCCA IR, OCCA implements an application programming interface (API) based on an offload model. This section introduces the abstractions used to generalize modern parallel architectures into an offload model containing features present in high performance computing. We begin with the device, memory, and kernel abstractions, followed by kernel compilation and concluding with additional HPC-related features.

2.3.1 Offload Model and Device Abstractions

While section 2.2 discussed abstractions for programming current parallel architectures, this section discusses the API which make the offload model abstraction possible. The model consists on the three key components that influenced the OCCA host API development: the device, the device memory, and the device kernels.

2.3.1.1 `occa::device` Class

An OCCA device acts as a layer of abstraction between the OCCA API and the API from supported languages. One of the essential features in OCCA is the ability to target a device at run-time. Choosing the preferred platform at run-time is possible due to just-in-time code generation as seen in code listing 2.10.

```
occa::device device1("mode = Serial");
occa::device device2, device3, device4, device5, device6;

device2.setup("mode = Pthreads, threadCount = 8, schedule = compact");
device3.setup("mode = OpenMP  , threadCount = 8, schedule = compact");
device4.setup("mode = OpenCL  , platformID  = 0, deviceID = 0");
device5.setup("mode = CUDA    , deviceID    = 0");
device6.setup("mode = COI     , deviceID    = 0");
```

Listing 2.10: The code listing shows the `occa::device` class initialization, choosing a backend at run-time while providing a portable and modular method.

An OCCA device generates a self-contained context and stream from the chosen device, being a socketed processor, GPU or other OpenCL supported devices such as a Xeon Phi or an FPGA. Although multiple contexts within a device are not supported, asynchronous computations are supported through the use of multiple OCCA device instances. The device's main purpose is to allocate memory and compile kernels for the chosen device. Additional responsibilities of the device wrapper include stream management. Work enqueued onto a stream are launched sequentially, but streams give greater access to asynchronous work; examples can be seen in code listing 2.11.

```
occa::device device;
// ...
// Get current stream
occa::stream streamA = device.getStream();
// Generate a new stream
occa::stream streamB = device.createStream();
// Switch stream
device.setStream(streamB);

occa::streamTag startTag = device.tagStream();
// Work
occa::streamTag endTag = device.tagStream();
double timeTaken = device.timeBetween(startTag, endTag);
```

Listing 2.11: The code listing shows the `occa::device` class options for creating, updating, and destroying streams as well as timing work between streams.

2.3.1.2 `occa::memory` Class

The OCCA memory class abstracts the different device memory handles and provides some useful information such as device array sizes. Although memory handling in OCCA facilitates host-device communication, the management of reading and writing between host and device, for performance reasons, currently requires programmer management. Chapter 4 describes an alternative to manual data management using emulated unified memory addressing and automatic data management.

Basic memory initialization, similar to `malloc` in `C`, and data transfer can be seen in code listing 2.12.

```
occa::device device;
// ...
// Allocate memory on the host
int *A = (int*) malloc(10 * sizeof(int));
// Allocate memory on the device
occa::memory memoryA = device.malloc(10 * sizeof(int), A);
occa::memory memoryB = device.malloc(10 * sizeof(int));
// Forms of transfering data between
//   host and device
memoryA.copyTo(memoryB);
memoryB.copyTo(A);
memoryB.copyFrom(A, 10 * sizeof(int));
// Asynchronous data transfer
memoryA.asyncCopyTo(memoryB);
memoryB.asyncCopyTo(A);
memoryB.asyncCopyFrom(A, 10 * sizeof(int));
```

Listing 2.12: The code listing shows the `occa::memory` initialization and data transfer options.

2.3.1.3 `occa::kernel` Class

The OCCA kernel class unites device function handles with a single interface, whether for a function pointer (CPU-based modes), cl_kernel (OpenCL), or cuFunction (CUDA). When using the OpenCL and CUDA kernel handles, passing the arguments through their respective API is simple, but there are discrepancies when comparing to the OpenMP wrapper. For example, OpenCL and CUDA kernels work-items have access to work-group and work-item counts implicitly. However, C++ functions only have access to the function scope and global namespace, requiring the work-group and work-item counts to be passed as macros or as an argument to the kernel.

Two formats are available for launching `occa::kernel`s, through the function-like `()` operator or with an argument list. Before launching a kernel, the user is required to pass the outer and inner loop bounds. Examples of kernel building and launching are given in code listing 2.13.

```
occa::device device;
// ...
// Allocate memory on the device
occa::memory memoryA = device.malloc(10 * sizeof(int));
// Build kernel from source
occa::kernel kernelA = device.buildKernel("filename.occa",
                                          "kernelName");
// Build kernel from a string
occa::kernel kernelB = device.buildKernel(
  "occaKernel void kernelName(occaKernelInfoArgs,"
  "                           occaPointer int *A){}",
  "kernelName");

// Set kernel work dimensions
kernelA.setWorkingDims(1, occa::dim(16), occa::dim(1));
kernelB.setWorkingDims(1, occa::dim(16), occa::dim(1));

// Launch kernel natively
kernelA(memoryA);
// Launch kernel through an argument list
kernelB.addArgument(0, memoryA);
kernelB.runFromArguments();
```

Listing 2.13: The code listing shows the `occa::kernel` initialization and launching options.

2.3.2 Kernel Compilation

To enable run-time platform selection and provide users the ability to write custom kernels, we chose to use run-time compilation. The major advantage of this compilation method is the language flexibility presented over wrapper approaches which are limited to their available routines. A performance-based advantage is the ability to reveal run-time processed information to the compiler for additional compiler optimizations; for example, compiling with known element polynomial order for finite element methods and finite-difference stencil size for aiding compiler vectorization. An example of injecting run-time information into OCCA kernels is given in code listing 2.14.

```
occa::device device;
// ...
// Hypothetical parameter N
int N = 10;
// Initializing kernel information
occa::kernelInfo kernelAInfo;
kernelAInfo.addDefine("N", N);
// Build kernel from source using
//   additional information stored
//   inside kernelAInfo
occa::kernel kernelA = device.buildKernel("filename.occa",
                                          "kernelName",
                                           kernelAInfo);
```

Listing 2.14: The code listing shows code injection prior to building an `occa::kernel`.

Application development can also benefit from run-time compilation. Libraries such as OCCA are primarily used for offloading heavy computations, a relatively small portion of large applications. Being able to compile specific sections of code allows for rapid prototyping by reducing overall compilation time. Additionally, parameter testing is simplified through the use of `occa::kernelInfo` as shown in code listing 2.14 by providing parameters at run-time.

Utilizing a run-time compilation strategy allowed for the inclusion of one additional feature, building a kernel from a string rather than from a file. Compiling from a string is essentially equivalent to compilation from a file; however, by providing this feature without requiring users to safely handle file generation, projects developed with OCCA gain extended flexibility. Code listing 2.15 provides an example where a user can create a skeleton kernel with a user-specified operation on arrays.

```
occa::kernel arrEq2Arr(occa::device &device,
                       const std::string &aType, const std::string &aName,
                       const std::string &bType, const std::string &bName,
                       const std::string &cType, const std::string &cName,
                       const std::string &aEqFbc){

  std::stringstream ss;
  ss << "occaKernel void arrEq2Arr(occaConst int occaVariable entries,\n"
     << "                                 occaPointer " << aType << " *_" << bName ",\n"
     << "                           occaConst occaPointer " << bType << " *_" << bName ",\n"
     << "                           occaConst occaPointer " << cType << " *_" << cName "){\n"
     << "  occaOuterFor0{\n"
     << "    occaInnerFor0{\n"
     << "      const int n = (occaOuterId0 * occaInnerDim0) + occaInnerId0;\n"
     << "      if(n < entries){\n"
     << "        " << aType << ' ' << aName << ";\n"
     << "        " << bType << ' ' << bName << " = _" << bName << "[n];\n"
     << "        " << cType << ' ' << cName << " = _" << cName << "[n];\n"
     << "        " << aEqFbc << ";\n"
     << "        _" << aName << "[n] = " << aName << "\n"
     << "      }\n"
     << "    }\n"
     << "  }\n"
     << "}\n";

 return device.buildKernel(ss.str(), "arrEq2Arr");
}

// ...

occa::device device;

occa::kernel sumArrays = arrEq2Arr(device,
                                   "float", "a",
                                   "float", "b",
                                   "float", "c",
                                   "a = (b + c)");

occa::kernel multiplyArrays = arrEq2Arr(device,
                                        "float", "a",
                                        "float", "b",
                                        "float", "c",
                                        "a = (b * c)");
```

Listing 2.15: The code listing shows the use of kernel building from string to provide a templated `occa::kernel`.

Caveats to the run-time compilation model arose while working with industrial codes and their respective computational clusters. The first issue became apparent due to the amount of time spent continuously compiling all kernels at run-time each time an application is run. Additional issues emerged with the run-time compilation model including: potentially time-consuming kernel setup; network file system (NFS) issues with parallel compilations; overloading the compiler license manager; lacking a compiler outside login nodes in supercomputers/clusters; and safety in releasing an application without revealing the source code.

Kernel caching was implemented to provide a solution towards the continuous kernel compilation setup when an application using OCCA runs. When a kernel is com-

piled, the binary is saved and accessible through a hash of its content and compilation information. For example, the OCCA backend and compilation flags used to generate the kernel binary are included along with the kernel content when producing the hash. Prior to building a kernel, a search is launched for the kernel's generated hash where the kernel binaries are cached. If found, the binary is used to avoid the unnecessary compilation; otherwise, the kernel is compiled and it's binary cached.

Two additional complications surfaced when running OCCA on computational clusters at an industrial site. The first issue came from running an MPI-based job and caching the kernels within the cluster's NFS. Each MPI process attempted to compile a kernel and would overwrite the intermediate OCCA IR files generated, causing compilations (and thus the application) to fail. Restricting each MPI process to compile in separate directories was a simple fix which exhibited the second problem with running OCCA on an NFS. Using a proprietary compiler controlled by a license manager, such as Intel's `icpc`, would result in failure due to the large query of license instances. Adding an NFS-safe lock for kernel generation fixed both complications through the use of file locks.

The use of supercomputers in industrial and academic settings revealed another issue with run-time compilation in computation clusters. Users accessed the cluster through a login node which, while unable to run application codes, allowed for jobs to be queued into the cluster. When an application is compiled and sent to the cluster for execution, problems arise when the application requires a compiler for building kernels at run-time. A prototype for compiling kernels at compile-time was developed, where kernels would be built for available devices and packaged with the application binary.

The last problem became relevant through inquiries from industrial collaborators: how OCCA could be used without releasing the application's source code. While the

amount of code used in OCCA kernels would most likely be a small fraction of the overall application, OCCA kernels usually target the computational intensive portions of an application, customarily containing the core application code. Although OCCA kernel binaries can be bundled together with the main application binary, they have yet to be unitized and analyzed for production. A prototype was developed for compiling kernels in the available backends and archived into a single database. Rather than building the kernels from source, we provide the following call to allow users to load from the produced database, or library.

```
kernel& loadFromLibrary(const char *library,
                        const std::string &functionName);
```

2.3.3 High Performance Computing Features

The OCCA library development has been and will continue to be in C++, restricting user applications to C++ if they use OCCA. Therefore, I added a C wrapper for OCCA which enabled quick native distribution to other languages. Subsequently, the OCCA API is available now with library interfaces for C, C++, C#, Fortran, Python, Julia and MATLAB. Additionally, I focused on backend support and added two modes to support multithreading through Pthreads and Intel's Xeon Phi backend, COI (Intel's Coprocessor Offload Infrastructure).

Supporting multiple frontend and backend features enables programmers to use a wide variety of programming languages to use OCCA, yet it maintained to be an intrusive library. By obscuring object handles from each backend, the user was forced to use OCCA disjointly from other libraries, which was one of the issues with the wrapper approach discussed in chapter 1. I added methods to wrap and obtain backend related object handles to give users the ability of combining OCCA seamlessly with pure usage of the backends and other libraries. Thus, a developer could reuse existing objects

with OCCA and avoid copying the object into memory twice as shown in code listing 2.16.

```
// Initialize device
occa::device device("mode = Serial");

// Allocate memory in the host
const int N = 10;
int *a = new int[N];
// Wrap a into an occa::memory object
occa::memory o_a = device.wrapMemory(a, 10 * sizeof(int));
```

Listing 2.16: The code listing shows the use of memory wrappers to reuse memory objects from other libraries.

2.4 Concluding Remarks

This chapter described the abstracted programming model targeting manycore devices and the OCCA IR specification to test the model. In addition, the API implementation to manage the discussed abstract offload model was included. Advantages for the run-time compilation model were outlined while describing ways to resolve its deficiencies. Although OCCA IR was itself a functional unified kernel language, it also serves as the foundation for two more kernel languages whose specifications are introduced in the following chapter. By developing a parser which serves for transforming and analyzing OCCA kernels, I was able to extend the C and Fortran languages and transform them to the OCCA IR. Rather than programming with the OCCA IR, the introduced languages OKL (C-based) and OFL (Fortran-based) allow for native-like kernels.

Currently, the OCCA IR contributes as an intermediate layer to facilitate OKL and OFL code transformation. Further development on the OCCA IR will be most likely limited to adding arithmetic support functions such as trigonometric and special functions. Additional features will presumably be implemented through the OKL and

OFL specifications to further ease kernel development.

3 OKL and OFL: OCCA Kernel Languages

The contributions depicted in this chapter center around a language design for facilitating low-level programming of manycore devices. Two implementations were developed to extend C and Fortran, prominent programming languages in the high performance computing (HPC) community. The OCCA *kernel language (*OKL*) is based on C and the* OCCA *Fortran kernel language (*OFL*) is based on Fortran 90. The language designs are built above the* OCCA *intermediate representation (IR) from chapter 2. To realize these kernel languages, I designed a parser for C and Fortran with core extensions in the language specifications for parallel execution.*

In this chapter, I discuss a design for unifying parallel languages and standards for heterogeneous computing. The language design described in this chapter is built above the OCCA intermediate representation (OCCA IR) from chapter 2. Using the underlying OCCA IR, I present OKL (OCCA Kernel Language), a minor proposal to extend C, exposing parallelism for current many-core architectures. Similarly, I present OFL (OCCA Fortran Language) as a minor extension to Fortran, the Fortran counterpart to OKL. I have designed a C and Fortran parser with core extensions in the extended language specifications for parallel execution.

I begin by discussing some compiler-aided tools and the compiler-based approach taken for the development of OKL and OFL. Following the description of development tools, I introduce the OKL language specifications and its mapping to the OCCA IR. Lastly I introduce OFL by noting structural differences between OKL and OFL.

During the initial process in designing OKL, the specification was constantly being modified due to inspirations for useful features for this new language. It became apparent that full control over the language would be the fastest way for implementing the OKL language, specially under the time constraints for the thesis work. Thus, I implemented a C parser that would structure parsed code such that the generated abstract syntax tree (AST) was highly modifiable. For example, the parser can easily patch run-time generated source code into the AST and analyze constituents of expression trees to generate variable dependency graphs. With the parser available, I designed OFL and by reusing the tools found in C parser, implemented features required to load Fortran and store C-equivalent statements during the parsing stage. Therefore any updates to OKL will automatically affect OFL, offering support to both, the C and Fortran community. An additional motivation to develop a parser was to maintain the open-source OCCA library without additional library dependencies aside from the enabled backends.

3.1 Compiler Tools

Various tools for parsing languages have been made available, such as the gcc-extension MELT (Starynkevitch 2011), LLVM's tool-chain used in clang (Lattner and Adve 2004, Lattner 2002), the ROSE compiler from Lawrence Livermore National Laboratory (Quinlan 2000) and others (Johnson 1975). Because this thesis proposal focuses more on the language than compiler tools, for a more in-depth explanation in the compilation process the reader is referred to the similar tools mentioned or compiler development sources (Cooper and Torczon 2011, Turbak 2008). This section, however, will briefly discuss the approach taken for developing OKL and OFL, starting with language parsing and language-specific features. Code transformation between the kernel languages to the OCCA IR occurs through four steps:

- Merging and splitting file by lines while preprocessing macros and directives
- Parsing lines into tokens representing preset-values such as numbers or strings, unknowns and built-ins
- Detecting the statement type in each line and generating an expression tree from its tokens
- Code transformations of OKL and OFL to the OCCA IR

For simplification, discussions will focus on the C-syntax and OKL unless explicitly stated.

3.1.1 Preprocessor

A working C-preprocessor was implemented to maintain compatibility with C, usable in OKL and OFL. Code is scanned in a top-down fashion, merging lines containing escape characters and progressively removing comments and comment blocks from segments or complete lines. Macros and directives are then loaded and applied to the resulting lines of code.

3.1.2 Parsing and Tokenization

Following the preprocessor step, the code goes through the second step of code transformation by tokenizing the resulting lines of code. Each line is partitioned into tokens comprising of numbers (integers or floating-precision constants), string segments surrounded by quotes, and special keywords such as operators, types, and built-ins. Aside from keeping line numbers for debugging purposes, all tokens are then stored in a linear format since effects from new-line characters were processed prior to this stage.

Number detection is done by detecting either `[0-9]` or `.[0-9]`. We purposely

ignore the `-` and `+` unary symbols since they are later handled by the expression tree formatting. Number loading, however, takes the form:

```
INT[.][e[+-]INT][fFlL],
```

where `INT` represents `[0-9]*` and content inside `[]` is optional.

We skip loading string segments since they are detected by pairs of quotes. Special keywords, such as operators, types, and built-ins, are mapped prior parsing to the representation type or types. For instance, the `*` operator can be considered as a multiplication operator or dereference operator Similarly, `long`, `short`, `signed` and `unsigned` can be used as both, qualifiers and specifiers. During this stage, all types are stored and later resolved when building the expression trees.

To fully support the Fortran-community, we handle part of the Fortran-to-C translation during this phase by converting Fortran such as operators (`.LT.`, `.GT`, ...) and `.NOT.` and literals (`.TRUE.`, `.FALSE.`). Operators such as the power operator denoted by `**` in Fortran are transformed to the proper `pow()` functions in the fourth stage of the code transformation. Because white-space is important in Fortran, we add a token to signify when an end-of-line took place and maintain a linear chain of tokens after this parsing phase.

3.1.3 Statement Labeling

At this point, each line has been tokenized and labeled into a sequence of tokens with minor code transformations between Fortran to C. By doing a left-to-right search on the tokens, we're able to label sequences of tokens as statement types, including:

- Declaration statements (e.g. `int x;`, `int x = 0, y = 1;`)

- Update statements (e.g. `x = 0;`, `x = y = pow(1,1);`, `kernelCall(x,y);`)
- Flow-control statements (`for`, `while`, `do`, `if`, `else if`, `else`, `switch`, `goto`)
- Function declaration and definition statements (`int one();` and `int one(return 1;)`)
- Block statements (e.g. {`int x;`})
- Struct definitions (e.g. `struct int2 { int x, y; };`)

For example, statements beginning with the token:

`for`, `while`, `do`, `if`, `else if`, `else`, `switch`, `goto`

are taken to be

- For loop
- While loop
- Do-while loop
- If, else-if, else statements
- Switch statement
- Goto label

respectively. More complex statement labeling can be found when scanning keywords such as:

`struct`, `union`, `enum`

which could be used to provide extra information about the following type or could be used to begin a new type definition as seen in code listing 3.1.

```
struct int2 { int x, y; };
struct int2 int2Var;
```

Listing 3.1: Keywords such as `struct` could be used on different statement types

All tokens making up a statement are initially stored and later processed into expression trees. Statement objects store a reference to its parent statement and all nested statements through a linked-list as seen in figure 3.1, allowing for simple modifications to the abstract syntax tree (AST) through API calls.

```
if(N < 256){
  for(int i = 0; i < N; ++i; outer0){
  }
}

N = 0;
```

3.1.4 Expression Trees

After a statement is detected, the consequent tokens are used to build expression trees to describe the statement. A statement might contain multiple expression trees to ease data analysis and extraction. For example, the OKL for-loop

```
for(int i = 0; i < N; ++i; outer0)
```

would be expressed by four expression trees as seen in figure 3.2. The linear chain of tokens is organized with respect to operator precedence from the C and Fortran specifications. Similarly, custom types and variable definitions are loaded and stored separately as the expression trees are being built, rather than using basic tokens to express

```
int i;
```

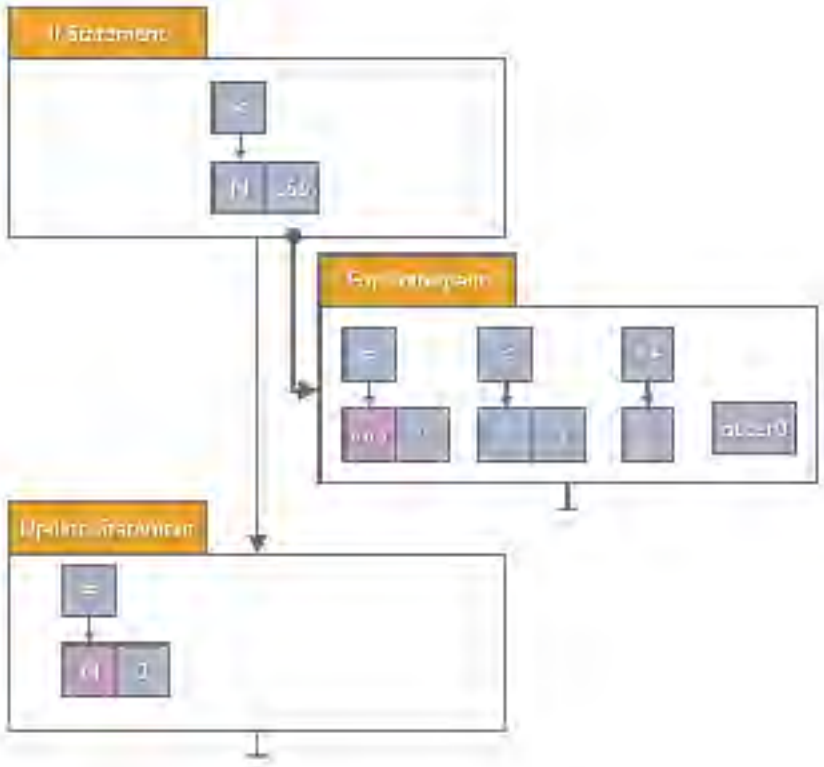

Figure 3.1: Statements are stored by linked-list nodes, containing it's parent, next node, and contained nodes. Depending on the statement type, each statement is contained by one or many expression trees for simplification.

as seen in figure 3.2. Modifying and finding data types becomes simple and helps the modification of OKL and OFL code to the OCCA IR.

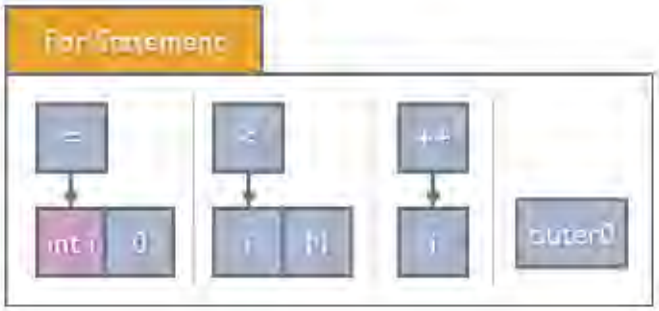

Figure 3.2: The OKL for-loop statement is described by four expression trees for keeping edits to the AST simple. Likewise, variable and type definitions are stored through their native storage class rather than a multi-node branch in the expression tree.

A few modifications are handled at this step when translating from Fortran to C. Fortran multidimensional arrays are flattened and their strides are stored for updating array accesses. Another difference comes from the operator `**` which is not defined in C and hence is replaced by a `pow()` call.

3.1.5 Types and Variable Information

The initial parser prototype only used tokenization and statement labeling, using metadata for distinguishing between statement types and the tokenization to describe each statement. Analysis quickly became complex and thus the expression trees were used, facilitating data management for statements and proper ordering of operations rather than using a left-to-right token chain. However, qualifiers, types, variables, and functions were still maintained through tokens in expression trees. While metadata for each token supplied information of the token type, fetching variable or function information also become convoluted. Hence, types and variables were stored in special tokens inside expression trees.

Code listing 3.2 shows the original tokenization (`OLD`) compared to the current

implementation (NEW). Combining the statement and token metadata to provide printing information, we're able to use references to variables and types for facilitating code analysis.

```
const int i = j;
// OLD: [const] [int] [i] [=] [j]
// NEW: [varInfo: const int i] [=] [varInfo: j]

typedef struct {
  int x, y, z;
} int3;

// OLD: [typedef] [struct] [{] [int3]
//                          v
//                         [int] [x] [,] [y] [,] [z] [;]
//
// NEW: [typeInfo: typedef struct { int x, y, z } int3] [;]
```

Listing 3.2: This code listing shows two simple statements with their original tokenization and updated parser format.

3.2 OKL Specification and Features

This section will introduce the mapping between OKL to the OCCA IR. Although we use the developed parsing tool, OKL was designed to try and achieve minimal extensions to the native C languages respectively. I discuss some language benefits through the use of the parser, including dynamic work-range/work-group setup, multiple kernel instances, automatic barrier detection for shared/global memory.

3.2.1 Exposing Parallelism

Chapter 2 discusses the approach taken to unify multithreading platforms by exposing parallelism. Rather than using the `occaFor` macros, OKL extends the C for-loops by adding a fourth statement as seen in code listing 3.3 Aside from showing the explicit loops, OKL now has the feature to automatically assign working dimensions to the offload model through the outer and inner loops. A kernel launch in OpenCL and

CUDA are always separate from the kernel source, but maintain a connection through the working dimensions used in a kernel execution. Errors due to a mismatch between a kernel and kernel launch are now resolved solely in the outer-loop source code.

```
kernel void kern(const int N){
  for(int group = 0; group < N; group += 16; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      // Work-item body here
    }
  }
}
```

Listing 3.3: OKL extends native C for-loops by using the fourth statement to tag the type of loop

The start, end, and strides used in the outer and inner loops support argument-based variables and the working dimensions are resolved at run-time. Currently, working dimensions must stay constant across on all inner-loops defined in a single outer-loop; dynamic working dimensions is work in progress. Additionally, OKL loop iterators can be dependent on each other only if the working dimensions stay constant. For example, `kern` seen in code listing 3.3 could be reinterpreted in code listing 3.4.

```
kernel void kern(const int N){
  for(int group = 0; group < N; group += 16; outer0){
    for(int item = group; item < (group + 16); ++item; inner0){
      // Work-item body here
    }
  }
}
```

Listing 3.4: OKL loop iterators can only be dependent if the working dimensions for the inner-loops do not change.

Code listing 3.5 shows the exact transformation between the gathered loop bounds from code listing 3.4 and the OCCA IR counterpart. The bounds and their dependencies are appended to the host-kernel which in turn uses those dependencies to setup the `occa::kernel`'s and launch them. Because the work dimensions for nested

`occa::kernel`s are gathered from the bounds, it is apparent that the kernel bounds contain restrictions. For example, bounds cannot be adjusted through the use of a for-loop as seen in code listing 3.6 although for-loops are still supported in OKL and OFL. Additionally, because loop bounds are obtained and used in the host kernel, accesses from pointer kernel arguments cannot be used for deriving inner and outer loop bounds.

```
occaKernel void kern(occaKernelInfoArg, occa::kernel *nestedKernels, occaConst int
    occaVariable N) {
  {
    const int dims = 1;
    occa::dim outer, inner;
    outer[0] = (((N) - (0) + ((16) - 1)) / ((16)));
    occaConst int group = 0 + (0 * (16));
    inner[0] = ((((group + 16)) - (group) + ((1) - 1)) / ((1)));
    nestedKernels[0].setWorkingDims(dims, inner, outer);
    nestedKernels[0](N);
  }
}

occaKernel void kern0(occaKernelInfoArg, occaConst int occaVariable N) {
  occaParallelFor0
  occaOuterFor0{
    occaConst int group = 0 + (occaOuterId0 * (16));
    occaInnerFor0{
      occaConst int item = group + occaInnerId0;
      // Work-item body here
    }
  }
}
```

Listing 3.5: This code listing shows the OKL to OCCA IR transformation displayed in code listing 3.4.

```
kernel void kern(const int N){
  for(int group = 0; group < N; group += 16; outer0){
    for(int innerSize = 0; innerSize < 16; ++innerSize){
      for(int item = group; item < (group + innerSize); ++item; inner0){
        // Work-item body here
      }
    }
  }
}
```

Listing 3.6: This code listing shows a combination of for-loops and OKL inner loops which are not supported due to the bounds depending on a changing variable.

Currently, the only tags introduced as the fourth statement in for-loops used to expose parallelism include `outerX` and `innerX` where `X` $\in$ {`0, 1, 2`}. These `outer` and `inner` tags were derived from the `occaOuterFor` and `occaInnerFor` loops found in the OCCA IR. An additional `tile` tag was introduced to facilitate kernel development due to many kernels only requiring the use of simple bounds and iteration strides. Code listing 3.7 shows two examples of the `tile` tag, tiling for-loops as one and two dimensional sets of inner/outer loops. The fourth statement could be used for further describing work inside for-loops in future architectures and programming models.

```
kernel void kern(const int N){
  //---[ 1D tile loop ]-----------------
  for(int i = 0; i < N; ++i; tile(16){
  }

  //---[ 1D tile loop expansion ]------
  for(int o = 0; o < N; o += 16; outer0){
    for(int i = o; i < (o + 16); ++i; inner0){
    }
  }

  //---[ 2D tile loop ]-----------------
  for(int2 i(0,0); i.x < N, i.y < N; ++i.x, ++i.y; tile(16,16){
  }

  //---[ 2D tile loop expansion ]------
  for(int o1 = 0; o1 < N; o1 += 16; outer1){
    for(int o0 = 0; o0 < N; o0 += 16; outer0){
      for(int iy = o; iy < (o1 + 16); ++iy; inner1){
        for(int ix = o; ix < (o0 + 16); ++ix; inner0){
          int2 i(ix,iy);
        }
      }
    }
  }
}
```

Listing 3.7: This code listing shows the use of the `tile` tag as outer-loops.

Introducing explicit outer and inner loops facilitates development by facilitating the comprehension of the kernel code for the programmer and future readers of the code. We further augment kernel development by supporting loops and flow control statements containing outer and inner-loops; an example is displayed in code listing 3.8. Similarly, multiple outer-loops are supported inside a single kernel. An example is

given in code listing 3.9 which displays two sets of outer-loops followed by a simplified code listing that would be generated from the OKL parser. Restrictions on for-loops follow those from outer and inner loops where values from array entries cannot be used as loop bounds.

```
kernel void kern(const int iterations, const int N){
  for(int iteration = 0; iteration < iterations; ++iterations){
    if(iteration < N){
      for(int group = 0; group < N; group += 16; outer0){
        for(int innerSize = 0; innerSize < 16; ++innerSize){
          for(int item = group; item < (group + innerSize); ++item; inner0){
            // Work-item body here
          }
        }
      }
    }
  }
}
```

Listing 3.8: This code listing shows a combination of for-loops and conditionals containing OKL outer-loops.

```
kernel void kern(const int N){
  // True kernel 1
  for(int group = 0; group < N; group += 16; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      // Work-item body here
    }
  }

  // True kernel 2
  for(int group = 0; group < N; group += 16; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      // Work-item body here
    }
  }
}

//---[ Expands to ]---------------------

// Host function
void kern(const int N){
  kern1(N); // Call OKL kernel 1
  kern2(N); // Call OKL kernel 2
}

// OKL kernel 1
kernel void kern1(const int N){
  for(int group = 0; group < N; group += 16; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      // Work-item body here
    }
  }
}

// OKL kernel 2
kernel void kern2(const int N){
  for(int group = 0; group < N; group += 16; outer0){
    for(int item = 0; item < 16; ++item; inner0){
      // Work-item body here
    }
  }
}
```

Listing 3.9: The code listing displays pseudocode of the transformation occurring when an OKL kernel `kern` contains multiple OKL-tagged loops. The code transformation shown below contains the host kernel `kern` which calls the separated kernels `kern1` and `kern2`.

3.2.2 Memory Types

Two memory types are introduced through the OCCA IR, `occaShared` and `occaPrivate`. In OKL, we introduce these memory types through the qualifiers `shared` and `exclusive`, respectively, due to `private` already being a C keyword. Rather than using non-

intuitive macros, their OKL and OFL counterparts enable a more native approach to decorating variables. A comparison between the OCCA IR, OKL, and OFL can be seen in code listing 3.10.

```
//---[ OCCA IR ]----------------------
occaShared float s_A[16];
occaPrivate(float, p_A);
occaPrivateArray(float, p_B, 10);

//---[ OKL ]--------------------------
shared float s_a[16];
exclusive float p_A, p_B[10];

//---[ OFL ]--------------------------
real(4), shared    :: s_a(16)
real(4), exclusive :: p_A, p_B(10)
```

Listing 3.10: This code listing shows a comparison between the use of `occaShared` and `occaPrivate` in the OCCA IR, and their respective `shared` and `exclusive` qualifiers in OKL and OFL.

A large caveat using `occaPrivate` in the OCCA IR for CPU-modes, such as Serial, OpenMP, Pthreads, and COI, is the lack support for structs. Because `occaPrivate` is implemented as a class, accessing member variables or functions with the underlying type would result in a class-formatting error. An `occaPrivate` variable cannot support structs, making certain important data types for HPC inaccessible, such as float2 and float4. The CPU-based implementation for `exclusive` uses linear arrays, similar to the `occaPrivate` class, which transforms variable accesses into a work-item's proper array access. By exposing the underlying array and treating `exclusive` variables as linearly iterating accesses in an array, the compiler vectorization is facilitated for vectorizing operations on exclusive variables.

Additional benefits arise from using code transformations to deal with the different memory types. In GPU-based architectures, shared memory guarantees the use of cache for data storage while expecting cache reuse in CPU-based modes. For the upcoming Intel's Knights Landing (Bender et al. 2015), a special scratchpad of memory in the

processor will become available. Although the latency and bandwidth is inferior to cache, the parser can make use of it automatically through pre-allocating memory needed for shared arrays in OKL and OFL kernel. Adjusting to future architectures becomes easier through the introduction of custom qualifiers, allowing programmers to further decorate code to express their programming intentions.

3.2.3 Device Functions

There are two types of functions in OKL, a function called from an inner-loop and another called from an outer-loop. An example is given in code listing 3.11 where function `groupReduce256` is designed to be called from an outer-loop and function `one` is called from an inner-loop. Automatic differentiation between device function scopes allows for additional code reuse.

```
// (1) Device function with a work-group scope, launching an inner-loop
int groupReduce(int *ptr, int N){
  for(int off = N/2; 0 < off; off /= 2){
    for(int i = 0; 0 < N; ++i; inner0){
      if(i < off)
        ptr[i] += ptr[i + off];
    }
  }
}

// (2) Device function with a work-item scope
int one(){
  return 1;
}

// Assume ASize is given through the
//   occa::kernelInfoArg

kernel void kern(int *A){
  for(int o = 0; o < 1; ++o; outer0){
    shared int s_A[ASize];

    for(int i = 0; i < N; ++i; inner0){
      // one() is used inside an inner-loop
      s_A[i] = one();
    }

    // Example of an inner-loop function
    //   called inside an outer-loop
    groupReduce(s_A, ASize);
  }
}
```

Listing 3.11: Device functions can be called with a work-group scope (1) or a work-item scope (2).

Dynamic inner-loop sizes are currently not supported, where different inner-loops inside a group of outer-loops contain varying bounds. Detection of inner-loop bounds inside inner-loop device functions such as `groupReduce` in code listing 3.11 is also not currently supported. Basic analysis on kernel code is the extent of the parser, hence an inner-loop must be present inside an outer-loop. Future work on inlining non-recursive functions will permit inner-loop functions to be used more freely. Likewise, adding varying bounds to kernel inner-loops will eliminate the restriction of entrusting programmers to maintain identical bounds on an outer-loop's inner-loop bounds. Enabling dynamic inner-loop bounds will be solved by adding checks inside inner-loops based on their bounds and the maximum inner-loop bounds as shown in code listing 3.12.

```
kernel void kern(int *A, int N){
  for(int o = 0; o < 1; ++o; outer0){
    for(int iSize = 0; iSize < N; ++iSize){
      for(int i = 0; i < iSize; ++i; inner0){
        // Inner-loop work
      }
    }
  }
}

// Transformed into

kernel void kern(int *A, int N){
  for(int o = 0; o < 1; ++o; outer0){
    for(int iSize = 0; iSize < N; ++iSize){
      for(int i = 0; i < N; ++i; inner0){
        if(i < iSize){
          // Inner-loop work
        }
      }
    }
  }
}
```

Listing 3.12: This code listing shows a proposed solution for enabling dynamic inner-loop bounds.

3.3 OFL Specification

The development of OFL was motivated by the limited support for Fortran to program accelerators, even though Fortran is a prominent language in the high performance community. This section will introduce the OFL specification and mapping between OFL and OKL. Using the code organization in the OKL parser, a Fortran to C translator was built-in to reuse the code transformations supported by OKL. Hence, features added to OKL will therefore be automatically enabled in OFL.

3.3.1 Exposing Parallelism

The same concept of using tags on for-loops is taken in OFL. By tagging `DO` loops in OFL, the programmer exposes parallel regions of code to the parser. An example is

given in code listing 3.13

```
kernel subroutine kern(N){
  integer(4), intent(in) :: N

  integer :: group, item

  do group = 1, entries, 16, outer0
    do item = 1, 16,  inner0
      ! Work-item body here
    end do
  end do
end subroutine kern
```

Listing 3.13: Shown in the code listing is the use of OFL tags on DO loops to expose parallelism. The OKL counterpart to this code listing can be seen in code listing 3.3

Issues quickly arose regarding variable dependence across kernels when designing OFL. Fortran uses constant bounds when launching DO loops, even if the variables introduced inside bounds are updated inside the loop. For example, the translation for a DO loop in OFL can be seen in code listing 3.14. In order to handle the variable dependencies for each outer-loop-generated kernel, a data dependency graph is generated to find each outer-loop's dependent statements. Additionally, modifications of variables inside outer-loops cannot be propagated across outer-loops due to the programming model.

```
INTEGER :: I

DO I = 1, N, 2, OUTER0
END DO

DO I = 1, N, OUTER0
END DO

//---[ Translates to ]-------------------------------------------

int I;

const int doStart0      = 1;
const int doEnd0        = N;
const int doStride0     = 2;
const int doStrideSign0 = (1 - (2*(doStride0 < 0)));

for(I = doStart0; 0 <= (doStrideSign*(doEnd0 - I)); I += 2; outer0){
}

const int doStart1 = 1;
const int doEnd1   = N;

for(I = doStart1; I <= doEnd1; ++I; outer0){
}
```

Listing 3.14: `DO` loops in OFL use constant evaluations for it's iterator's initial value, stride and final value check. The translation to OKL can be seen in this code listing

3.3.2 Memory Types

The qualifiers to label variables to use distinct memory types are synonymous to those found in OKL. Both `shared` and `exclusive` are supported in OFL and can be implemented as such:

```
integer(4), shared    :: sharedVar
integer(4), exclusive :: exclusiveVar
```

Some language differences between Fortran and C have been identified and resolved. For example, Fortran natively supports multidimensional arrays:

```
integer(4), shared :: sharedVar(X,Y,Z)
```

The parser resolved this issue by using the stored stride information and flattening the multidimensional arrays when called. Therefore, array accesses such as

```
integer(4), shared :: sharedVar(X,Y,Z)
sharedVar(1,2,3) = 0;
```

are converted to

```
shared int sharedVar[X*Y*Z];
sharedVar[3*(X*Y) + 2*X + 1] = 0;
```

3.4 Support for CUDA and OpenCL

With the parser available, it was of interest to provide additional capability to support OpenCL and CUDA kernel languages. We discussed OKL as a language extension for C and now introduce support for OpenCL and CUDA. Because OCCA was derived from GPU-languages and both OpenCL and CUDA extend the C language, it's possible to convert OpenCL and CUDA to the OCCA IR. A prototype was developed which would properly translate simple kernels with support for GPU memory types, barriers, and function calls.

The code transformation contains some challenges, many which were embedded in the OCCA IR, such as `occaPrivate` or `exclusive` in OKL. For example, the translation from code listing 3.15 to code listing 3.16 shows the effect when a variable's scope crosses a barrier. The prototype implemented can handle such detections and transformations.

```
__kernel__ void kern(__global float *A){
  int r_A = 0;
  // ...
  barrier(localMemFence);
  // ...
  r_A = A[get_global_id(0)];
  // ...
}
```

Listing 3.15: Simple OpenCL example with a variable's scope crossing barriers

```
occaKernel void kern(occaKernelInfoArg, occaPointer float *A){
  occaOuterFor0{
    occaPrivate(int, r_A);

    occaInnerFor0{
      r_A = 0;
      // ...
    }

    occaBarrier(occaLocalMemFence);

    occaInnerFor0{
      // ...
      r_A = A[occaOuterId0 * occaInnerDim1 + occaInnerId0];
      // ...
    }
  }
}
```

Listing 3.16: Translation of code listing 3.15 to the OCCA IR. Note the transformation applied to `r_A` due to it's scope crossing a barrier.

Properly transforming loops and conditional statements to the OCCA IR representation remains a complication for portable performance. Using a GPU-based backend on the OCCA IR kernels generated from CUDA or OpenCL will maintain a one-to-one mapping. Different loop orderings on a CPU-based backend, however, changes performance of the kernel. While OCCA does not guarantee portable performance, it is of interest to maintain properly mapped code transformations between CUDA and OpenCL to the CPU counterparts. Different injections of `occaInnerFor` loop translations of code listing 3.17 are shown in code listing 3.18, code listing 3.19, and code listing 3.20. Currently, option `(1)` in code listing 3.18 is chosen by default, but future work could include detection operation count and estimate which permutation generates better vectorization by the compiler.

```
__kernel__ void kern(__global float *A){
  const int id = get_global_id(0);
  int a = 0;

  if(id % 2)
    a = 1;

  for(int i = 0; i < 16; ++i)
    a += A[16 * id + i];

  A[16 * id] = a;
}
```

Listing 3.17: This code listing displays an example CUDA kernel used for displaying multiple combinations of inserting OCCA IR inner-loops inside for-loops found in code listing 3.18, code listing 3.19, and code listing 3.20.

```
// (1) Loops inside occaInnerFor loops
occaKernel void kern(occaKernelInfoArg, occaPointer float *A){
  occaOuterFor0{
    occaPrivate(int, id);
    occaPrivate(int, a);

    occaInnerFor0{
      id = (occaOuterId0 * occaInnerDim0) + occaInnerId0;
      a  = 0;
    }

    occaInnerFor0{
      if(id % 2)
        a = 1;
    }

    for(int i = 0; i < 16; ++i){
      occaInnerFor0
        a += A[16 * id + i];
    }

    occaInnerFor0{
      A[16 * id] = a;
    }
```

Listing 3.18: This code listing displays combination 1 of translating code listing 3.17 to OCCA IR.

```
// (2) Only loops inside occaInnerFor loops
occaKernel void kern(occaKernelInfoArg, occaPointer float *A){
  occaOuterFor0{
    occaPrivate(int, id);
    occaPrivate(int, a);

    occaInnerFor0{
      id = (occaOuterId0 * occaInnerDim0) + occaInnerId0;
      a  = 0;

      if(id % 2)
        a = 1;
    }

    for(int i = 0; i < 16; ++i){
      occaInnerFor0
        a += A[16 * id + i];
    }

    occaInnerFor0{
      A[16 * id] = a;
    }
```

Listing 3.19: This code listing displays combination 2 of translating code listing 3.17 to OCCA IR.

```
// (3) Loops and conditionals inside occaInnerFor loops
occaKernel void kern(occaKernelInfoArg, occaPointer float *A){
  occaOuterFor0{
    occaInnerFor0{
      const int id = (occaOuterId0 * occaInnerDim0) + occaInnerId0;
      int a = 0;

      if(id % 2)
        a = 1;

      for(int i = 0; i < 16; ++i)
        a += A[16 * id + i];

      A[16 * id] = a;
    }
  }
}
```

Listing 3.20: This code listing displays combination 3 of translating code listing 3.17 to OCCA IR..

3.5 Concluding Remarks

Two kernel language specifications based on the OCCA IR discussed in chapter 2 were presented: OKL (OCCA Kernel Language) and OFL (OCCA Fortran Language). Both kernel languages adequately extend the C and Fortran languages to expose parallelism found in current many-core architectures. A parser was developed to explore and analyze C, assembled to simplify code analysis, code transformations, and code generation. Kernels written in OFL are translated from C to Fortran by the parser, guaranteeing code transformations present in OKL will be automatically applied to the OFL specification. Developing a custom parser allowed for the rapid prototyping of the OKL and OFL specifications while being sufficiently modular to amend the specifications. Future work on the specifications will focus on facilitating kernel development, for example the use of attributes discussed in Subsection 5.2.3. The goal is, however, to incorporate OKL and OFL kernels with their respective native languages, incorporating kernels with the application code. Steps toward blending kernels with the *host* code have been developed for automatic data movement (chapter 4) and automatic kernel generation (chapter 5).

4 Automated Data Movement

Contributions discussed in this chapter target the automation of data movement in the abstract offload model used in OCCA*. The abstract offload model is augmented through the use of a unified memory address space. An implementation of the proposed automation of data movement is provided in the* OCCA *API.*

Chapter 2 focused on developing an intermediate representation for a unified kernel language, followed by chapter 3 which introduced OKL and OFL kernel languages to facilitate kernel development. This chapter discusses the second layer of assistance towards developers by providing an option to automate data movement between the *host* and *device*. By adding the feature to remove memory management between the host and device, we can provide an environment for easing programming for the offload model and allow for more rapid prototyping. I include current ways to achieve automatic data movement and the proposed method for the thesis work. Likewise, I indicate the additional control over the automated data movement through the OCCA API.

4.1 Automated Data Movement Approaches

Using the offload model in OCCA, we assume the host and device are separate entities and thus assume their memory spaces are disjoint. We wish to emulate unified memory that will treat memory from the host and device through a single address space, approaches found in CUDA and tentatively in the OpenCL 2.0 standard. Completely

hiding the distinction between host and device memory would further simplify the offload model. I will discuss two approaches that make it possible to hide device memory through emulating a unified address space.

The first approach that has automated data movement mimics the use of OpenACC and OpenMP 4.0 directives. By default, data is always copied to and from the device for each parallel region. However, the user can specify whether the respective data is allocated, copied and/or written before and after each parallel region as seen in code listing 4.1. We avoid this approach to avoid the user from having to specify address ranges and data transfers prior to each kernel call.

```
#pragma omp target map(from:a[0:entries]) \
                   map(to:b)              \
                   map(to:c[0:entries])
{
#pragma omp parallel for
  for(i = 0; i < entries; ++i){
    a[i] = b * c[i];
  }
}
```

Listing 4.1: Code listing of an OpenMP 4.0 example showing manual data transfers and ranges

The second approach discussed is taken by CUDA, and tentatively the OpenCL 2.0 standard. The `cudaMallocManaged()` in CUDA's specifications returns a host-pointer that can be passed to CUDA kernels. By default, the host has ownership over managed pointers but loses said ownership between a CUDA kernel launch using it's mapped device memory and a `cudaDeviceSynchronize()`. This approach relies on full data transfers across the host and device during kernel launches and device synchronization if needed. In comparison, the directive approach can reduce redundant data transfers by labeling "dirty" regions that require synchronization at the price of having the programmer input the required mapping for each address-range used. Aside from a few optimizations, such as checking which address ranges were declared constant, automatic data transferring would not be used for high performance computing. However, the

reasoning of automated memory management is to ease programmers into developing for the heterogeneous programming model and for prototyping, where efficiency is of less importance.

4.2 Emulating Unified Memory

We adopt a similar approach seen in CUDA's `cudaMallocManaged()` by maintaining ownership information in our map between virtual addresses to their respective `occa::memory` objects. An implementation is discussed in this thesis and an implementation will be provided for the final thesis work, together with benchmark comparisons discussed in chapter 6.

To first hide the device memory, memory associated with `occa::memory` objects must be mapped to an address space in the host. Memory managers assume that no two distinct and proper memory allocations contain an overlap of addresses. Thus, reserving a matching sized address range from the host guarantees the address range received is reserved. When a user queries memory allocation in the device, we reserve a virtual address range of the same size through the use of `mmap` on Unix-based operating systems and `VirtualAlloc` on Windows operating systems. A map is used to contain the starts and ends of each virtual address resulting from `device::malloc`, `device::textureAlloc`, and `device::managedMalloc` calls and map them to their respective `occa::memory` objects. With the implementations discussed, we have a method of obtaining `occa::memory` objects corresponding to a host address range.

The next step is to use host memory with automatic device synchronization. Prior to an OCCA kernel launch, data is managed by the *host* and presumed to be correct. When an OCCA kernel is launched, the respective `occa::memory` objects are obtained through the memory map implementation, an *O(log(n))* operation; likewise when memory copies are called through `occa::memcpy`. For each kernel argument in a kernel

launch, the *device* memory is synchronized if it's the kernel argument's first use. Between the *device* memory synchronization and an `device::finish()` function call, the memory is managed by the *device*. During the gap between the device synchronization and ttfdevice::finish(), using the data in the *host* becomes undefined behavior. Code listing 4.2 shows an example of using automatic memory management found in the OCCA API.

```
//---[ addVectors.okl ]-----------------
kernel void addVectors(const int N,
                       const int *a,
                       const int *b,
                       int *ab){

  for(int i = 0; i < N; ++i; tile(16)){
    if(i < N)
      ab[i] = a[i] + b[i];
  }
}

//--------------------------------------

int main(int argc, char **argv){
  occa::device device("mode = CUDA, deviceID = 0");
  occa::kernel addVectors = device.buildKernel("addVectors.okl",
                                               "addVectors");

  int N = 10;

  // Allocate data simulataneously on the device
  //   similar to malloc()
  int *a  = (int*) device.managedUvaMalloc(N * sizeof(int));
  int *b  = (int*) device.managedUvaMalloc(N * sizeof(int));
  int *ab = (int*) device.managedUvaMalloc(N * sizeof(int));

  // Update in the host
  for(int i = 0; i < N; ++i){
    a[i]  = i;
    b[i]  = (1 - i);
    ab[i] = 0;
  }

  // Call a kernel using host data
  addVectors(N, a, b, ab);

  // Synchronize with the device
  device.finish();

  for(int i = 0; i < N; ++i)
    std::cout << "a[" << i << "] = " << a[i] << '\n';

  return 0;
}
```

Listing 4.2: The code listing displays how automatic memory management occurs with the OCCA API.

4.3 Optimizations

Automatic data transfers facilitate managing the offload model provided by the OCCA API but unfortunately can diminish performance drastically. Fortunately, the

custom parser used for OKL and OFL contains features to analyze kernels to prevent extraneous data transfer. Although counter intuitive, manual management for automatic managed data is also available through the OCCA API.

The `addVectors` kernel shown in code listing 4.2 contains four arguments: `const int N`, `const int *a`, `const int *b`, `int *ab`. Omitting `N` since it's passed by value, we can infer `a` and `b` are not modified throughout `addVectors` due to their `const` qualifier. Hence, only the kernel argument represented by `ab` requires a memory transfer back to the *host*. Were the `const` qualifiers missing from the kernel arguments, code analysis on the kernel would provide the same conclusion. Conservative analysis is currently used applied to handle conditional statements. Code listing 4.3 contains a similar kernel found in code listing 4.2 without kernel argument decorations but gives the same conclusion as `addVectors` found in code listing 4.2. Statements inside loop `(1)` are analyzed due to the conditional statement `(i < N)` only known at run-time while loop `(2)` is omitted.

```
kernel void addVectors(const int N,
                       int *a,
                       int *b,
                       int *ab){

  for(int i = 0; i < N; ++i; tile(16)){
    if(i < N) // (1)
      ab[i] = a[i] + b[i];

    if(false) // (2)
      a[i] = 0;
  }
}
```

Listing 4.3: The code listing contains a kernel with conditional statements, some known at compile-time and others only at run-time.

Code analysis, even with a less conservative approach compared to that of the current implementation, will always limit the data transfer efficiency. When data is updated in an OCCA kernel, it can no longer be assumed to be synchronized with the

host and thus requires a data transfer. In the instance where data should be kept in the *device* but `occa::device` synchronization is required (i.e. timing kernels or waiting for kernels), the user should possess the option of intervening with data management. Manual residency of data can be achieved through the use of

```
bool needsSync(void *ptr);
void dontSync(void *ptr);
```

Additionally, the user can use

```
void syncToDevice(void *ptr, const uintptr_t bytes);
void syncFromDevice(void *ptr, const uintptr_t bytes);
```

to specify the memory ranges used in the *device*.

4.4 Concluding Remarks

The work described in this chapter targeted automatic data movement between the *host* and *device*. Using the *host* to initialize and modify data simultaneously used in `occa::kernel`s facilitates development and porting attempts to use OCCA. Future work on the automatic data movement includes less conservative approaches for identifying possible read and writes to prevent extraneous data transfer between the *host* and *device*. For example, implementing constant propagation for identifying unwritten data through function calls (Callahan et al. 1986) and detecting unused data segments as opposed to the assumption the complete data array was modified.

5 OAK: OCCA Automagic Kernel

The contributions described in this chapter introduce an additional layer of automation for the proposed unified programming model. Code patterns in numerical method implementations are analyzed to detect obstructions in code analysis. Through the use of language constructs and run-time capabilities available in OCCA*, auxiliary information can be passed for additional code analysis. Examples of automatic kernel generation from serial codes are presented.*

Chapters 2, 3, and 4 have discussed the OCCA intermediate representation (IR), native-based kernel languages, and automating data movement; each chapter focusing on a set of features, supplying additional layers of code automation to ease developers onto heterogeneous-programming. The OCCA IR unified multiple backend programming languages and standards which would be used by programmers with prior knowledge on GPU programming. With the addition of the OKL and OFL kernel languages, a programmer writes a serial-like code but still requires the knowledge to label parallel loops. Lastly, adding unified virtual addressing and managed memory removed the need to manually transfer data between devices, requiring only device synchronizations. This chapter discusses another level of automation for automating parallel detection in serial code.

While OCCA is a general purpose approach for facilitating programming current architectures, the current approach taken for automatically detecting parallelism is tailored for numerical applications. I first discuss patterns that emerge from numerical applications and how they can be used to automatically detect parallel loops. Following

the pattern descriptions, I describe common polyhedral optimization methods used in compilers for extracting code information. I implemented and augmented some of the described methods and introduce two additional types of kernels: OAK (OCCA automagic kernel) and OAF (OCCA automagic Fortran). Both OAK and OAF take serial code with the option of additional qualifiers to automatically generate parallel kernels, falling back to guiding users when code does not match discussed patterns. In addition to its primary purpose, the discussed code-analysis tools can benefit the development OKL and OFL kernels. Detections of loop-carried dependencies, or data dependencies inside loops requiring a loop to be executed serially to avoid undefined behavior, can offer guidance on the correctness of OKL and OFL kernels.

5.1 Coding Patterns in Numerical Applications

Promising the automatic transformation of any code is beyond the scope of this project. Rather, the implementations and developed tools tailor common routines found in numerical methods and other HPC-related topics. I analyze three different numerical methods and high performing algorithms used for their implementations.

5.1.1 Finite Difference

The first numerical method whose algorithm patterns are analyzed is the finite difference (FD) method. We take the wave equation, a simple partial differential equation (PDE) used to describe the FD method. Assuming a three dimensional problem, the wave equation is given by

$$\frac{\partial^2 u}{\partial t^2} = c^2 \Delta u \tag{5.1}$$

where u represents pressure or displacement over time t on a domain whose material coefficients are denoted by a spatial-dependent variable c. Analytical derivatives are

approximated by Taylor expansion based integration rules, such as trapezoidal, Simpson's, or higher order rules. Hence, the domain of interest Ω is usually approximated by block-based structured discretization to obtain solution values at integration points efficiently.

We'll generalize the integration method for the discretized derivatives of equation (5.1) to

$$\frac{\partial^n u}{\partial \mathbf{x_i^n}} \approx \sum_{i=0}^{N} w_i u(\vec{x}_i), \;\; \vec{x}_i \in \mathbb{R}^3 \times [0, T) \tag{5.2}$$

Note that the set of parameters $N, w_i, \vec{x}_i$ are chosen for each discretized derivative and thus can differ between x, y, z, t. Rather than creating a matrix representing the discrete operator Δu, the stencils are applied to generate the right-hand side of equation (5.1) for each time step. The pseudocode for marching the FD method in time can be shown in algorithm 1, where w_i is kept the same for all dimensions with a first-order integration scheme in time.

Algorithm 1 Pseudocode for the finite difference method implementation of equation (5.1)

for all $n \in [2, T]$ **do** { For each time step }
  for all points (i, j, k) **do** { For each point in our discrete domain }
    $lap \leftarrow 0$
    for $-r \leqslant \hat{r} \leqslant r$ **do** { For all nodes in the 1D stencil }
      $lap \leftarrow lap + w_o\big($ $c2_{(i+\hat{r},j\quad,k\quad)} u_n(i+\hat{r},\ j\quad,\ k\quad) +$ { Update lap }
      $c2_{(i\quad,j+\hat{r},k\quad)} u_n(i\quad,\ j+\hat{r},\ k\quad) +$
      $c2_{(i\quad,j\quad,k+\hat{r})} u_n(i\quad,\ j\quad,\ k+\hat{r})\big)$
    end for
    $u_{n+1}(i,j,k) \leftarrow \big($ $-\Delta^2 t * lap$ { Store solution at $t = t_{n+1}$}
    $+\ u_{n-1}(i,j,k)$
    $-\ 2u_n(i,j,k)\big)$
  end for
end for

Implementing algorithm 1 efficiently has been thoroughly investigated for CPU and GPU based architectures. Optimizations for the CPU methods include vectoriza-

tion of the stencil operations, cache blocking (or tiling) for spatial data re-use, and time skewing for a combination of spatial and temporal data re-use (Datta et al. 2008, Zhou et al. 2014, Medina et al. 2015). We won't focus on time skewing due to their complexity when additional phases are introduced in a single time step; for example, checkpointing to save data in time or using data transfers in parallel applications. Code listing 5.1 outlines how common optimization techniques would be applied to algorithm 1.

```
// Block points such that each block can be optimally be stored in cache
for(int blockZ = r; blockZ < (pointsZ - r); blockZ += blockOffZ; outer2){
  for(int blockY = r; blockY < (pointsY - r); blockY += blockOffY; outer1){
    for(int blockX = r; blockX < (pointsX - r); blockX += blockOffX; outer0){
      // Points updated in this scope will begin to be
      //   fetched to cache and remain there
      for(int z = blockZ; z < (blockZ + blockOffZ); ++z; inner2){
        for(int y = blockY; y < (blockY + blockOffY); ++y; inner1){
          for(int x = blockX; x < (blockX + blockOffX); ++x; inner0){
            float lap = 0;
            // Vectorization occurs in the inner-most loop
            //   which pertains to the stencil applicaiton
            //   for each dimension
            for(int i = -r; i <= r; ++i)
              lap += w[i]*(u0[z  ][y  ][x+i] +
                           u0[z  ][y+i][x  ] +
                           u0[z+i][y  ][x  ]);

            u2[z][y][x] = -dt2*c2[z][y][x]*lap + u0[z][y][x] + u1[z][y][x];
      }}}
}}}
```

Listing 5.1: Code listing showing the use of vectorization and code blocking for the update step implementation of algorithm 1

Although GPUs are architecturally similar to CPUs, the parallel model and explicit use of cache allows for alternative algorithms to be used. A common GPU implementation for the finite difference method uses coalesced loads for efficient global memory fetches which are stored into shared memory for reuse (Micikevicius 2009, Medina et al. 2015). The domain is partitioned into blocks, similar to cache blocking on the CPU implementation found in code listing 5.1 but differs in the update of each block. The work on each block is partitioned into xy planes, sharing the xy plane data

for each CUDA block while maintaining the z data in each block in a register array. Code listing 5.2 outlines the efficient implementation discussed in (Micikevicius 2009) when applied to algorithm 1.

```
// Block points to partition work for each block/work-group
for(int blockZ = r; blockZ < (pointsZ - r); blockZ += blockOffZ; outer2){
  for(int blockY = r; blockY < (pointsY - r); blockY += blockOffY; outer1){
    for(int blockX = r; blockX < (pointsX - r); blockX += blockOffX; outer0){
      shared    float s_xy[blockOffY + 2*r][blockOffX + 2*r];
      exclusive float r_z[blockOffZ + 2*r];

      // Gather all z values in a register array
      for(int y = 0; y < blockOffY; ++y; inner1){
        for(int x = 0; x < blockOffX; ++x; inner0){
          for(int z = -r; z <= (blockOffZ + r); ++z)
            r_z[z + r] = u0[blockZ + z][y][x];
      }}

      // Update shared memory per z plane
      for(int z = 0; z < blockOffZ; ++z){
        for(int y = 0; y < blockOffY; ++y; inner1){
          for(int x = 0; x < blockOffX; ++x; inner0){
           const int y2 = y + blockOffY;
           const int x2 = x + blockOffX;

           // Store inner square
           s_xy[y+r][x+r] = u0[z][blockY + y][blockX + x];

           // Store top and bottom halo
           if(x < r){
             s_xy[y  + r][x] = u0[z][blockY + y ][blockX + x - r];
             s_xy[y2 + r][x] = u0[z][blockY + y2][blockX + x - r];
           }

           // Store left and right halo
           if(y < r){
             s_xy[y][x  + r] = u0[z][blockY + y - r][blockX + x ];
             s_xy[y][x2 + r] = u0[z][blockY + y - r][blockX + x2];
           }
        }}

        barrier(localMemFence);

        for(int y = 0; y < blockOffY; ++y; inner1){
          for(int x = 0; x < blockOffX; ++x; inner0){
            const int gx = (blockX + x);
            const int gy = (blockY + y);
            const int gz = (blockZ + z);

            float lap = 0;
            // Vectorization occurs in the inner-most loop
            //   which pertains to the stencil applicaiton
            //   for each dimension
            for(int i = 0; i < (2*r + 1); ++i)
              lap += w[i]*(s_xy[y  ][x+i] +
                           s_xy[y+i][x  ] +
                           r_z[z+i]);

          u2[gz][gy][gx] = -dt2*c2[gz][gy][gx]*lap + u0[gz][gy][gx] + u1[gz][gy][gx];
    }}}
}}}
```

Listing 5.2: Code listing showing the use of vectorization and code blocking for the update step implementation of algorithm 1

Both implementations found in code listing 5.1 and code listing 5.2 fit the OKL programming model and hence were described with it. On a naive implementation,

the loops on code listing 5.1.

```
// Block points such that each block can be optimally be stored in cache
for(int blockZ = r; blockZ < (pointsZ - r); blockZ += blockOffZ; outer2){
  for(int blockY = r; blockY < (pointsY - r); blockY += blockOffY; outer1){
    for(int blockX = r; blockX < (pointsX - r); blockX += blockOffX; outer0){
      // Points updated in this scope will begin to be
      //   fetched to cache and remain there
      for(int z = blockZ; z < (blockZ + blockOffZ); ++z; inner2){
        for(int y = blockY; y < (blockY + blockOffY); ++y; inner1){
          for(int x = blockX; x < (blockX + blockOffX); ++x; inner0){
```

would be written as

```
for(int z = r; z < (pointsZ - r); ++z){
  for(int y = r; y < (pointsY - r); ++y){
    for(int x = r; x < (pointsX - r); ++x){
```

If no loop carried dependencies are detected in the naive loops, for-loops can be automatically tagged as OKL `outer` loops and splitting them to produce `inner` loops; note tiling the loops reproduces code listing 5.1. Although it would not reproduce code listing 5.2, the preliminary generated OKL kernels detect no loop-carried dependencies on the `x`, `y`, and `z` loops in code listing 5.3.

```
//---[ Parameters ]-----------------------------
#define pointsX 1000
#define pointsY 1000
#define pointsZ 1000

#define pointsXY (pointsX * pointsY)

#define r 5
//=============================================

kernel void fdUpdate(float *u0,
                     float *u1,
                     float *u2,
                     float *w,
                     float *c2,
                     const float dt2){

  for(int z = r; z < (pointsZ - r); ++z){
    for(int y = r; y < (pointsY - r); ++y){
      for(int x = r; x < (pointsX - r); ++x){
        float lap = 0;

        for(int i = -r; i <= r; ++i)
          lap += w[i]*(u0[(z  )*pointsXY + (y  )*pointsX + (x+i)] +
                       u0[(z  )*pointsXY + (y+i)*pointsX + (x  )] +
                       u0[(z+i)*pointsXY + (y  )*pointsX + (x  )]);

        const int id = (z*pointsXY + y*pointsX + x);

        u2[id] = -dt2*c2[id]*lap + u0[id] + u1[id];
      }
    }
  }
}
```

Listing 5.3: Code listing includes an OAK kernel from a serial finite difference implementation. No loop-carried dependencies are detected in the `x`, `y`, and `z` loops. Note that the parameters would be passed at run-time through the use of `occa::kernelInfo`.

5.1.2 Finite Element and Discontinuous Galerkin Methods

The second and third numerical methods analyzed are the finite element method (FEM) and discontinuous Galerkin method (DG). Both methods act on elements, or cells, approximating the domain of interest but differ when managing the connectivity between elements. We take the same wave equation described in subsection 5.1.1 as a simple partial differential equation (PDE) used to describe the the FEM and DG methods. The domain of interest, Ω, is then discretized to an approximate domain $\Omega_h = \bigcup_k D_k$ decomposable into elements D_k. A mapping between a reference element $\hat{D}$ and each triangular element D_k allows each element to be processed identically, with

some exceptions; for example, boundary conditions or external source terms.

$$\begin{aligned} p_t &= -c^2(u_x + v_y + w_z) \\ u_t &= -p_x \\ v_t &= -p_y \\ w_t &= -p_z \end{aligned} \tag{5.3}$$

Equation (5.3) shows the first-order form of the wave equation in equation (5.1), where p is the pressure field, the vector (u, v, w) denotes the particle velocity components, and c^2 corresponds to the material coefficients in the domain. The matrix formulation of equation (5.3) is given by equation (5.4).

$$q_t = -\nabla \cdot F(q), \tag{5.4}$$

where

$$q = \begin{bmatrix} p \\ u \\ v \\ w \end{bmatrix}, \quad F = \left(\hat{A}_1 e_1 + \hat{A}_2 e_2 + \hat{A}_3 e_3\right).$$

I abuse the notation for $\nabla \cdot F(q)$ to mean

$$\begin{aligned} \nabla \cdot F(q) &= \nabla \cdot \hat{A}_1 q e_1 + \hat{A}_2 q e_2 + \hat{A}_3 q e_3 \\ &= \frac{\partial \hat{A}_1 q}{\partial x} + \frac{\partial \hat{A}_2 q}{\partial y} + \frac{\partial \hat{A}_3 q}{\partial z}, \end{aligned}$$

where

$$\hat{A}_1 = \begin{bmatrix} 0 & c^2 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \; \hat{A}_2 = \begin{bmatrix} 0 & 0 & c^2 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \; \hat{A}_3 = \begin{bmatrix} 0 & 0 & 0 & c^2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix}.$$

Due to computational limitations, the test space is then constrained to the span of a finite number of basis functions belonging to the desired test space presented, say V. The variational formulation of equation (5.4) is given by

$$\underbrace{(q_t, \phi)_{\Omega_h}}_{A} = \underbrace{-(\nabla \cdot F(q), \phi)_{\Omega_h}}_{B} + \underbrace{(G^*(q) \cdot \vec{n}, \phi)_{\Omega_h}}_{C}, \; \forall \phi \in V \tag{5.5}$$

where the numerical flux (C) is present in discontinuous Galerkin methods but vanishes in finite element methods.

There is a large literature on FEM and DG and multiple formulations of each method and because it is over the reach of the thesis to investigate each method, only two methods are detailed. The first method uses nodal basis functions to add another layer of granularity for computing element updates (Hesthaven and Warburton 2007, Klöckner et al. 2009, Gandham et al. 2015, Modave et al. 2015). Specialized methods using nodal basis exist, such as the use of tensor-product basis for efficiency and low-memory costs, yet are usually applicable only on quadrilateral and hexahedral elements (Fischer et al. 2008, Fischer et al. 2007, Giraldo and Rosmond 2003, Fahrenholtz et al. 2015). The examples covered will assume no structural composition of the basis functions. The second method covers implementations represented by building the global matrix, dense or sparse, (Bangerth et al. 2007, Besson and Foerch 1997) used to apply the FEM or DG operators (Rivière 2008). In no way do these methods cover every implementation possible, but they do cover a majority of implementations. The second approach which opts to use a matrix application will not be covered in

this chapter due to it being a thoroughly covered topic (Golub and Van Loan 2012, Williams et al. 2009, Bell and Garland 2008) and hence we focus on the first and second methods which avoid assembling stiffness matrices.

Treating equation (5.4) numerically allows us to represent the operators as matrices as seen in equation (5.6), later applied in a matrix-free fashion (Hesthaven and Warburton 2007).

$$\frac{\partial q}{\partial t} = \underbrace{-M^{-1}\big(S_x A_1 q + S_y A_2 q + S_z A_3 q\big)}_{B} + \underbrace{LIFT\big((G\cdot\vec{n})^* - (G\cdot\vec{n})\big)}_{C} \tag{5.6}$$

$$= \begin{bmatrix} Dr,\ Ds,\ Dt \end{bmatrix} \left(\begin{bmatrix} r_x^e \\ s_x^e \\ t_x^e \end{bmatrix} A_1 q + \begin{bmatrix} r_y^e \\ s_y^e \\ t_y^e \end{bmatrix} A_2 q + \begin{bmatrix} r_z^e \\ s_z^e \\ t_z^e \end{bmatrix} A_3 q \right) \tag{5.7}$$

$$+\ LIFT((G\cdot\vec{n})^* - (G\cdot\vec{n})^-). \tag{5.8}$$

$$\tag{5.9}$$

The Jacobian for the reference mapping is given by

$$\begin{bmatrix} r_x^e & s_x^e & t_x^e \\ r_y^e & s_y^e & t_y^e \\ r_z^e & s_z^e & t_z^e, \end{bmatrix}$$

which are precomputed for each element D_e. For implementation efficiency, we note the matrices Dr, Ds, and Dt are independent of the element and thus can be reused for each element. Methods for numerically computing parts (B) and (C) in equation (5.4) are given in code listing 5.4 and code listing 5.5 respectively.

```
for(int e = 0; e < elements; ++e; outer0){
  shared float s_p[nodesPerElement];
  shared float s_u[nodesPerElement];
  shared float s_v[nodesPerElement];
  shared float s_w[nodesPerElement];
  shared float s_geo[9];

  for(int n = 0; n < nodesPerElement; ++n; inner0){
    if(n < 9)
      s_geo[n] = g_geo[9*e + n];

    int Np = nodesPerElement;

    s_p[n] = g_Q[e*Np*fields + 0*Np + n]; // Fields = 4 in this case
    s_u[n] = g_Q[e*Np*fields + 2*Np + n];
    s_v[n] = g_Q[e*Np*fields + 3*Np + n];
    s_w[n] = g_Q[e*Np*fields + 4*Np + n];
  }

  barrier(localMemFence); // Make sure shared memory is in sync

  for(int n = 0; n < nodesPerElement; ++n; inner0){
    for(int m = 0; m < Np; ++m){
      // Define:
      //    [rx,sx,tx] = s_geo[0,1,2]
      //    [ry,sy,ty] = s_geo[3,4,5]
      //    [rz,sz,tz] = s_geo[6,7,8]

      p_dx = rx*g_Dr[m][n] + sx*g_Ds[m][n] + tx*g_Dt[m][n];
      p_dy = ry*g_Dr[m][n] + sy*g_Ds[m][n] + ty*g_Dt[m][n];
      p_dz = rz*g_Dr[m][n] + sz*g_Ds[m][n] + tz*g_Dt[m][n];

      p_px += p_dx*s_p[m];
      p_py += p_dy*s_p[m];
      p_pz += p_dz*s_p[m];
      p_Du += (p_dx*s_u[m] + p_dy*s_v[m] + p_dz*s_w[m]);
    }

    p_c2 = g_c2[e*Np + n];    // c^2

    Q2[e*Np*fields + 0*Np + n] = -p_c2*p_Du;
    Q2[e*Np*fields + 1*Np + n] = -p_px;
    Q2[e*Np*fields + 2*Np + n] = -p_py;
    Q2[e*Np*fields + 3*Np + n] = -p_pz;
  }
}
```

Listing 5.4: Pseudocode for computing part (B) in equation (5.4), the volume integral computation for updating the solution of equation (5.1)

```
for(int e = 0; e < elements; ++e; outer0){
  shared float s_p[nodesPerFace * facesPerElement];
  shared float s_u[nodesPerFace * facesPerElement];
  shared float s_v[nodesPerFace * facesPerElement];
  shared float s_w[nodesPerFace * facesPerElement];

  const int Nfp    = nodesPerFace;
  const int Nfaces = facesPerElement;

  for(int n = 0; n < nodesPerElement; ++n; inner0){
    if(n < (Nfp*Nfaces)){
      int face = (n / Nfp);

      // Get values for node and normal dot-product
      //   on the (-) side of the element
      int idM      = k*2*Nfp*Nfaces + n;
      float pM     = fQ[idM + 0*Nfp*Nfaces];
      float ndotuM = fQ[idM + 1*Nfp*Nfaces];

      // Get values for node and normal dot-product
      //   on the (+) side of the element
      int idP      = mToP[n + k*Nfp*Nfaces];
      float pP     = fQ[idP + 0*Nfp*Nfaces];
      float ndotuP = fQ[idP + 1*Nfp*Nfaces];

      // Jumps between element nodes
      float dp     = (pP     - pM);
      float ndotdU = (ndotuP - ndotuM);

      // Get geometric factors
      float nx     = g_geo[face + 0*Nfaces + 16*k];
      float ny     = g_geo[face + 1*Nfaces + 16*k];
      float nz     = g_geo[face + 2*Nfaces + 16*k];
      float Fscale = g_geo[face + 3*Nfaces + 16*k];

      float pTmp = -0.5*c*(ndotdU - dp)*Fscale;
      float pTmp = -0.5*  (dp - ndotdU)*Fscale;

      s_p[n] = pTmp;
      s_u[n] = uTmp * nx;
      s_v[n] = uTmp * ny;
      s_w[n] = uTmp * nz;
    }
  }

  barrier(localMemFence); // Make sure shared memory is in sync

  for(int n = 0; n < nodesPerElement; ++n; inner0){
    float p_lift = 0;
    float u_lift = 0;
    float v_lift = 0;
    float w_lift = 0;

    for(int m = 0; m < (Nfp*Nfaces); ++m){
      float lift = LIFT[n + m*nodesPerElement];

      p_lift = LIFT*s_p[m];
      u_lift = LIFT*s_u[m];
      v_lift = LIFT*s_v[m];
      w_lift = LIFT*s_w[m];
    }

    int id = n + k*Nfields*nodesPerElement;

    rhsQ[id + 0*nodesPerElement] = p_lift;
    rhsQ[id + 1*nodesPerElement] = u_lift;
    rhsQ[id + 2*nodesPerElement] = v_lift;
    rhsQ[id + 3*nodesPerElement] = w_lift;
  }
}
```

Listing 5.5: Pseudocode for computing part (C) in equation (5.4), the sufrace integrals along boundary faces for updating the solution of equation (5.1)

```
for(int dof = 0; dof < dofCount; ++dof){
  int dofOffset = dofOffsets[dof];
  int dofCount  = (dofOffsets[dof + 1] - dofOffset);

  float u_sum = 0;

  // Gather the values of each degree of freedom copy
  for(int i = dofOffset; i < (dofOffset + dofCount); ++i)
    u_sum += u[ LToG[i] ];

  // Scatter the real value back to each copy
  for(int i = dofOffset; i < (dofOffset + dofCount); ++i)
    u[ LToG[i] ] = u_sum;
}
```

Listing 5.6: Pseudocode for applying a gather and scatter operation to incorporate and update values of each degree of freedom copy

We note the parallelism found in both methods by providing the OKL loop tags in code listing 5.4 and code listing 5.5. The volume integrals for each element can be updated in parallel but require an additional processing step to include element connectivity. In DG implementations, the weak connectivity across face nodes is handled with the surface integrals consisting of the flux across elements. However, FEM implementations that contain the same memory structure as that of DG retain copies of a nodal value for each element containing it; hence, each copy is gathered to obtain the global solution and scattered back to update each copy as seen in code listing 5.6. Finding loop-carried dependencies in code listing 5.6 would not be possible without additional knowledge about the `LToG` mapping. Subsection 5.2.3 discusses attributes used to bypass the claimed compiler-issues and avoid false positives in the loop-carried dependency analysis.

5.2 Automagic Analysis

Using the compiler, not just for machine-code generation, but as an optimization tool for code has been prevalent since before the 1980's (Allen and Cocke 1972, Padua and Wolfe 1986, Allen and Johnson 1988). As hardware evolved, such as with the intro-

duction of vectorization instructions and multicore processors, compiler optimizations adapted. I'll describe some basic compiler tools used to detect loop-carried dependencies tailored for the methods discussed in section 5.1. Assumptions on the tailored analysis are more conservative and demanding for these preliminary pattern detections when compared to libraries focused on polyhedral optimization and integer-set constraint detections (Grosser et al. 2011, Benabderrahmane et al. 2010). The notion of using language constructs to portray additional information about the program is explored for the analysis tools. The additional constructs and their capabilities are further explained throughout this section, targeting code patterns described in section 5.1. To experiment these concepts, I implemented an analyzer for detecting parallelism in serial code with features discussed throughout this section.

5.2.1 Value Extractions

For-loops, or DO loops in Fortran, are the main targets for code analysis due to the objective being guiding users towards writing loops without loop-carried dependencies and auto-generating the resulting kernels. Although other libraries only require the loop bounds and strides to be constant or affine at compile-time, I target constant-valued loop bounds and strides. Making use of the run-time compilation capabilities in the OCCA library, strides and bound expressions in targeted for-loops can be given during compile-time using the OCCA API. I now discuss the extent of the analyzer developed for OAK and OAF.

The analyzer stores information about each variable, accesses, and if known, it's current value. For example, variables used as for-loop iterators are tagged as iterators and their bounds and strides are kept. Values are symbolically simplified by expanding expressions, an example shown in code listing 5.7

```
   1 + 2*x + 3*(x + 4*y)
↳ 1 + 2*x + 3*x + 3*4*y
↳ 1 + (2 + 3)*x + 12*y
↳ 1 + 5*x + 12*y
```

Listing 5.7: Example showing symbolic simplification of an expression

Values in the analyzer are stored as either constant values or stride expressions. If the symbolic simplification of the expression is constant at compile-time, the constant value is stored. If the simplified expression is not constant, the expression is split by their summed parts as seen in code listing 5.8.

```
   1 + 2*x + 3*(x + 4*y) + x*y
↳ 1 + 5*x + 12*y + x*y
↳ [1(1), x(5), y(12), x*y]
```

Listing 5.8: Example showing the symbolic simplification split by its summed parts

Code listing 5.8 shows an offset of `1` with iterators (`x`, `y`) and their respective strides (`5`, `12`). Note that the expression `x*y`, because not a linear combination of variables or iterators, is labeled as complex and thus not given a stride. This same processes of evaluating expression information is done for loop-bounds. For example, code listing 5.9 displays a for-loop and it's respective expression information.

```
for(int i = 0; i < 10; ++i){}         // (1)
↳ [Bounds: [0, 10), Stride: [ 1]]

for(int i = A; 0 <= i; --i){}         // (2)
↳ [Bounds: [0, A ], Stride: [-1]]

for(int i = A; i < B; ++i){}          // (3)
↳ [Bounds: [A, B ), Stride: [ 1]]

// Dead code, not analyzed               (4)
for(int i = 0; i < 0; ++i){}
```

Listing 5.9: Loop information extracted for different cases

Example `(1)` in code listing 5.9 contains loop-bounds known at compile-time and

can be extensively used for loop-carried dependency analysis. Example `(2)` has it's lower-bound and stride known at compile-time while it's upper-bound and example `(3)` bounds are limited by their symbolic expressions. Example `(4)` is an example where the analyzer ignores code segments which will never get executed. This leads to the following subsection where there loop-carried dependency analysis is described.

5.2.2 Detection of Loop Carried Dependencies

Variable information gathered through value extractions discussed in subsection 5.2.1 are used to determine the existence of loop-carried dependencies in code. Reads and writes of each variable is recorded sequentially in addition to the accesses for all pointer variables. Two forms of loop-carried dependencies are inspected, reductions and access conflicts.

When a variable is updated relative to its own value, for example with operators `+=`, `-=`, `*=`, `/=`, a dependence is formed with itself. For example, code listing 5.10 shows the variable `sum` used to store a reduction of the iterator `i`.

```
int sum = 0;

for(int i = 0; i < 10; ++i)
  sum += i;
```

Listing 5.10: The variable `sum` stores the reduction of `i`, thus causing a loop-carried dependency on the `i` for-loop.

The scope of the loop-carried dependency can propagate across multiple statements, not just on the loop containing the reduction variable. A statement containing a dependency on a reduction variable will cause the loop-carried dependency to propagate across. To be specific, the least common ancestor of the statement containing the reduction variable and statements depending on the reduction variable must be serialized to prevent undefined behavior. The code listing 5.11 contains an example with a

reduction variable `sum` which is then stored in a subsequent for-loop, hence requiring the `j` for-loop to be run serially.

```
for(int j = 0; j < 10; ++j){
  int sum = 0;

  for(int i = 0; i < 10; ++i)
    sum += i;

  for(int i = 0; i < 10; ++i)
    sumArray[i] = sum;
}
```

Listing 5.11: The loop-carried dependence formed by the `sum` reduction is propagated to the `j` for-loop due to its use in updating `sumArray`.

The second cause of loop-carried dependencies is caused by read-write accesses that would cause undefined behavior if executed out-of-order. Code listing 5.12 contains two loops with different access patterns, the first containing a loop-carried dependence and the second can be executed in parallel due to no read-write conflicts.

```
// Reads    : [1, 2, ..., 10]
// Writes   : [0, 1, ...,  9]
// Conflicts: [1, 2, ...,  9]
for(int i = 0; i < 10; ++i)
  A[i] = A[i + 1];

// Reads : [0, 2, ..., 8]
// Writes: [1, 3, ..., 9]
// No conflicts
for(int i = 0; i < 10; i += 2)
  B[i] = B[i + 1];
```

Listing 5.12: In this code listing, the accesses in the first loop form a loop-carried dependence while the second loop avoids it due to reads in odd entries and writes in even entries.

The analysis is not done for each possible read and write, but rather a conservative approach for detecting duplicate solutions to the Diophantine equations is used. Assuming two accesses,

$$A\left[\sum_{i=0}^{N}\alpha_i s_i^1\right], A\left[\sum_{i=0}^{M}\beta_i s_i^2\right],$$

with coefficients α_i, β_i and strides s_i^1, s_i^2 are given, an access conflict exists if and only if a solution to the Diophantine equation

$$\sum_{i=0}^{N} \alpha_i s_i^1 = \sum_{i=0}^{M} \beta_i s_i^2,$$

exists within the bounds given by the coefficients and strides. Instead of detecting for multiple solutions, bounds for each stride and coefficient are computed to expose overlapping bounds. I will use the following notation for simplification To compress the notation, I use the following simplification

`[Bounds: [0, 10), Stride: [1]]` → `[0,10)(1)`

```
for(int i = 0; i < 10; ++i){
// ↳ i: [Bounds: [0, 10), Stride: [1]]

  // Strides consist of i and 1
  A[i] = A[i + 1];
  // ↳ A (Read) : [1(i) + 0(1)]
  // ↳          : [[0, 10)(1) + [0,0](1)]

  // ↳ A (Write): [1(i) + 1(1)]
  // ↳          : [[0, 10)(1) + [1,1](1)]
}

for(int i = 0; i < 10; i += 2){
// ↳ i: [Bounds: [0, 10), Stride: [2]]

  // Strides consist of i and 1
  B[i] = B[i + 1];
  // ↳ B (Read) : [1(i) + 1(1)]
  // ↳          : [[0, 10)(2) + [1,1](1)]

  // ↳ B (Write): [1(i) + 0(1)]
  // ↳          : [[0, 10](2) + [0,0](1)]
}
```

Listing 5.13: The analysis tools generate the following detections based on code listing 5.12.

Code listing 5.13 displays the metadata obtained from code listing 5.12. The read and write statements in the first loop in would generate race conditions due to the

conflict between the offset `1` and range `[0, 10)(1)`. Although similar, the second loop wouldn't generate a conflict due to the range having a stride of `2`. Assume the following two accesses are present

$$A\left[\sum_i [LB_i^1, UB_i^1)(S_i^1)\right], A\left[\sum_i [LB_i^2, UB_i^2)(S_i^2)\right],$$

where LB, UB, and S correspond to the lower bound, upper bound, and strides of the access. No conflict exists if

$$UB_i^1 \leqslant (LB_j^2 + S_j^2) \quad \text{or} \quad UB_j^2 \leqslant (LB_i^1 + S_i^1), \ \forall i, j,$$

otherwise a conservative approach is taken which will possibly detect a false positive. Similar to reduction variables, if conflicts are discovered, the loop-carried dependency is propagated to the least common ancestor of the two access statements.

Because simple affine access patterns are currently being targeted, the discussed conservative approach is sufficient for the analyzed serial codes. However, future work includes the implementation of methods which can analyze access conflicts more accurately. For example, possessing the stride and bound information can lead to a more flexible approach towards detecting access conflicts by solving the Diophantine equations (Banerjee et al. 1979).

5.2.3 Additional Language Constructs

As previously mentioned, the analysis focuses on language constructs to provide additional code information at compile-time. Rather than using `#pragma`'s for decorating statements, I opted for the use of attributes for a more compact form of decorating code. The use of decorations gives developers another layer of expressing programming motives to facilitate compiler optimizations (Lattner and Adve 2004, Necula et al.

2002, Pominville et al. 2000); however, using attributes does not follow the language specifications similar to `#pragma` to concise the decoration.

I added the use of the `@` symbol to the OKL and OFL specifications for a preliminary implementation of attributes, shorter than the use of `__attribute__(())` seen in GCC and LLVM based compilers. The use of the attribute symbol can be applied to give additional information to the compiler about variables and their use or statements in general. Code listing 5.14 shows different forms of using `@` with variable declarations, variable update statements, and loop statements.

```
void kernelName(int *A,
                int *k @(permutation)){
  for(int i = 0; i < 10; ++i) @(safe) {
    A[ k[i] ] = i @(safe);
  }
}
```

Listing 5.14: The use of `@` attribute symbol is shown through the its use in a variable declaration, variable update statement, and a loop statement

Currently, only the `safe` and `permutation` attributes are supported. A statement decorated with `@(safe)` is ignored when checking loop-carried dependencies. Variables decorated with `@(permutation)` is used to indicate a one-to-one mappings which would be impossible for the compiler to validate at compile-time unless the whole array is known. Using the `@(permutation)` attribute would indicate loop-carried dependencies that were falsely found on the gather and scatter in . However, the gather and scatter jointly causes a read-write conflict detection; hence, using the attribute `@(safe)` would be used to indicate the lack of read-write conflicts and kernels would be automatically generated.

With the prototype implementation in effect, future types would be added to augment OKL and OFL kernels and the depth of the OAK and OAF analyzer. For example, adding a `@(dim(16,16))` or `@(dim(X,Y))` could allow bounds to be known

at compile-time by value or relatively to other variables. Notation could be abused for simpler development, for example

```
void kernelName(int *A,
                int *k @(permutation, dim(16,16))){
  for(int j = 0; j < 16; ++j) @(safe) {
    for(int i = 0; i < 16; ++i) @(safe)
      A[ k(j,i) ] = i @(safe);
  }
}
```

Listing 5.15: The code listing shows a predicted use of `@(dim(...))` simplifying kernel development

5.3 Auto-generation of Kernels

The detection tools and language constructs previously mentioned are used to detect loop candidates to be tagged as `outer` or `inner` loops. We refer back to the specifications detailed in subsection 3.2.1 for the requirements on outer and inner loops. To summarize, outer or inner loops must contain no loop-carried dependencies. In addition, outer loops must only contain variable declarations, constant variable definitions, and statements embedded with inner loop candidates with the same bounds.

I will give an example showing the use of OAK kernels from serial code. Given the following serial code,

```
for(int i = 0; i < N; ++i)
  A[i] = B[i] + C[i];
```

Listing 5.16: An example serial code summing two arrays

the user would pack the serial code into an OAK kernel as such

```
kernel void addVectors(int N,
                       float * restrict A,
                       float * restrict B,
                       float * restrict C){

  for(int i = 0; i < N; ++i)
    A[i] = B[i] + C[i];
}
```

Listing 5.17: The kernel displayed shows code listing 5.16 wrapped in an OAK kernel

Note the use of `restrict`, a keyword used to denote pointers are not aliased and can hence be treated as non-overlapping with other pointers. Running the OAK parser on the `addVectors` kernel would generate the following OKL code which is composed of OCCA API, OCCA IR, and OKL.

```
occaKernel void addVectors(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
   occa::setupMagicFor(A);
   occa::setupMagicFor(B);
   occa::setupMagicFor(C);
   addVectors0(N, A, B, C);
   occa::syncToDevice(A);
   occa::syncToDevice(B);
   occa::syncToDevice(C);
   addVectors1(N, A, B, C);
   occa::syncToDevice(A);
   occa::syncToDevice(B);
   occa::syncToDevice(C);
   addVectors2(N, A, B, C);
   occa::syncToDevice(A);
   occa::syncToDevice(B);
   occa::syncToDevice(C);
   addVectors3(N, A, B, C);
   occa::syncToDevice(A);
   occa::syncToDevice(B);
   occa::syncToDevice(C);
   addVectors4(N, A, B, C);
   occa::syncToDevice(A);
   occa::syncToDevice(B);
   occa::syncToDevice(C);
   addVectors5(N, A, B, C);
 }
```

Listing 5.18: The kernel displayed will launch a variety of kernels based on code listing 5.17

```
occaKernel void addVectors0(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
  for(int i = 0; i < N; ++i; tile(8)) {
    A[i] = B[i] + C[i];
  }
}

occaKernel void addVectors1(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
  for(int i = 0; i < N; ++i; tile(16)) {
    A[i] = B[i] + C[i];
  }
}

occaKernel void addVectors2(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
  for(int i = 0; i < N; ++i; tile(32)) {
    A[i] = B[i] + C[i];
  }
}

occaKernel void addVectors3(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
  for(int i = 0; i < N; ++i; tile(64)) {
    A[i] = B[i] + C[i];
  }
}

occaKernel void addVectors4(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
  for(int i = 0; i < N; ++i; tile(128)) {
    A[i] = B[i] + C[i];
  }
}

occaKernel void addVectors5(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
  for(int i = 0; i < N; ++i; tile(256)) {
    A[i] = B[i] + C[i];
  }
}
```

Listing 5.19: The kernels displayed are automatically generated based on code listing 5.17

If nested loops existed, however, additional kernels would be generated. For example

```
for(int i = 0; i < 16; ++i)
  for(int j = 0; j < 16; ++j)
    A[i*16 + j] = 0;
```

would generate not only tiled kernels, but auto-detect the available outer/inner loop

combination as well as experiment with tiling on each feasible loop.

```
// ...

occaKernel void addVectors5(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
    for(int i = 0; i < 16; ++i; tile(256)) {
      for(int j = 0; j < 16; ++j) {
        A[i * 16 + j] = 0;
      }
    }
}

// ...

occaKernel void addVectors6(int N, float * occaRestrict A, float * occaRestrict B,
    float * occaRestrict C) {
  for(int i = 0; i < 16; ++i; outer0) {
    for(int j = 0; j < 16; ++j; inner0) {
      A[i * 16 + j] = 0;
    }
  }
}

// ...

 occaKernel void addVectors12(int N, float * occaRestrict A, float * occaRestrict B,
     float * occaRestrict C) {
  for(int i = 0; i < 16; ++i) {
    for(int j = 0; j < 16; ++j; tile(256)) {
      A[i * 16 + j] = 0;
    }
  }
}
```

Note that the bounds were explicitly stated due to the assumptions taken in OAK kernels, as mentioned in subsection 5.2.3. In addition to the kernel generations, errors are reported when conflicts in accesses are detected. If `j` iterated between 0 and 32, the error

```
Access strides overlap:  [VI: i (C: 16) + VI: j (C: 1)]
```

would be presented and kernels would fail to be generated. To show the functionality of these tools, I present some serial code from the Rodinia benchmarks (code listing 5.20) and a few of its auto-generated OKL kernels (code listing 5.21).

```
//---[ Parameters ]-----------------------------
#define ETA      0.1
#define MOMENTUM 0.2

#define n1 100
#define n2 1000
//============================================

kernel void bpnn_layerforward(float *l1,
                              float *l2,
                              float *conn //,
                              // int n1,
                              // int n2
                              ){
  float sum;
  int j, k;

  l1[0] = 1.0;

  for (j = 1; j <= n2; j++) {
    sum = 0.0;

    for (k = 0; k <= n1; k++)
      sum += conn[k*n2 + j] * l1[k];

    l2[j] = squash(sum);
  }
}

#define ndelta 1000
#define nly    10

kernel void bpnn_adjust_weights(float *delta,
                                // float ndelta,
                                float *ly,
                                // float nly,
                                float *w,
                                float *oldw){
  float new_dw;
  int k, j;

  const int kOff = (ndelta + 1);

  ly[0] = 1.0;

  for (j = 1; j <= ndelta; j++) {
    for (k = 0; k <= nly; k++) {
      new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k*kOff + j]));

      w[k*kOff + j]   += new_dw;
      oldw[k*kOff + j] = new_dw;
    }
  }
}
```

Listing 5.20: The back propagation kernels from the Rodinia benchmark suite were taken and wrapped in OAK kernels. Modifications on the source-code included the explicit bounds passed as defines (normally `kernelInfo::addDefine` would be used in the application)

```
// ...
occaKernel void bpnn_layerforward5(float *l1, float *l2, float *conn, int n1, int n2) {
  float sum;
  int j, k;
  l1[0] = 1.0;
  for(j = 1; j <= n2; j++; tile(256)) {
    sum = 0.0;
    for(k = 0; k <= n1; k++) {
      sum += conn[k * n2 + j] * l1[k];
    }
    l2[j] = squash(sum);
  }
}

occaKernel void bpnn_layerforward6(float *l1, float *l2, float *conn, int n1, int n2) {
  float sum;
  int j, k;
  l1[0] = 1.0;
  for(j = 1; j <= n2; j++; outer0) {
    sum = 0.0;
    for(k = 0; k <= n1; k++; inner0) {
      sum += conn[k * n2 + j] * l1[k];
    }
    l2[j] = squash(sum);
  }
}

// ...

occaKernel void bpnn_adjust_weights5(float *delta, float *ly, float *w, float *oldw) {
  float new_dw;
  int k, j;
  occaConst int kOff = (1000 + 1);
  ly[0] = 1.0;
  for(j = 1; j <= 1000; j++; tile(256)) {
    for(k = 0; k <= 10; k++) {
      new_dw = ((0.1 * delta[j] * ly[k]) + (0.2 * oldw[k * kOff + j]));
      w[k * kOff + j] += new_dw;
      oldw[k * kOff + j] = new_dw;
    }
  }
}

occaKernel void bpnn_adjust_weights6(float *delta, float *ly, float *w, float *oldw) {
  float new_dw;
  int k, j;
  occaConst int kOff = (1000 + 1);
  ly[0] = 1.0;
  for(j = 1; j <= 1000; j++; outer0) {
    for(k = 0; k <= 10; k++; inner0) {
      new_dw = ((0.1 * delta[j] * ly[k]) + (0.2 * oldw[k * kOff + j]));
      w[k * kOff + j] += new_dw;
      oldw[k * kOff + j] = new_dw;
    }
  }
}

// ...
```

Listing 5.21: This code listing contains a few auto-generated kernels resulting from the Rodinia benchmark back propagation kernels in code listing 5.20.

5.4 Concluding Remarks

I presented methods for utilizing the OCCA run-time tools and analyzing serial code to attempt generation proper OKL kernels. By detecting loop-carried dependencies,

we can guide developers to convert serial code to fit the OCCA programming model. After recognizing at least one loop without loop-carried dependencies, kernels can be generated to test performance and provide further guidance on proper loop-labeling. Future work for the OAK and OAF kernels include examining additional polyhedral optimization and integer set tools. Analysis methods are comprised of code movement of loop-invariant statements, inlining non-recursive functions, and additional tests for detecting access conflicts.

6 Implementation Studies and Benchmarks

A careful set of validations are performed to support claims stated throughout this thesis proposal. For example, I choose to compare the performance obtained using OCCA *mediated kernels with the native language counterpart, such as OpenMP, OpenCL, and CUDA. Likewise, the chosen benchmarks not only for examine performance comparisons with native counterparts, but examine portable performance across backends using the kernel language portability.*

In this chapter, I briefly cover a careful set of benchmarks that will be used to validate performance and portability when using OCCA. It is also of interest to discuss performance differences alongside portability challenges, specially for HPC-tailored numerical applications. Two applications from Argonne National Laboratory (ANL) are being developed for simulating neutron transport: XSBench and SimpleMOC. Other affiliated applications include gNuma, a mini-app of the non-hydrostatic unified atmospheric model (NUMA) from the naval postgraduate school and group applications using finite element, discontinuous Galerkin, and spectral element methods. When available, examples compare OCCA IR or OKL with the backend's native language counterpart such as OpenMP, OpenCL, and CUDA.

6.1 Finite Difference Method

In a collaboration with Dr. Amik St-Cyr, I investigated the capability of the OCCA IR on finite difference method implementations. One major goal of this collaboration

was to test the performance obtained with OCCA IR compared to an already optimized application developed at Shell. I summarize the problem and performance comparisons outlined in (Medina et al. 2015).

Because finite difference is computationally efficient on modern architectures, it is a commonly used method in seismic imaging. For this application, a mini-application was developed using OCCA with OCCA IR kernels to implement a high-order wave propagator for the model using vertical transversely isotropic (VTI) media (Du et al. 2008). The model is given by

$$\begin{aligned} p_{tt} &= c^2\big((1+2\epsilon)(p_{xx}+p_{yy}) + \sqrt{1+2\delta}(q_z z)\big), \\ q_{tt} &= c^2\big(\sqrt{1+2\delta}(p_{xx}+p_{yy}) + (q_z z)\big) \end{aligned} \tag{6.1}$$

where p and q are the horizontal and vertical stress components, c is the acoustic velocity of the material, and ϵ and δ are the Thomsen anisotropy parameters (Thomsen 1986). A first order time-stepping method is applied to equation (6.1) and a stencil of radius r is used for the spatial derivative discretization giving

$$\begin{aligned} p^{n+1}(x_i,y_j,z_k) =& 2p^n(x_i,y_j,z_k) - p^{n-1}(x_i,y_j,z_k) \\ &- dt^2c^2\sum_{s=-r}^{r}(1+2\epsilon)\left(w_x^k p^n(x_{i+s},y_i,z_k) + w_y^k p^n(x_i,y_{j+s},z_k)\right) \\ &- dt^2c^2\sum_{s=-r}^{r}\sqrt{1+2\delta}(w_z^k q^n(x_i,y_j,z_{k+s})), \\ q^{n+1}(x_i,y_j,z_k) =& 2q^n(x_i,y_j,z_k) - q^{n-1}(x_i,y_j,z_k) \\ &- dt^2c^2\sum_{s=-r}^{r}\sqrt{1+2\delta}\left(w_x^k p^n(x_{i+s},y_i,z_k) + w_y^k p^n(x_i,y_{j+s},z_k)\right) \\ &- dt^2c^2\sum_{s=-r}^{r} w_z^k q^n(x_i,y_j,z_{k+s}) \end{aligned} \tag{6.2}$$

where w_*^k are the finite difference stencil weights. Two algorithms for implementing

equation (6.2) tailored for the CPU and GPU architectures respectively were implemented, both detailed in code listing 5.1 and code listing 5.2.

Performance results for the finite difference kernels were taken on a dual-socket node equipped with Intel Xeon E5-2640 processors for comparing OpenMP and the OCCA OpenMP-mode. GPU timings were also obtained on NVIDIA K10 and K20x GPUs comparing CUDA and the OCCA CUDA-mode. Two OCCA IR kernels each tailored for the CPU or GPU architecture were developed for fair comparisons, but both are also compared to each other as an example where performance portability is lacking.

Project	Distribution	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
Native	Compact	92	183 (98%)	360 (96%)	668 (89%)	1226 (82%)
Native	Scatter	92	183 (98%)	356 (95%)	686 (92%)	1191 (80%)
OCCA	Compact	115	229 (99%)	448 (97%)	820 (89%)	1548 (84%)
OCCA	Scatter	115	230 (100%)	454 (98%)	884 (96%)	1411 (76%)

Table 6.1: Displayed are the nodes updated per second (in millions) together with multithreading scaling using alternative thread distributions on two Xeon E5-2640 Processors (higher is better).

Table 6.1 contains results between the original application using native OpenMP and the OCCA IR kernels using OpenMP-mode, both with scatter and compact thread pinning arrangements. Timings are displayed in million of node updates per second, a common way of showing performance in finite difference applications, together with the scaling factors between thread counts. Note that the OCCA IR kernels resulted in better performance compared with the native OpenMP kernels. The option of injecting run-time information when building the OCCA IR kernels, such as the finite difference stencil-size or manually unrolling the finite difference stencil updates, allowed for more compiler optimizations. As expected, when substituted, running the OpenMP translations of the OCCA IR kernels resulted in identical performance on the original application.

Project	Kernel Language	K10 (1-chip)	K20x
Native	CUDA	1068	1440
Native (2)	CUDA	1296	2123
OCCA	OCCA:CUDA	1241	1934
OCCA (2)	OCCA:CUDA	1579	2431
OCCA	OCCA:OpenCL	1303	1954
OCCA (2)	OCCA:OpenCL	1505	2525

Table 6.2: Displayed are the nodes updated per second (in millions) for four different kernels, two CUDA kernels and two OCCA IR kernels run on CUDA and OpenCL modes (higher is better). Update kernels use 1-point updates per work-item/thread or are labeled with (2) to represent 2-point update kernels. Two NVIDIA GPUs were used, a single processor of a dual-processor K10 GPU and a K20x GPU.

Table 6.2 contains results between the original application using native CUDA and the OCCA IR kernels using the CUDA and OpenCL modes on NVIDIA K10 and K20x GPUs. Each kernel was modified to update two nodes per GPU work-item/thread and included in the performance timings, also displayed in million of node updates per second. Note that the OCCA IR kernels in both, CUDA and OpenCL modes also resulted in better performance compared with the native CUDA kernels.

	CPU-tailored Kernel	GPU-tailored Kernel
OCCA::OpenMP	1548	364 (23%)
OCCA::CUDA (1 K10 core)	515 (41%)	1241
OCCA::OpenCL (1 K10 core)	665 (51%)	1302

Table 6.3: Displayed are performance comparisons between the GPU and CPU tailored code running on the OpenMP, CUDA and OpenCL modes to examine performance portability.

We examining table 6.3 by joining table 6.1 and table 6.2 timings and compare performance portability. At best, OpenCL running on the CPU-tailored OCCA IR kernel ran at 51% efficiency compared to running on the GPU-based kernel. At worst, OpenMP running on the GPU-tailored OCCA IR kernel ran at 23% efficiency compared to running on the CPU-based kernel. The GPU-tailored kernel used shared-memory to store spatially-local but not memory-local data in cache memory, while the CPU-mode only emulates it. Likewise, the CPU-tailored kernel makes the assumption local

data is automatically loaded into cache which is not as mature on GPU architectures. These implementation features, when heavily used, can cause a lack of performance portability.

6.2 Monte Carlo

In addition to the collaboration with Shell, I collaborated with Ron Rahaman and Amanda Lund from Argonne National Laboratory in Monte Carlo mini-apps. The discussed mini-app is XSBench, currently being developed by the Center for the Exascale Simulation of Advanced Reactors (CESAR) group at the ANL (Tramm et al. 2014). The project is based on OpenMC, an application which models neutron transport at a macroscopic scale using Monte Carlo (Romano and Forget 2013). Rather than optimizing the whole application, XSBench extracts the computationally intensive tasks from OpenMC for focusing on the application bottlenecks. Computations needed for the Monte Carlo calculations consist of around 85% of total runtime, making XSBench a lightweight single-kernel mini-app for testing purposes. The developers have looked at different approaches which address performance and portability for the heterogeneous programming model, namely through the use of OpenMP, OpenACC, CUDA and OKL (Rahaman et al. 2015).

To compare the original XSBench kernels and the OKL ports, we display the number of lookups calculated per second where a lookup represents a single Monte Carlo simulation. Two OKL kernels were implemented, the first tailored for the CPU architecture and the second for the GPU architecture. Table 6.16 compares the performance results between the native XSBench implementation and the OKL ports using two 8-core Intel Xeon E5-2650 processors. The OKL implementation showed comparable performance with its OpenMP counterpart and displayed better thread scaling. For this application, the GPU-based kernel outperformed and scaled better than the CPU-

based kernel in addition to the native OpenMP implementation. The emphasis on the parallel model used in the GPU architecture exploited the thread and vectorization parallel hierarchy seen in the multicore CPUs.

Implementation	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
Native	0.20	0.37 (90%)	0.73 (89%)	1.33 (81%)	2.38 (72%)
OKL (CPU-based)	0.19	0.40 (103%)	0.60 (78%)	1.24 (81%)	2.37 (78%)
OKL (GPU-based)	0.18	0.37 (103%)	0.72 (102%)	1.41 (99%)	2.78 (97%)

Table 6.4: The table displays performance and scaling between the native XSBench OpenMP kernel, and two OKL port running in OpenMP-mode, each tailored for the CPU and GPU respectively. Performance is presented as million lookups per second, where a lookup represents the number of times a neutron changes energy or crosses a material boundary per second. Tests were run to compute 1.5 million lookups on two 8-core Intel Xeon CPU E5-2650 processors.

A similar comparison between the original CUDA implementation and the OKL::CUDA implementation is given in table 6.5. Timings were run on an NVIDIA K40m GPU and tested occupancy through a varying number of threads per block. When compared with the native CUDA kernel, the GPU-based OKL kernel outperformed it by 8% while the CPU-based OKL kernel matched 98% of its performance.

Implementation	32 threads	64 threads	96 threads	128 threads
Native	3.31	3.27	3.14	3.21
OKL (CPU-based)	3.25	2.00	1.69	1.90
OKL (GPU-based)	3.52	2.77	2.85	2.65

Table 6.5: The table displays performance between the original XSBench CUDA kernel, and two OKL port running in CUDA-mode, each tailored for the CPU and GPU respectively. Performance is presented as million lookups per second, where a lookup represents the number of times a neutron changes energy or crosses a material boundary per second. Tests were run to compute 1.5 million lookups on different number of threads per block with an NVIDIA K40m graphics card.

Because OKL kernels are portable across supported platforms, we are able to display the performance of OpenMP on the CUDA-tailored kernel and vice-versa intable 6.6. This is an example where performance is portable across the CPU and GPU architectures, showing optimal performance using the GPU-tailored kernel. Using an

OKL kernel based on the native OpenMP code is still comparable to the GPU-tailored kernel, the major difference arises from the efficiency in scaling.

	CPU-tailored Kernel	GPU-tailored Kernel
OKL::OpenMP (16 threads)	2.37	2.78 (117%)
OKL::CUDA (32 threads)	3.25 (92%)	3.52

Table 6.6: The table displays portable performance available with OCCA. OpenMP achieved an increase of 17% overall performance using the GPU-tailored OKL kernel when compared to the CPU-tailored OKL kernel. The CPU-tailored kernel on CUDA achieved 92% overall performance when compared to the GPU-tailored kernel, showing portable performance even across the different targeted architectures. Tests were run on two 8-core Intel Xeon CPU E5-2650 processors and an NVIDIA K40m graphics card.

6.3 Finite Element Method

The third project examined is GNEK, a spectral element method (SEM) solver for the incompressible Navier Stokes equations seen in equation (6.3) (Stilwell 2013). GNEK uses a spectral element method with tensor product basis which was derived from the open source project Nek5000 led by Paul Fischer at Argonne National Laboratory (Fischer et al. 2008, Fischer and RÃČÂÿnquist 1994, Fischer and Patera 1991). The mini-app was ported to use OCCA from the core operations found in the Nek5000 computational fluid dynamics (CFD) solver.

$$u_t + (u \cdot \nabla)u = -\nabla p + \nu \Delta u \tag{6.3}$$

$$\nabla \cdot u = 0 \tag{6.4}$$

A splitting scheme found in (Karniadakis et al. 1991) is used to treat the nonlinear

term found in Equation (6.3) explicitly, giving

$$\begin{aligned} \tfrac{1}{\Delta t}\left(\tilde{u} - u^n\right) &= -(u^n \cdot \nabla)u^n, \\ \tfrac{1}{\Delta t}\left(\tilde{\tilde{u}} - \tilde{u}\right) &= -\nabla p^{n+1}, \\ \tfrac{1}{\Delta t}\left(u^{n+1} - \tilde{\tilde{u}}\right) &= v\Delta u^{n+1}. \end{aligned} \tag{6.5}$$

The second and third equations in equation (6.5), however, can be generalized as

$$-\Delta u + \lambda^2 u = f, \tag{6.6}$$

with the variational formulation

$$(\nabla u, \nabla \phi)_\Omega + \lambda^2 (u, \phi)_\Omega - (\nabla u \cdot n, \phi)_{\partial\Omega} = (f, \phi)_\Omega \quad \forall v \in V, \tag{6.7}$$

Although a brief description of the finite element method can be found in subsection 5.2.3, we leverage the tensor product structure of the basis functions to more efficiently apply the discrete SEM operator found in equation (6.7). The pseudocode for the operator application is given in code listing 6.1 and I refer the reader to (Medina 2014) for a more in-depth explanation of the method.

```
for(int e = 0; e < elementCount; ++e; outer0){
  shared    float LD[N][N];
  shared    float Lu[N][N], Lv[N][N], Lw[N][N];
  exclusive float uk[N];
  exclusive float lapu[N];

  exclusive float ur, us, ut;
  exclusive float GDut, lapuk;

  for(int j = 0; j < N; ++j; inner1){
    for(int i = 0; i < N; ++i; inner0){
      // Load derivative matrix onto shared memory
      LD[i][j] = g_D[N*j + i];

      // Load the z direction values onto a register array
      for(int k = 0; k < N; ++k){
        uk[k]   = u[e*N3 + k*N2 + j*N + i];
        lapu[k] = 0;
      }
    }
  }

  for(int k = 0; k < N; ++k){
    for(int j = 0; j < N; ++j; inner1){
      for(int i = 0; i < N; ++i; inner0){
        // Store a plane of data
        Lu[j][i] = uk[k];

        // Derivative in the z direction
        ut = 0;

        for(int m = 0; m < N; ++m)
          ut += LD[k][m] * uk[m];
      }
    }

    for(int j = 0; j < N; ++j; inner1){
      for(int i = 0; i < N; ++i; inner0){
        // Derivative in the x and y directions
        ur = us = 0;

        for(int m = 0; m < N; ++m){
          ur += LD[i][m] * Lu[j][m];
          us += LD[m][j] * Lu[m][i];
        }

        Lv[j][i] = rx[e]*ur + sx[e]*us + tx[e]*ut;
        Lw[j][i] = ry[e]*ur + sy[e]*us + ty[e]*ut;
        GDut     = rz[e]*ur + sz[e]*us + tz[e]*ut;

        lapuk = J[e] * (lambda * uk[k]);
      }
    }

    for(int j = 0; j < N; ++j; inner1){
      for(int i = 0; i < N; ++i; inner0){
        for(int m = 0; m < N; ++m)
          lapuk += LD[m][j] * Lv[m][i];
        for(int m = 0; m < N; ++m)
          lapuk[m] += LD[k][m] * GDut;
        for(int m = 0; m < N; ++m)
          lapuk[m] += LD[m][i] * Lw[j][m]

        lapuk[k] += lapuk;
      }
    }

    for(int j = 0; j < N; ++j; inner1){
      for(int i = 0; i < N; ++i; inner0){
        u2[e*N3 + k*N2 + j*N + i] = lapu[k];
      }
    }
  }
}
```

Listing 6.1: Pseudocode for the discrete spectral element method operator found in equation (6.7)

For this application, it was of interest to examine performance in high-order methods. Hence, the provided timings are used to display the performance with relation to the polynomial order of the method. Timing comparisons in table 6.7 include OpenMP with 12 threads and OpenCL running on an Intel i7-3930K. Intel's `icpc` compiler and the GNU `gcc` compiler were compared alongside AMD's and Intel's OpenCL implementations. Performance results match with the data-reuse found at higher polynomial orders, thus increasing the floating point operations per byte loaded.

Kernel\Poly. Order	1	2	3	4	5	6
OCCA::OpenMP (gcc)	9.45803	14.8285	21.9099	23.1743	25.162	26.0598
OCCA::OpenMP (icpc)	14.9699	20.0807	21.5094	23.4524	25.8857	27.682
OCCA::OpenCL (Intel)	9.8729	14.0412	16.7597	19.2133	26.7667	34.9429
OCCA::OpenCL (AMD)	2.97154	3.67021	3.97436	4.09612	4.16386	4.1186

Table 6.7: This table displays performance in GFLOPS, comparing OpenMP and OpenCL on the same CPU processor. In addition, we capture the performance difference when using different compilers for the OpenMP backend as well as different OpenCL vendor implementations. Tests were run on a 6-core Intel i7-3930K processor.

To examine the performance of these algorithms in a GPU architecture, timing comparisons were provided in table 6.8 to include CUDA and OpenCL running on an NVIDIA K40c GPU and OpenCL on an AMD Radeon 7990. Although the kernel is the same, the gap in performance between running CUDA and OpenCL on the K40 is substantial yet NVIDIA provides the drivers for both implementations.

Kernel\Poly. Order	1	2	3	4	5	6
OCCA::CUDA	20.334	53.8351	107.857	162.347	159.477	206.594
OCCA::OpenCL (NVIDIA)	15.2936	36.2901	70.3809	95.6581	100.876	123.208
OCCA::OpenCL (AMD)	15.3175	42.6366	91.9491	225.795	280.854	289.601

Table 6.8: This table displays performance in GFLOPS, comparing CUDA and OpenCL on an NVIDIA K40, and OpenCL on an AMD 7990. In addition, we capture the performance difference when using NVIDIA's CUDA and NVIDIA's OpenCL implementation although they use the same LLVM backend.

As opposed to labeling a specific backend as the best, the portability gained using OCCA allows users to pick the best-performing backend for each device. Two cases were

illustrated through timing this application, testing the OpenMP compiler and choosing between CUDA or OpenCL when running on an NVIDIA GPU. While OpenCL provides portability to multiple backends, the performance is dependent on the vendor and thus having the option to re-run the same kernel using the CUDA backend often benefits in performance.

6.4 Discontinuous Galerkin

The last project discussed is PasiDG, a discontinuous Galerkin (DG) numerical solver for the shallow water equations (SWE) (Gandham et al. 2015). The equations examined are the two dimensional non-linear equations to simulate tsunami propagations,

$$\begin{aligned} \frac{\partial h}{\partial t} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} &= 0, \\ \frac{\partial}{\partial t}(hu) + \frac{\partial}{\partial x}\left(hu^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial y}(huv) &= -gh\frac{\partial B}{\partial x}, \\ \frac{\partial}{\partial t}(hv) + \frac{\partial}{\partial x}(huv) + \frac{\partial}{\partial y}\left(hu^2 + \frac{1}{2}gh^2\right) &= -gh\frac{\partial B}{\partial y}, \end{aligned} \tag{6.8}$$

with water depth h, spatial velocities u, v, bathymetry B, and gravity g. Grouping equation (6.8) by partial derivatives simplifies the equations to

$$q_t + \nabla \cdot [F(q), G(q)] = S(q), \tag{6.9}$$

where

$$q = \begin{bmatrix} p \\ u \\ v \\ w \end{bmatrix}, \; F(q) = \begin{bmatrix} hu \\ hu + \frac{gh^2}{2} \\ huv \end{bmatrix}, \; G(q) = \begin{bmatrix} hv \\ huv \\ hv + \frac{gh^2}{2} \end{bmatrix}, \; S(q) = \begin{bmatrix} 0 \\ -gh\frac{\partial B}{\partial x} \\ -gh\frac{\partial B}{\partial y} \end{bmatrix}$$

Although a brief description of the DG method was provided in subsection 5.2.3, the example was based on the wave equation. The similarities between the two conservative equations, equation (5.4) and equation (6.9), causes a similarity between the SWE operators and those found in code listing 5.4 and code listing 5.5. I refer the reader to (Gandham et al. 2015) or (Gandham 2015) for additional detail in the implementations.

For this application, it was of interest to examine performance in high-order methods. Hence, the provided timings are used to display the performance with relation to the polynomial order of the method. Timing comparisons in table 6.9 include OpenMP with 12 threads and OpenCL running on an Intel i7-3930K. Performance results show the matrix-free algorithm behaves better at higher-orders, mainly due to the data-reuse and thus increasing the floating point operations per byte loaded.

To examine the performance of these algorithms in a GPU architecture, timing comparisons were provided in table 6.10 to include CUDA and OpenCL running on an NVIDIA K40 GPU and OpenCL on an AMD Radeon 7990. Similar to the results obtained for the spectral element method, a performance gap is visible between NVIDIA's CUDA and OpenCL implementations on the K40 GPU.

6.5 Rodinia Benchmarks

Kernel\Poly. Order	1	2	3	4	5	6
OCCA::OpenMP (icpc)	45.7	58.6	64.2	64.0	47.6	35.0
OCCA::OpenCL (Intel)	38.7	49.8	188	112.8	98.7	104

Table 6.9: This table displays performance in GFLOPS, comparing OpenMP using the `icpc` compiler and Intel's OpenCL implementation on the same CPU processor. Tests were run on a 6-core Intel i7-3930K processor.

Kernel\Poly. Order	1	2	3	4	5	6
OCCA::CUDA	824	1047	1255	1361	752	659
OCCA::OpenCL (NVIDIA)	576	707	856	1025	464	410
OCCA::OpenCL (AMD)	446	716	1121	1253	913	729

Table 6.10: This table displays performance in GFLOPS, comparing CUDA and OpenCL on an NVIDIA K40, and OpenCL on an AMD 7990. In addition, we capture the performance difference when using NVIDIA's CUDA and NVIDIA's OpenCL implementation although they use the same LLVM backend.

Performance validations are shown through a range of algorithms by implementing a subset of benchmarks seen in Rodinia (Che et al. 2009). The benchmarks were motivated by a study aiming to encompass an abundance of the applications found in the HPC community (Asanovic et al. 2006). From the benchmarks, I picked the back propagation, breadth-first-search, and Gaussian elimination mini-apps to benchmark and compare. All OCCA ports used automatic memory management and a single OKL kernel to assist the validation of performance, portability, and ease-of-development.

Comparisons include timing results using OpenMP, CUDA, and OpenCL in addition to multithreaded scaling results for OpenMP. Timings for the native OpenMP and OCCA OpenMP backend implementations were conducted on an Intel Xeon CPU E5-2650 and an Intel i7-4820K. The GPUs used to test the CUDA, OpenCL and their OCCA counterparts include an NVIDIA K20c and an NVIDIA 980.

6.5.1 Back Propagation

Implementation	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
(1) Native	0.218	0.171 (64%)	0.096 (56%)	0.075 (36%)	0.036 (38%)
(1) OKL	0.225	0.181 (62%)	0.102 (55%)	0.053 (52%)	0.033 (42%)
(2) Native	0.209	0.115 (91%)	0.067 (77%)	0.054 (49%)	0.107 (12%)
(2) OKL	0.205	0.113 (91%)	0.079 (65%)	0.038 (67%)	0.066 (19%)

Table 6.11: The table displays timings and scaling between the native back propagation OpenMP kernel and its respective OKL port running in OpenMP-mode. Tests were run to compute X on: (1) two 8-core Intel Xeon CPU E5-2650 processors, (2) 4-core Intel i7-4820K.

Implementation	Timings in seconds
(1) Native CUDA	0.0484
(1) Native OpenCL	0.0344
(1) OKL::CUDA	0.0084
(1) OKL::OpenCL	0.0105
(2) Native CUDA	0.0243
(2) Native OpenCL	0.0378
(2) OKL::CUDA	0.0177
(2) OKL::OpenCL	0.0190

Table 6.12: The table displays timings and speedups from using OCCA from the native back propagation CUDA kernel and its respective OKL port, comparing CUDA and OpenCL. Tests were run to compute X on: (1) NVIDIA K20c, (2) NVIDIA 980.

6.5.2 Breadth-First Search

Implementation	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
(1) Native	0.10218	0.06140 (83%)	0.05439 (47%)	0.04006 (32%)	0.03245 (20%)
(1) OKL	0.13212	0.10250 (64%)	0.05944 (56%)	0.03767 (44%)	0.02840 (29%)
(2) Native	0.07699	0.04809 (80%)	0.03371 (57%)	0.05505 (17%)	0.04739 (10%)
(2) OKL	0.10387	0.05600 (92%)	0.03243 (80%)	0.03375 (38%)	0.03412 (19%)

Table 6.13: The table displays timings and scaling between the native breadth-first search OpenMP kernel and its respective OKL port running in OpenMP-mode. Tests were run to compute X on: (1) two 8-core Intel Xeon CPU E5-2650 processors, (2) 4-core Intel i7-4820K.

Implementation	Timings in seconds
(1) Native CUDA	0.00845
(1) Native OpenCL	0.00918
(1) OKL::CUDA	0.01044
(1) OKL::OpenCL	0.01033
(2) Native CUDA	0.00533
(2) Native OpenCL	0.00537
(2) OKL::CUDA	0.00493
(2) OKL::OpenCL	0.00840

Table 6.14: The table displays timings and speedups from using OCCA from the native breadth-first search CUDA kernel and its respective OKL port, comparing CUDA and OpenCL. Tests were run to compute X on: (1) NVIDIA K20c, (2) NVIDIA 980.

6.5.3 Gaussian Elimination

Implementation	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
(1) OKL	2.47567	1.34976 (92%)	0.92982 (67%)	0.65721 (47%)	0.49321 (31%)
(2) OKL	1.81264	0.98354 (92%)	0.74674 (61%)	0.39831 (57%)	0.65954 (17%)

Table 6.15: The table displays timings and scaling for the Gaussian elimination kernels ported into OKL, no native OpenMP code was given for this benchmark. Tests were run to compute X on: (1) two 8-core Intel Xeon CPU E5-2650 processors, (2) 4-core Intel i7-4820K.

Implementation	Timings in seconds (Speedup relative to native code)
(1) Native CUDA	0.292
(1) Native OpenCL	0.277
(1) OKL::CUDA	0.112
(1) OKL::OpenCL	0.182
(2) Native CUDA	0.178
(2) Native OpenCL	0.126
(2) OKL::CUDA	0.103
(2) OKL::OpenCL	0.129

Table 6.16: The table displays timings and speedups from using OCCA from the native Gaussian elimination CUDA kernel and its respective OKL port, comparing CUDA and OpenCL. Tests were run to compute X on: (1) NVIDIA K20c, (2) NVIDIA 980.

6.6 Concluding Remarks

A spectrum of numerical methods were investigated to demonstrate the performance and their portability of OCCA kernels. The finite difference method comparisons showed OCCA kernels can match and surpass natively coded kernels, but can lack in performance portability. Results for the finite element method and discontinuous Galerkin examples exhibited high throughput with varying performance across different backends, for example CUDA vs OpenCL. The last demonstrated numerical method came from a collaboration with Argonne National Lab to optimize a Monte Carlo mini-app, where a single OCCA-mediated kernel out-performed the OpenMP and CUDA native implementations.

Additionally, a subset of benchmarks from the Rodinia benchmark suite were ported to use OCCA and compared with the available OpenMP, OpenCL, and CUDA implementations. In all cases, one kernel was used to outperform the native OpenMP, OpenCL, and CUDA implementations.

I investigated the application of OCCA on various numerical methods and diverse

benchmarks to validate that OCCA kernels compete with native implementations of the supported backends. Aside of the range of testing, I also note that throughout its development, OCCA has been included in other applications. Examples include: discontinuous Galerkin for shallow water equations (Gandham et al. 2015), continuous and discontinuous Galerkin methods for atmospheric modeling (Wilcox et al. 2013), and lattice Boltzmann for core sampling (Chen et al. 2015).

7 Conclusions and Future Work

The aim of this thesis work was to provide a unified programming model towards the evolving heterogeneous systems commonly seen in high performance computing. While we can only predict the future, I analyzed the changes on traditional multicore processors and popular many-core graphics processing units to reason the use of parallel models prevalent in current and upcoming architectures. By abstracting these programming models, I have provided a model that when used obtains comparable results with their native implementations. In addition, methods tailored for facilitate programming numerical methods in the OCCA programming model are described and implemented. I will briefly summarize contributions that have been stated throughout the thesis and provide suggestions for future work on OCCA.

7.1 Conclusions

Chapter 2 and chapter 3 introduced kernel language specifications for unifying several backends (serial code, multithreading with Pthreads, OpenMP, OpenCL, CUDA, and COI) alongside their foundation intermediate representation. Chapter 2 described the OCCA intermediate representation (IR), unifying parallel languages and specifications, and the OCCA application programming interface (API) used to interact with the offload model. Chapter 3 covered the OCCA kernel language (OKL) and OCCA Fortran language (OFL), C and Fortran extensions which utilize the OCCA IR for exposing parallelism in native languages. By providing a modular intermediate representation and a customized parser, appending backends introduced in the future is possible and

has been shown with the incorporation of the COI backend.

Supplemental tools are described in chapter 4 and chapter 5 which focused on adding additional layers of automating for kernel generations. Chapter 4 described a solution for automating data movement between the *host* and *device*, masking the offload model by only requiring the user to provide device synchronizations. Chapter 5 focused on automating kernel generation through the use of array bound analysis and loop-carried dependency detections, expediting porting across serial code to OKL or OFL based kernels.

The thesis concludes by providing examples validating the portability and performance proposed by the OCCA project in chapter 6. Investigated examples include applications on a range of numerical applications and a subset of Rodinia benchmarks, some which include a contrast between performance obtained with the use of OCCA and its native counterpart. Although performance portability is not always achieved, most comparisons exhibited OCCA kernels to perform equally or outperform their native counterparts due to run-time information at compile-time.

7.2 Future Work

While I achieved the goal of providing and implementing kernel specifications together with automating tools, the substantial scope of the OCCA project leaves a large number of unfinished tasks. For example, enhancements can be applied to the OKL and OFL kernel language specifications to leverage the parser on applying code transformations as opposed to the programmer. Additional data segment detections can be applied to the automated data movement, moderating the data transfers between the *host* and *device*. Less conservative testing can lead to more accurate detection of loop-carried dependencies in OAK and OAF kernels, presenting a larger range of code to be automatically ported to OKL and OFL. All efforts end up cooperating for the

greater goal of providing embedded kernels inside application *host* code.

A Appendix: OCCA Kernel Keywords

Provided are OCCA keywords categorized in tables by their use or purpose. Each table provides similar OCCA keywords on the left-most column. Adjacent to the OCCA keyword column is the OpenMP macro expansion, followed by the OpenCL macro expansion and the CUDA macro expansion on the right-most column.

OCCA	OpenMP	OpenCL	CUDA
`occaInnerId0`		`get_local_id(0)`	`threadIdx.x`
`occaInnerId1`		`get_local_id(1)`	`threadIdx.y`
`occaInnerId2`		`get_local_id(2)`	`threadIdx.z`
`occaOuterId0`		`get_group_id(0)`	`blockIdx.x`
`occaOuterId1`		`get_group_id(1)`	`blockIdx.y`
`occaGlobalId0`	`occaInnerId0` `+ occaInnerDim0*occaOuterId0`	`get_global_id(0)`	`threadIdx.x` `+ blockIdx.x*blockDim.x`
`occaGlobalId1`	`occaInnerId1` `+ occaInnerDim1*occaOuterId1`	`get_global_id(1)`	`threadIdx.y` `+ blockIdx.y*blockDim.y`
`occaGlobalId2`	`occaInnerId2`	`get_global_id(2)`	`threadIdx.z`

Table A.1: OCCA keywords used to obtain a scope's work-group and work-item ID.

OCCA	OpenMP	OpenCL	CUDA
occaInnerDim0	occaDims[0]	get_local_size(0)	blockDim.x
occaInnerDim1	occaDims[1]	get_local_size(1)	blockDim.y
occaInnerDim2	occaDims[2]	get_local_size(2)	blockDim.z
occaOuterDim0	occaDims[3]	get_num_groups(0)	gridDim.x
occaOuterDim1	occaDims[4]	get_num_groups(1)	gridDim.y
occaGlobalDim0	occaInnerDim0*occaOuterDim0	get_global_size(0)	occaInnerDim0*occaOuterDim0
occaGlobalDim1	occaInnerDim1*occaOuterDim1	get_global_size(1)	occaInnerDim1*occaOuterDim1
occaGlobalDim2	occaInnerDim2*occaOuterDim2	get_global_size(2)	occaInnerDim2

Table A.2: OCCA keywords related to storing work-group and work-item sizes.

OCCA	OpenMP	OpenCL	CUDA
occaInnerFor	occaInnerFor2 occaInnerFor1 occaInnerFor0		
occaInnerFor2	for(occaInnerId2 = 0; occaInnerId2 < occaInnerDim2; ++occaInnerId2)		
occaInnerFor1	for(occaInnerId1 = 0; occaInnerId1 < occaInnerDim1; ++occaInnerId1)		
occaInnerFor0	for(occaInnerId0 = 0; occaInnerId0 < occaInnerDim0; ++occaInnerId0)		
occaOuterFor2	for(occaOuterId2 = 0; occaOuterId2 < occaOuterDim2; ++occaOuterId2)		
occaOuterFor1	for(occaOuterId1 = 0; occaOuterId1 < occaOuterDim1; ++occaOuterId1)		
occaOuterFor0	for(occaOuterId0 = 0; occaOuterId0 < occaOuterDim0; ++occaOuterId0)		
occaGlobalFor2	occaInnerFor2		
occaGlobalFor1	occaOuterFor1 occaInnerFor1		
occaGlobalFor0	occaOuterFor0 occaInnerFor0		

Table A.3: OCCA keywords related to explicitly displaying work-group and work-item loop scopes.

OCCA	OpenMP	OpenCL	CUDA
occaShared		__local	__shared__
occaPointer		__global	
occaConstant		__constant	__constant__
occaVariable			
occaRestrict	__restrict__	restrict	__restrict__
occaVolatile		volatile	__volatile__
occaConst	const	const	const
occaAligned	__attribute__ ((aligned (__BIGGEST_ALIGNMENT__)))		

Table A.4: OCCA keywords related to OCCA variables attributes.

OCCA	OpenMP	OpenCL	CUDA
occaKernelInfoArg	const int *occaDims	__global int *dims	int *dims
occaFunctionInfoArg	const int *occaDims, int occaInnerId0, int occaInnerId1, int occaInnerId2	int _dummy	int dummy
occaFunctionInfo	occaDims, occaInnerId0, occaInnerId1, occaInnerId2	999	1
occaKernel	extern "C"	__kernel	extern "C" __global__
occaFunction			__device__
occaFunctionShared		__local	
occaInnerReturn	continue;	return;	return;
occaParallelFor	_Pragma("omp parallel for")		

Table A.5: OCCA keywords related to kernel prototypes and kernel setup.

OCCA	OpenMP	OpenCL	CUDA
`occaLocalMemFence`		`CLK_LOCAL_MEM_FENCE`	
`occaGlobalMemFence`		`CLK_GLOBAL_MEM_FENCE`	
`occaBarrier(Fence)`		`barrier(Fence)`	`__syncthreads();`

Table A.6: OCCA barriers needed in parallel threading synchronization.

OCCA	OpenMP	OpenCL	CUDA
`occaPrivateArray`	`occaPrivateClass<type,sz> name`	`type name[n]`	`type name[n]`
`occaPrivate`	`occaPrivateClass<type,1 > name`	`type name`	`type name`

Table A.7: OCCA keywords expanding to platform-dependent private memory types and used to carry over loop-breaks in OpenMP due to barriers.

OCCA	OpenMP	OpenCL	CUDA
`occaCPU`	`1`	`0`	`0`
`occaGPU`	`0`	`1`	`1`
`occaOpenMP`	`1`	`0`	`0`
`occaOpenCL`	`0`	`1`	`0`
`occaCUDA`	`0`	`0`	`1`

Table A.8: OCCA keywords specifying platform for platform-dependent kernel optimization.

References

Advanced Micro Devices 2013, 'Porting cuda applications to opencl'.
URL: *http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/porting-cuda-applications-to-opencl*

Allen, F. E. and Cocke, J. 1972, Catalogue of Optimizing Transformations, *in* R. Rustin, ed., 'Design and Optimization of Compilers', Prentice-Hall, Englewood Cliffs, NJ, pp. 1–30.

Allen, R. and Johnson, S. 1988, Compiling C for vectorization, parallelization, and inline expansion, *in* 'ACM SIGPLAN Notices', Vol. 23, ACM, pp. 241–249.

Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., McMahon, J. O., Pasquier, F.-X., Péan, G., Villalon, P. et al. 2012, Par4all: From convex array regions to heterogeneous computing, *in* 'IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012'.

Anderson, M. 2014, A Framework for Composing High-Performance OpenCL from Python Descriptions.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W. et al. 2006, The landscape of parallel computing research: A view from berkeley, Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Ayguadé, E., Badia, R. M., Igual, F. D., Labarta, J., Mayo, R. and Quintana-Ortí, E. S. 2009, An extension of the StarSs programming model for platforms with multiple GPUs, *in* 'Euro-Par 2009 Parallel Processing', Springer, pp. 851–862.

Banerjee, U., Chen, S.-C., Kuck, D. J. and Towle, R. A. 1979, Time and parallel processor bounds for Fortran-like loops, *Computers, IEEE Transactions on* **100**(9), 660–670.

Bangerth, W., Hartmann, R. and Kanschat, G. 2007, deal. II - a general-purpose object-oriented finite element library, *ACM Transactions on Mathematical Software (TOMS)* **33**(4), 24.

Bell, N. and Garland, M. 2008, Efficient sparse matrix-vector multiplication on CUDA, Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

Bell, N. and Hoberock, J. 2011, Thrust: A productivity-oriented library for CUDA,

GPU Computing Gems **7**.

Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A. and Bastoul, C. 2010, The polyhedral model is more widely applicable than you think, *in* 'Compiler Construction', Springer, pp. 283–303.

Bender, M. A., Berry, J., Hammond, S. D., Hemmert, K. S., McCauley, S., Moore, B., Moseley, B., Phillips, C. A., Resnick, D. and Rodrigues, A. 2015, Two-Level Main Memory Co-Design: Multi-Threaded Algorithmic Primitives, Analysis, and Simulation.

Besson, J. and Foerch, R. 1997, Large scale object-oriented finite element code design, *Computer Methods in Applied Mechanics and Engineering* **142**(1), 165–187.

Bodin, F. and Bihan, S. 2009, Heterogeneous multicore parallel programming for graphics processing units, *Scientific Programming* **17**(4), 325–336.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P. 2004, Brook for GPUs: stream computing on graphics hardware, *in* 'ACM Transactions on Graphics (TOG)', Vol. 23, ACM, pp. 777–786.

Callahan, D., Cooper, K. D., Kennedy, K. and Torczon, L. 1986, Interprocedural constant propagation, *in* 'ACM SIGPLAN Notices', Vol. 21, ACM, pp. 152–161.

Campbell, A. 2009, 'IBM have not stopped Cell processor development'.
URL: *http://www.hardwareheaven.com/news.php?newsid=344*

Chen, C., Wang, Z., Majeti, D., Vrvilo, N., Warburton, T., Sarkar, V. and Li, G. 2015, Optimization of Lattice Boltzmann Simulation by GPU Parallel Computing and the Application in Reservoir Characterization.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K. 2009, Rodinia: A benchmark suite for heterogeneous computing, *in* 'Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on', IEEE, pp. 44–54.

Cooper, K. and Torczon, L. 2011, *Engineering a compiler*, Elsevier.

Dally, W. 2009, The End of Denial Architecture and the Rise of Throughput Computing, Keynote at Async Conference UNC.

Dastgeer, U., Enmyren, J. and Kessler, C. W. 2011, Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems, *in* 'Proceedings of the 4th International Workshop on Multicore Software Engineering', ACM, pp. 25–32.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J. and Yelick, K. 2008, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, *in* 'Proceedings of the 2008 ACM/IEEE

conference on Supercomputing', IEEE Press, p. 4.

Demidov, D. 2012, 'VexCL: Vector Expression Template Library for OpenCL'.

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

Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E. and LeBlanc, A. R. 1974, Design of ion-implanted MOSFET's with very small physical dimensions, *Solid-State Circuits, IEEE Journal of* **9**(5), 256–268.

Diamos, G. F., Kerr, A. R., Yalamanchili, S. and Clark, N. 2010, Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems, *in* 'Proceedings of the 19th international conference on Parallel architectures and compilation techniques', ACM, pp. 353–364.

Dolbeau, R., Bihan, S. and Bodin, F. 2007, HMPP: A hybrid multi-core parallel programming environment, *in* 'Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)'.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G. and Dongarra, J. 2012, From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, *Parallel Computing* **38**(8), 391–407.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X. and Planas, J. 2011, Ompss: a proposal for programming heterogeneous multi-core architectures, *Parallel Processing Letters* **21**(02), 173–193.

Du, X., Fletcher, R. and Fowler, P. 2008, A new pseudo-acoustic wave equation for vti media, *in* '70th EAGE Conference & Exhibition'.

Edwards, H. C. and Trott, C. R. 2013, Kokkos: Enabling performance portability across manycore architectures, *in* 'Extreme Scaling Workshop (XSW), 2013', IEEE, pp. 18–24.

Enmyren, J. and Kessler, C. W. 2010, SkePU: a multi-backend skeleton programming library for multi-GPU systems, *in* 'Proceedings of the fourth international workshop on High-level parallel programming and applications', ACM, pp. 5–14.

Fahrenholtz, S. J., Moon, T., Franco, M., Medina, D., Danish, S., Gowda, A., Shetty, A., Maier, F., Hazle, J., Stafford, R. J., Warburton, T. and Fuentes, D. 2015, A model evaluation study for treatment planning of laser induced thermal therapy.

Fang, J., Varbanescu, A. L. and Sips, H. 2011, A comprehensive performance comparison of CUDA and OpenCL, *in* 'Parallel Processing (ICPP), 2011 International Conference on', IEEE, pp. 216–225.

Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S. and Schwan, K. 2011, A framework for dynamically instrumenting GPU compute applications within GPU Ocelot, *in* 'Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units', ACM, p. 9.

Ferrer, R., Planas, J., Bellens, P., Duran, A., Gonzalez, M., Martorell, X., Badia, R. M., Ayguade, E. and Labarta, J. 2011, Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL, *in* 'Languages and Compilers for Parallel Computing', Springer, pp. 215–229.

Fischer, P. F., Lottes, J. W. and Kerkemeier, S. G. 2008, nek5000 web page, *Web page: http://nek5000. mcs. anl. gov* .

Fischer, P. F. and Patera, A. T. 1991, Parallel spectral element solution of the Stokes problem, *Journal of Computational Physics* **92**(2), 380–421.

Fischer, P. F. and RÃČÂÿnquist, E. M. 1994, Spectral element methods for large scale parallel NavierÃćâĆňâĂİStokes calculations, *Computer Methods in Applied Mechanics and Engineering* **116**(1ÃćâĆňâĂIJ4), 69 – 76.
URL: *http://www.sciencedirect.com/science/article/pii/S004578259480009X*

Fischer, P., Lottes, J., Siegel, A. and Palmiotti, G. 2007, Large Eddy Simulation of wire-wrapped fuel pins I: hydrodynamics in a Periodic Array, *in* 'Joint International Topical Meeting on Mathematics & Computation and Super computingin Nuclear Applications (M&C+ SNA 2007)'.

Gandham, R. 2015, High Performance High-Order Numerical Methods: Applications in Ocean Modeling, PhD thesis, Rice University.

Gandham, R., Medina, D. and Warburton, T. 2015, GPU Accelerated discontinuous Galerkin methods for shallow water equations, *Communications in Computational Physics* .

Gardner, M., Sathre, P., Feng, W.-c. and Martinez, G. 2013, Characterizing the challenges and evaluating the efficacy of a CUDA-to-OpenCL translator, *Parallel Computing* **39**(12), 769–786.

Giraldo, F. X. and Rosmond, T. E. 2003, A scalable spectral element Eulerian atmospheric model (SEE-AM) for NWP: dynamical core tests, Technical report, DTIC Document.

Golub, G. H. and Van Loan, C. F. 2012, *Matrix computations*, Vol. 3, JHU Press.

Gregory, K. and Miller, A. 2012, *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*, " O'Reilly Media, Inc.".

Grewe, D., Wang, Z. and O'Boyle, M. F. 2013, Portable mapping of data parallel programs to opencl for heterogeneous systems, *in* 'Code Generation and

Optimization (CGO), 2013 IEEE/ACM International Symposium on', IEEE, pp. 1–10.

Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A. and Pouchet, L.-N. 2011, Polly-Polyhedral optimization in LLVM, *in* 'Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)', Vol. 2011.

Han, T. D. and Abdelrahman, T. S. 2009, hi CUDA: a high-level directive-based language for GPU programming, *in* 'Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units', ACM, pp. 52–61.

Han, T. D. and Abdelrahman, T. S. 2011, hiCUDA: High-level GPGPU programming, *Parallel and Distributed Systems, IEEE Transactions on* **22**(1), 78–90.

Harvey, M. J. and De Fabritiis, G. 2011, Swan: A tool for porting CUDA programs to OpenCL, *Computer Physics Communications* **182**(4), 1093–1099.

Herdman, J., Gaudin, W., McIntosh-Smith, S., Boulton, M., Beckingsale, D., Mallinson, A. and Jarvis, S. A. 2012, Accelerating hydrocodes with OpenACC, OpeCL and CUDA, *in* 'High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:', IEEE, pp. 465–471.

Hesthaven, J. S. and Warburton, T. 2007, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, Vol. 54, Springer Science & Business Media.

Hornung, R. and Keasler, J. 2013, A Case for Improved C++ Compiler Support to Enable Performance Portability in Large Physics Simulation Codes, Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

Johnson, S. C. 1975, *Yacc: Yet another compiler-compiler*, Vol. 32, Bell Laboratories Murray Hill, NJ.

Jordan, H., Pellegrini, S., Thoman, P., Kofler, K. and Fahringer, T. 2013, INSPIRE: The insieme parallel intermediate representation, *in* 'Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on', IEEE, pp. 7–17.

Karimi, K., Dickson, N. G. and Hamze, F. 2010, A performance comparison of CUDA and OpenCL, *arXiv preprint arXiv:1005.2581* .

Karniadakis, G., Israeli, M. and Orszag, S. 1991, High-order splitting methods for the incompressible Navier-Stokes equations , *Journal of computational physics* **97**(2), 414–443.

Klöckner, A., Warburton, T., Bridge, J. and Hesthaven, J. S. 2009, Nodal discontinuous galerkin methods on graphics processors, *Journal of Computational Physics* **228**(21), 7863–7882.

Lashgar, a., Majidi, A. and Baniasadi, A. 2014, IPMACC: Open Source OpenACC to CUDA/OpenCL Translator, *arXiv preprint arXiv:1412.1127* .

Lattner, C. A. 2002, LLVM: An infrastructure for multi-stage optimization, PhD thesis, University of Illinois at Urbana-Champaign.

Lattner, C. and Adve, V. 2004, LLVM: A compilation framework for lifelong program analysis & transformation, *in* 'Code Generation and Optimization, 2004. CGO 2004. International Symposium on', IEEE, pp. 75–86.

Lee, S. and Eigenmann, R. 2010, OpenMPC: Extended OpenMP programming and tuning for GPUs, *in* 'Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis', IEEE Computer Society, pp. 1–11.

Lee, S. and Vetter, J. S. 2014, OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing, *in* 'Proceedings of the 23rd international symposium on High-performance parallel and distributed computing', ACM, pp. 115–120.

Martinez, G., Gardner, M. and Feng, W.-c. 2011, CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures, *in* 'Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on', IEEE, pp. 300–307.

Medina, D. 2014, 'OCCA: A Unified Approach to Multi-Threading Languages'.

Medina, D. S., St-Cyr, A. and Warburton, T. 2014, OCCA: A unified approach to multi-threading languages, *arXiv preprint arXiv:1403.0968* .

Medina, D. S., St-Cyr, A. and Warburton, T. 2015, High-Order Finite-differences on multi-threaded architectures using OCCA, *in* 'ICOSAHOM 2015', Springer.

Micikevicius, P. 2009, 3D finite difference computation on GPUs using CUDA, *in* 'Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units', ACM, pp. 79–84.

Modave, A., St-Cyr, A., Warburton, T. and Mulder, W. 2015, Accelerated Discontinuous Galerkin Time-domain Simulations for Seismic Wave Propagation, *in* '77th EAGE Conference and Exhibition 2015'.

Munshi, A. 2008, OpenCL: Parallel Computing on the GPU and CPU, *SIGGRAPH, Tutorial* .

Necula, G. C., McPeak, S., Rahul, S. P. and Weimer, W. 2002, CIL: Intermediate language and tools for analysis and transformation of C programs, *in* 'Compiler Construction', Springer, pp. 213–228.

Padua, D. A. and Wolfe, M. J. 1986, Advanced compiler optimizations for

supercomputers, *Communications of the ACM* **29**(12), 1184–1201.

Pheatt, C. 2008, Intel® threading building blocks, *Journal of Computing Sciences in Colleges* **23**(4), 298–298.

Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L. and Verbrugge, C. 2000, A framework for optimizing Java using attributes, *in* 'Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research', IBM Press, p. 8.

Quinlan, D. 2000, ROSE: Compiler support for object-oriented frameworks, *Parallel Processing Letters* **10**(02n03), 215–226.

Rahaman, R., Medina, D., Lund, A., Tramm, J., Warburton, T. and Seigel, A. 2015, Portability and Performance of Nuclear Reactor Simulations on Many-Core Architectures, *in* '2015 Exascale Applications and Software Conference'.

Reyes, R., López, I., Fumero, J. and de Sande, F. 2012, A comparative study of OpenACC implementations, *Jornadas Sarteco* .

Rivière, B. 2008, *Discontinuous Galerkin methods for solving elliptic and parabolic equations: theory and implementation*, Society for Industrial and Applied Mathematics.

Rogers, P. and FELLOW, A. C. 2013, Heterogeneous System Architecture Overview, *in* 'Hot Chips'.

Romano, P. K. and Forget, B. 2013, The OpenMC Monte Carlo particle transport code, *Annals of Nuclear Energy* **51**, 274–281.

Russell, R. M. 1978, The CRAY-1 computer system, *Communications of the ACM* **21**(1), 63–72.

Schaller, R. R. 1997, Moore's law: past, present and future, *Spectrum, IEEE* **34**(6), 52–59.

Starynkevitch, B. 2011, MELT-a Translated Domain Specific Language Embedded in the GCC Compiler, *arXiv preprint arXiv:1109.0779* .

Stilwell, N. 2013, gNek: A GPU Accelerated Incompressible Navier Stokes Solver, PhD thesis, Masters Thesis, Rice University. http://hdl.handle.net/1911/72043.

Thomsen, L. 1986, Weak elastic anisotropy, *Geophysics* **51**(10), 1954–1966.

Tramm, J. R., Siegel, A. R., Islam, T. and Schulz, M. 2014, XSBench–The development and verification of a performance abstraction for Monte Carlo reactor analysis, *mcs. anl. gov* .

Turbak, F. 2008, *Design Concepts in Programming Languages*, MIT press.

Ventroux, N., Sassolas, T., Guerre, A., Creusillet, B. and Keryell, R. 2012, SESAM/Par4All: a tool for joint exploration of MPSoC architectures and dynamic dataflow code generation, *in* 'Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools', ACM, pp. 9–16.

Wang, W., Xu, L., Cavazos, J., Huang, H. H. and Kay, M. 2014, Fast Acceleration of 2D Wave Propagation Simulations Using Modern Computational Accelerators, *PloS one* **9**(1), e86484.

Wienke, S., Springer, P., Terboven, C. and an Mey, D. 2012, OpenACCâĂŤfirst experiences with real-world applications, *in* 'Euro-Par 2012 Parallel Processing', Springer, pp. 859–870.

Wilcox, L. C., Giraldo, F. X., Campbell, T., Klöckner, A., Warburton, T. and Whitcomb, T. 2013, NPS-NRL-Rice-UIUC Collaboration on Navy Atmosphere-Ocean Coupled Models on Many-Core Computer Architectures Annual Report, Technical report, DTIC Document.

Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K. and Demmel, J. 2009, Optimization of sparse matrix–vector multiplication on emerging multicore platforms, *Parallel Computing* **35**(3), 178–194.

Wulf, W. A. and McKee, S. A. 1995, Hitting the Memory Wall: Implications of the Obvious, *ACM SIGARCH computer architecture news* **23**(1), 20–24.

Zhou, M., Symes, W. W. et al. 2014, Wave Equation Based Stencil Optimizations on Multi-Core CPU, *in* '2014 SEG Annual Meeting', Society of Exploration Geophysicists.