Accelerator Benchmark Suite

Using OpenACC Directives

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Pooja Chitral

December 2014

Accelerator Benchmark Suite

Using OpenACC Directives

Acknowledgements

I would like to express my gratitude to my advisor, Dr. Barbara Chapman, for giving me an excellent opportunity to be part of the HPC (High Performance Computing) Tools group. It was an absolute privilege to work under her tutelage as she inspired me to explore prospective research topics and provided her feedback during the course of my stay. It’s my privilege to be part of such an enthusiastic, energetic, and inspiring team.

I would also like to extend my sincere gratitude to my mentor, Dr. Sunita Chandrashekaran. Her continuous guidance and support throughout my research work have been tremendous. Her directions at every point of my thesis helped me understand the problem better and motivated me to resolve them in the right spirit.

I would like to thank everyone in the HPC Tools group who has helped and motivated me at one point or another. I extend my gratitude to the Department of Computer Science for giving me this opportunity to be part of such a prestigious department at the University of Houston.

Last but not least, I would like to thank my family and friends who motivated and supported me throughout this journey. I would like to thank one of the most important persons in my life, my grandmother, for always inspiring me and believing in me.

Accelerator Benchmark Suite Using OpenACC Directives

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Pooja Chitral

December 2014

Abstract

In recent years, GPU computing has been very popular for scientific applications, especially after the release of programming languages like CUDA, OpenCL, and OpenACC. The growing popularity of GPU computation in commercial and scientific fields is attributed to the high computational power of GPU cores. The accelerator benchmark suite using OpenACC 2.0 is a combination of very popular benchmarks – the Parboil and NAS Parallel benchmarks. These benchmarks contain a wide range of throughput computing applications, which are useful for studying the performance of computing architectures and compilers. The Parboil benchmark includes applications from different scientific and commercial fields including image processing, biomolecular simulation, and astronomy. The NAS Parallel benchmark has a set of applications that target different areas of computational fluid dynamics.

The accelerator benchmark suite that has been designed exploits the computational power of GPU architecture by using the emerging directives and clauses provided by OpenACC 2.0. This benchmark can act as a reference point for new programmers in GPU computing, reducing the time taken to understand one of the most powerful parallel programming paradigms.

Finally, the goal of the accelerator benchmark is to evaluate the applicability of one of the high-level programming models OpenACC for accelerators. This benchmark will help provide the OpenACC community with valuable feedback to improve the model further.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Benchmarks are a set of programs designed for a very specific field, such as scientific computing, computational fluid dynamics, commercial applications, numerical analysis, and image processing. Many benchmarks are designed to target particular architectures. We use benchmarks for measuring performance characteristics related to the target platforms such as database management systems, I/O benchmarks, micro-benchmarks, parallel benchmarks, etc.

Benchmarks are extremely important in research, as they can be used to evaluate new technologies, architectures, and languages at their inception. Many vendors are willing to invest in the new technologies and research when the results of these benchmarks are promising. Benchmarks play a very important role in computer architecture research by helping programmers work on a set of shared data code and understand the behavior and performance of various technologies and system architecture. Benchmarks can also assist in understanding the best optimization techniques with respect to the computing architecture and the programming environment by enabling the comparison of different result sets.

1.1 Problem Statement

The main aim of the created accelerator benchmark is to understand GPU computing in general, specifically OpenACC. This setup gives insight into OpenACC 1.0 features and how OpenACC 2.0 has improved drastically by implementing more advanced features. OpenACC 2.0 simplifies application portability, thereby improving the productivity of the developer.

OpenACC is an emerging accelerator model, and a lot of work has been done to improve features of OpenACC while maintaining performance improvement and portability aspects in mind.

Porting using OpenACC 1.0 is cumbersome, as it involves a lot of code restructuring, function inlining, etc. With OpenACC 2.0, porting of an application is easy and can be done in a more natural way that helps achieve the best performance for sequential code. OpenACC 2.0 drastically improves productivity and also adds flexibility in data handling by providing clauses for expressing an unstructured data lifetime for the variables.

1.2 Thesis Contribution

- Developing an accelerator benchmark suite helped us study and analyze the emerging high-level programming model OpenACC 2.0. The benchmark is used to evaluate different types of computations to characterize execution behavior of several of the most frequently used OpenACC features. We also present different challenges while porting existing multithreaded accelerator benchmark suites to GPUs.

- Applications from the Parboil benchmark were ported to GPUs using OpenACC 2.0. The unique challenges while porting programs from OpenMP and the serial version to OpenACC are analyzed and discussed. As Parboil implements applications from different domains, the accelerator benchmark suite can give an overview of how to port applications from different architecture to accelerator architecture.

- OpenACC is an evolving programming model. Although NAS Parallel Benchmark codes were ported to OpenACC 1.0 [46], there were several limitations posed by the version of OpenACC (V 1.0) used for that work. In this thesis, those limitations are overcome by using the most recent version of the specification, OpenACC 2.0, that fills some of the programming gaps discussed in

[46]. Routine directives, unstructured data lifetime, and optimization techniques were some of the features used for this work.

- We analyzed the performance of OpenACC applications by comparing their performances with the alternative, accelerated version that used CUDA, OpenMP, and OpenCL. We optimized the accelerator benchmark suite applications by applying the loop collapse, loop unrolling, code restructuring, and some new data management techniques.

1.3 Thesis Outline

This thesis is organized into the following sections:

Chapter 2 introduces heterogeneous computing, the need for heterogeneous computing, and popular heterogeneous systems used in scientific and commercial computing. It also describes different challenges in heterogeneous computing.

Chapter 3 discusses the evolution of GPUs, the architecture of NVIDIA and AMD GPUs, including latest Kepler GPU from NVIDIA and Kaveri from AMD.

Chapter 4 presents some of the popular programming models for accelerators like CUDA, OpenMP, OpenACC, and OpenCL.

Chapter 5 provides the rationale for creating the accelerator benchmark suite. This chapter also involves discussion of OpenACC 2.0 features and how they are used in the accelerator benchmark. Further, we discuss implementation details of OpenACC 2.0 for NPB and Parboil benchmarks and performance analysis for the same.

Chapter 6 presents conclusions and related future work.

Chapter 2. Emerging Heterogeneous Systems

The heterogeneous computing environment, in contrast to the homogeneous systems, is a combination of different processing units. For some applications, we see considerable performance improvement when we combine different processing elements that incorporate special processing capabilities while handling some aspects of tasks, instead of adding more homogeneous cores [3].

A heterogeneous computing environment presents new challenges, as different architectures have different memory and execution models. The level of heterogeneity in the system can introduce non-uniformity in system development, programming practices, and overall system capability [1].

2.1 Need for Heterogeneous Computing

Understanding the nature of the application in terms of structure of the program, computational needs, and memory requirements will help us decide which devices will deliver better performance and productivity. Unlike CPUs, which are designed for handling large sequential programs with significant branching and complexity, GPUs are good for performing computationally intense applications with considerably less branching and complexity.

HPC allows exploitation of the inherent capabilities of a wide range of accelerators to solve computationally intensive problems.

The importance of heterogeneous computing is that we can achieve very large performance gains if we can consolidate and integrate the strengths of all the different platforms used. GPUs is excellent for floating point operations [4]. Multicore CPUs are good for command and control operations. FPGAs is an ideal solution for all kinds of operations other than floating point operations like binary, integer, fix point, text, and special formats [5]. As quoted by different companies that comprise the high-performance computing industry, “The future is heterogeneous” [54].

When we study the problems prevalent in heterogeneous systems, we divide the entire problem into many subproblems and then categorize and distribute these problems to different processing elements designed for specific types of operations.

2.2 Heterogeneous Systems

Coprocessors are used to supplement the functions of the primary processor or CPU. These are attached to the CPU via interconnect. Computationally intensive tasks can be offloaded to coprocessors in order to accelerate the performance of the application (coprocessors can be anything from general-purpose CPUs, GPUs, FPGAs, etc.).

2.2.1 Graphical Processing Unit (GPU)

The Graphical Processing Unit (GPU) is a specialized electronic circuit created to perform graphical processing operations. Computer graphics is fundamentally an “embarrassingly” parallel-problem as the workload can easily be spread across multiple compute unit, since each pixel on the screen can (mostly) be worked on independently [6].

GPUs are very fast compared to CPUs because every GPU contains hundreds of thousands “GPU cores” internally. And these cores work in parallel, as they are independent of each other while performing the computation. Thus, for some applications, using GPU cores is much better than adding more CPU cores.

As mentioned earlier, GPUs are specifically designed for graphics processing. Thus, they have very restricted operations, and their programming model is not same as CPU computing. To maximize GPU performance, programs are executed using streaming processing model [54].

A “stream” can simply be a set of data that requires similar computation, by providing data parallelism. As each data set is computed independently, there is no scope for having static or shared data in streaming processing. The order of the workflow would be to read the input, compute the results using input data, and then write the output back to the host device [54].

Ideally, GPU applications should have large data sets with structured loops for high parallelism and minimal dependency between data elements.

In our early efforts for harnessing the power of GPUs for general purpose computing, we used OpenGL and DirectX API. These APIs were difficult to understand and reduced productivity of the programmers. Later in 2006, NVIDIA developed a

CUDA programming extension for C/C++ and FORTRAN. OpenACC is a programming standard for parallel computing developed by Cray, CAPS, NVIDIA, and PGI.

2.2.2 Digital Signal Processor (DSP)

A digital signal processor (DSP) is a microprocessor chip that is widely used in many electronic devices. DSPs take signals as input, digitizes them by enhancing the quality, and then apply mathematical functions and manipulates them for the target application.

Though general-purpose DSPs can be used for many of the manipulations, it is very practical to have a dedicated DSP for specific applications. Portability is one of the major advantages of dedicated DSPs, as they address the power constraints associated with devices like mobile phones [7].

DSPs process the data in real time with precise output. Thus DSPs are used in audio signal processing, audio and video compression, speech processing and recognition, digital image processing, digital communications, biomedicine, seismology, and radar applications [7].

Digital signal processing typically requires a large number of mathematical operations to be performed concurrently on a large set of data. Heavy computation in the DSP is due to continuous conversion of data from analog to digital and vice versa [7].

One of the very important bottlenecks in digital signal processing is the transfer of data to and from memory. Memory-transfer operations typically include data from the signals to be processed, the instructions for processing the data, and the binary code that needs to be fed to the sequencer. Because of the high intensity of data processing and data movement in the memory DSPs have a special architecture [7].

2.2.2.1 Keystone Architecture

Keystone architecture is based on Texas Instruments (TI)'s System-on-Chip (SoC) processors. The design methodology and the architecture enable high-performance gains. This architecture gives the flexibility to include a single or multicore mix of DSP [8] [9]. Keystone architecture has a C66x Core Pac, memory subsystem, application-specific coprocessor, multicore navigator, network coprocessor, interfaces, embedded trace buffer, and system trace buffers for debugging. Keystone architecture can be used in many of the important wireless applications like base-station transceiver systems, cellular systems like 3G, 4G. Some of the media applications using keystone architecture are video infrastructure, medical imaging, military and defense, smart grids, etc. Because of the SoC concept, C66x-generation DSPs maximize throughput of on-chip data flows by eliminating the most important bottleneck caused by data flow. One of the very important features of this architecture is TeraNet, which is a packet-based high-speed non-blocking channel that transfers as much as two terabytes of data per second [8] [10].

With TeraNet and an extensive two-layer memory structure, data flows freely and effectively through C66x devices. Although it provides direct chip-to-chip connectivity for local devices, hyperlink is also integral to the internal processing architecture of C66x DSPs. The hyperlink is a fast and efficient interface with low protocol overhead and high throughput, running at an aggregate speed of 50 Gbps (four lanes at 12.5 Gbps each).

Working in conjunction with Multicore Navigator, Hyperlink transparently dispatches tasks to other local devices where they are executed as if they were being processed on local resources. In the keystone architecture, CorePac is defined as the main processing element in a multicore SoC. CorePac includes the infrastructure that supports the DSP cores, including shared memory and memory controllers. There are three levels of memory in the keystone architecture. Each C66x CorePac has level-1 program (L1P) and level-1 data (L1D) memory [8] [10].

2.3 Challenges in Heterogeneous Computing

Computing in a heterogeneous environment requires the separate compilation of code for different architectures. Mapping of the sub-programs on different architectures can sometimes be challenging. In a heterogeneous environment, program debugging becomes increasingly complicated because of heterogeneity [12].

2.3.1 Programming Model Support

When programming in a heterogeneous environment, it is very important to have an efficient programming model that will help us maximize computational output from the underlying architecture. The programming model should be easy to understand and follow a natural migration path from serial programs, without affecting the productivity of the programmer. These programming models should also allow portability across multiple machine generations. Finally, the programming model should provide evolved

API’s that will hide the low-level details of the underlying hardware, thus making heterogeneous programming seamless.

2.3.2 Data Transfer

In heterogeneous computing, the host is connected to all its devices via an interconnect. Thus, it is very important to understand the data transfer rate and limitations of interconnects to optimize the data transfers between hosts and devices. This will help in understanding how and when the data should be moved to the device and returned to the host. Neglecting data activity will undermine the entire computation and, in turn, lead to a performance reduction caused by bottlenecks in the interconnect.

Heterogeneous systems typically have hundreds and thousands of computing nodes when compared to many-core processors with more than 100 cores on a single electronic chip. This increases the distance between the memory and cores, placing limitations on the electronic networks used to connect the systems. The latency for accessing the external memory modules differs strongly, depending on the distance between the cores and memory and its locality in the network [11].

2.3.3 Memory Management

In multicore computing, parallelism is spread across multiple system levels, and the data used for computation is stored redundantly in memory subsystems at several levels. From 1986 to 2000, CPU speed improved at an annual rate of 55%, while memory speed only improved by 10% [13]. As cores become faster and faster, data moves in and out, leading to a bottleneck in the supply of data for computations.

As heterogeneous computing is host-directed, CPU allocates tasks to the connected subsystems. Thus, the entire computation moves to the device space, and then the results are brought back to the host.

As different systems have a different memory hierarchy and their private memory for computation, it is very important to synchronize these two memories between the host and device and also schedule tasks as effectively as possible.

The devices mainly differ in the configuration and arrangement of functional and control units, and the data flow from the main memory to the compute cores is organized differently. Consequently, the instruction set and the generic or vendor-specific programming concepts differ. In some programming approaches, only parts of the hardware structure are exposed to the user and are accessible to the programmer [1].

Chapter 3. GPUs for Extreme Computing – State-of-the-Art

With the emergence of extreme scale computing, modern GPUs have been extensively used in HPC applications like large data centers and supercomputers. The highly parallel nature of the GPU is attributed to its large number of compute cores and high-performance memory subsystems. We have GPUs from many different vendors. The most popular GPUs in the market are NVIDIA and AMD, which are used in high-performance computing. These two GPUs are different in many aspects with respect to the architecture, execution model, and memory hierarchy [23] [18].

3.1 Evolution of GPUs

Graphics processing units were created with graphic and image processing in mind. Entertainment and game industries use GPUs extensively for building games with high-quality graphics. A huge revolution in the GPU programming industry was when GPU industries realized that we could use the available programming infrastructure for performing scientific applications. This was when NVIDIA came up with C-like programming language called CUDA targeting scientific computing in 2008 [15]. In almost same time, AMD came up with OpenCL programming language for (Accelerator processing unit) APUs.

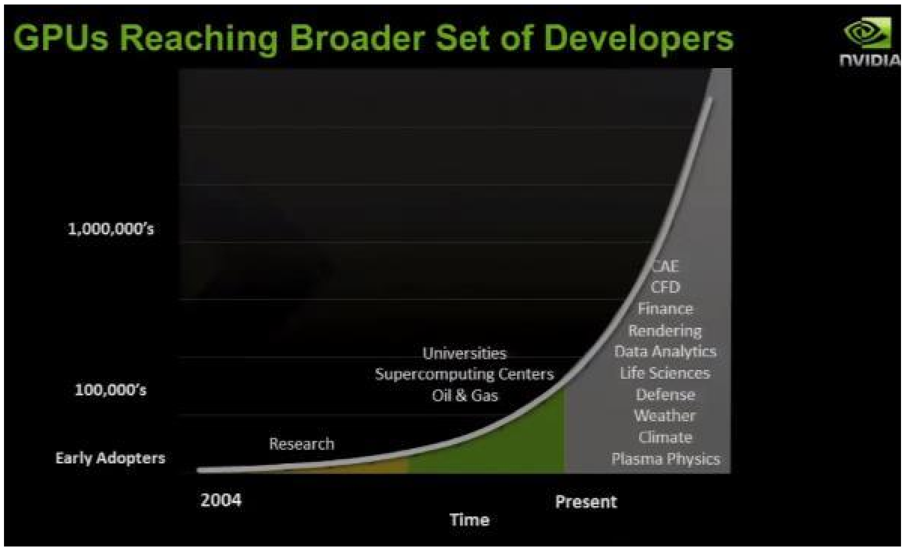

Figure 3.1: Trends in GPU Computing [15]

GPUs and their computing capabilities have changed over time, adding more computing power. In the gaming industry, these features enhance the user experience by adding rich graphics. The number of transistors in GPU hardware has increased drastically, providing increasingly advanced features in memory and execution models [15].

GPU computing tries to achieve the best performance by combining the power of both the CPU and GUP. CPUs are highly effective at making single threads go fast. Similarly, GPUs are excellent at creating thousands of threads and making all of them run really fast. In GPU computing, we combine the best computing capabilities of two worlds [18].

The basic idea behind GPU computing is running the majority of sequential code like file I/O, running the OS, and exchanging the data with sensors on the CPU. Then, we identify the computationally intensive parts of the program, which are highly parallel, and run them on GPUs. This helps to exploit both CPU and GPU architectures efficiently [18].

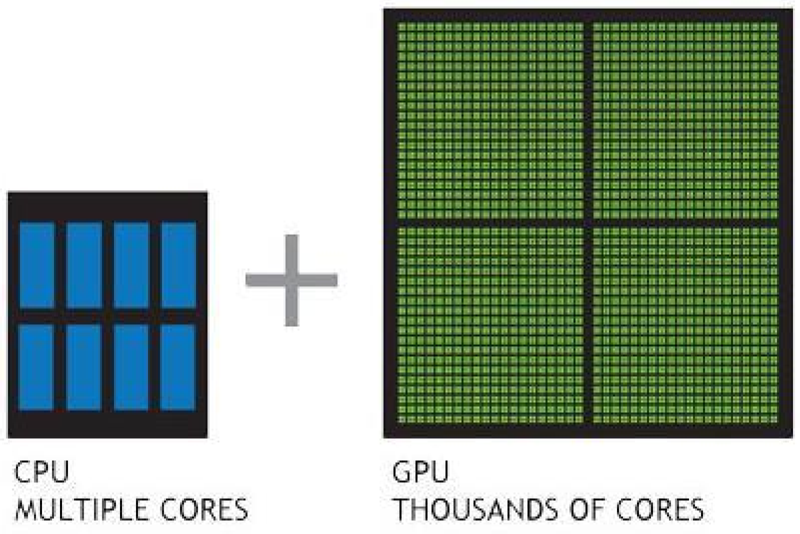

Figure 3.2: Comparing CPU and GPU Compute Cores [18]

In GPU acceleration, we transfer the parallel code to the GPU, and the sequential code is run on the CPU. Being the host, the CPU controls how part of the program is transferred to the GPU, and how the results are collected again upon completion of the program by GPUs.

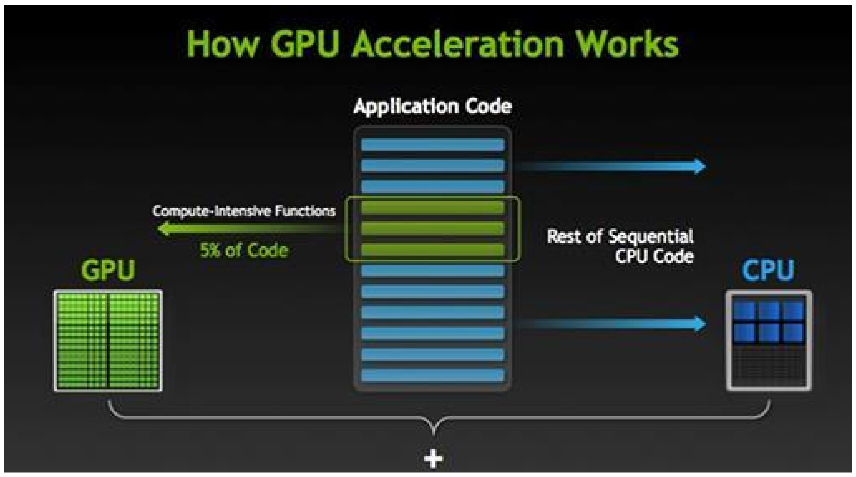
How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
Rest of Sequential CPU Code
GPU
CPU
+

Figure 3.3: How GPU Acceleration Works [18]

Per Moore's law, we have almost realized the highest performance, which can be drawn from the CPUs [15].

The new version of Moore's law of NVIDIA states, "As computers are not getting any faster they are just getting wider, parallel programming is here to stay". Because of this, data parallel computing is a more scalable solution for highly intensive applications [54].

These below graphs clearly indicate that parallel programming using GPUs will become the norm in the near future.

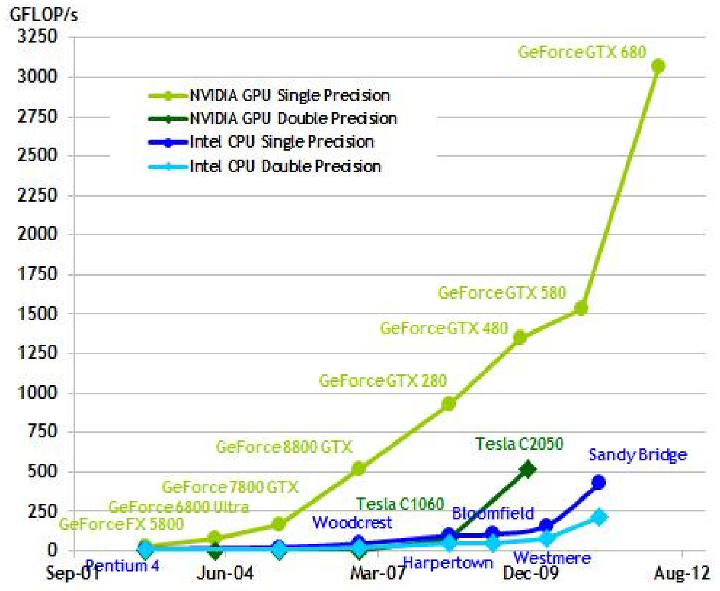

Figure 3.4: Floating-point Operations per Second for CPU and GPU [30]

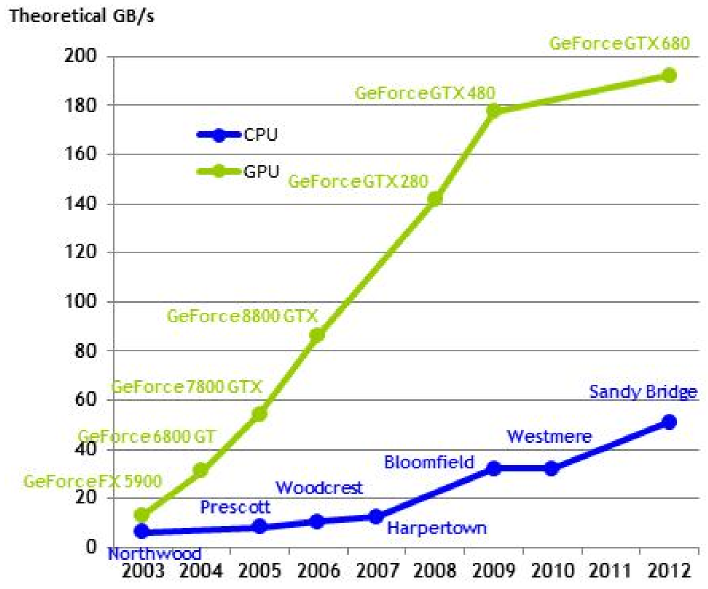

Figure 3.5: Memory Bandwidth for CPU and GPU [30]

3.2 NVIDIA GPUs

In recent years, the NVIDIA GPU has gained more popularity within the scientific computing fields because of language support provided for efficient programming and effective use of hardware.

3.2.1 NVIDIA GPU Acceleration

There are three ways to accelerate applications for NVIDIA GPUs as shown in the figure below:

Figure 3.6: Three Ways to Accelerate Applications for NVIDIA GPUs [19]

GPU – Accelerated Libraries: In this approach, we use libraries in order to accelerate the application. This method does not require any special knowledge of GPU architecture or programming language. As the libraries are designed to provide a standard API format, they are conducive to high-quality performance with minimal code change. Some of the GPU libraries include cuDNN (for high-level machine-learning frameworks), cuFFT

(Fast Fourier Transform Library), cuBLAS-XT (Basic Linear Algebra Subroutine), and MAGMA (next-generation linear algebra) [19] [20].

GPU Directives – OpenACC: OpenACC provides compiler directives for parallel programming to run the code on CPUs, GPUs, APUs, and coprocessors. This directive around the parallel region will help move the computation to the accelerator and execute the instructions on the accelerator [19] [21]. One of the benefits of using OpenACC is that we can improve the performance drastically by spending significantly less time on understating and writing the code. OpenACC as a parallel programming model will be discussed in the upcoming chapters of this document.

CUDA Language: CUDA C or C++ is a C / C++ interface to the parallel programming on accelerators. CUDA provides an extension to the C / C++ language, allowing the programs to execute on GPUs using GPU threads. In CUDA, we write kernels that are meant to run using the GPU thread model. These kernels are the computationally intensive loops that we transfer to the GPU. In the meantime, we also have to transfer data to and from the CPU to the GPU, which is required for computation. Thus, data transfer and kernel writing are two critical tasks that the CUDA user should understand to get the most from GPUs. CUDA is one of the low-level programming languages for GPUs compared to OpenACC and to use of GPU libraries. Thus, coding in CUDA has a learning curve that requires comparably more time to understand [19] [22]. We will discuss CUDA in a later section.

3.2.2 Kepler Architecture

Kepler is one of the fastest and architecturally most complex microprocessors, with as many as 7.1 billion transistors. One of the greatest improvements over previous GPUs like Fermi is Kepler's superior power efficiency mechanism. Kepler provides over 1Tflop of a double precision throughput. A Kepler GK110 implementation includes 15 SMX units and six 64-bit memory controllers [24] [25].

Key features of the Kepler architecture include: 1. New SMX processor architecture, 2. Enhanced memory subsystem, 3. Hardware support for new programming models [24].

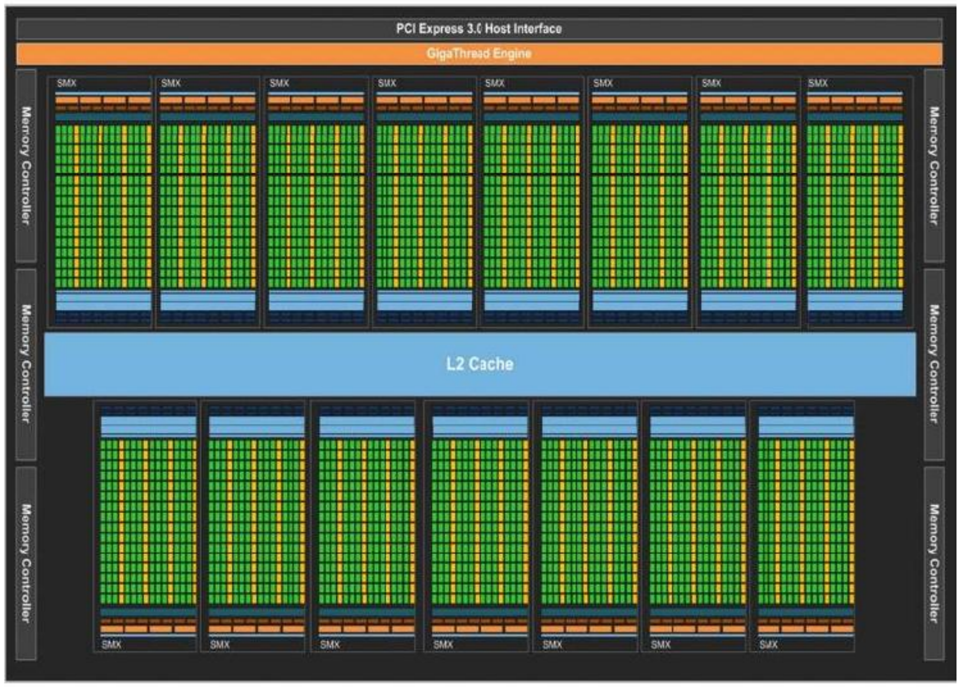

Figure 3.7: Kepler GK110 Full Chip Block Diagram [24]

3.2.2.1 Streaming Multiprocessor (SMX) Architecture:

Each streaming multiprocessor has 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

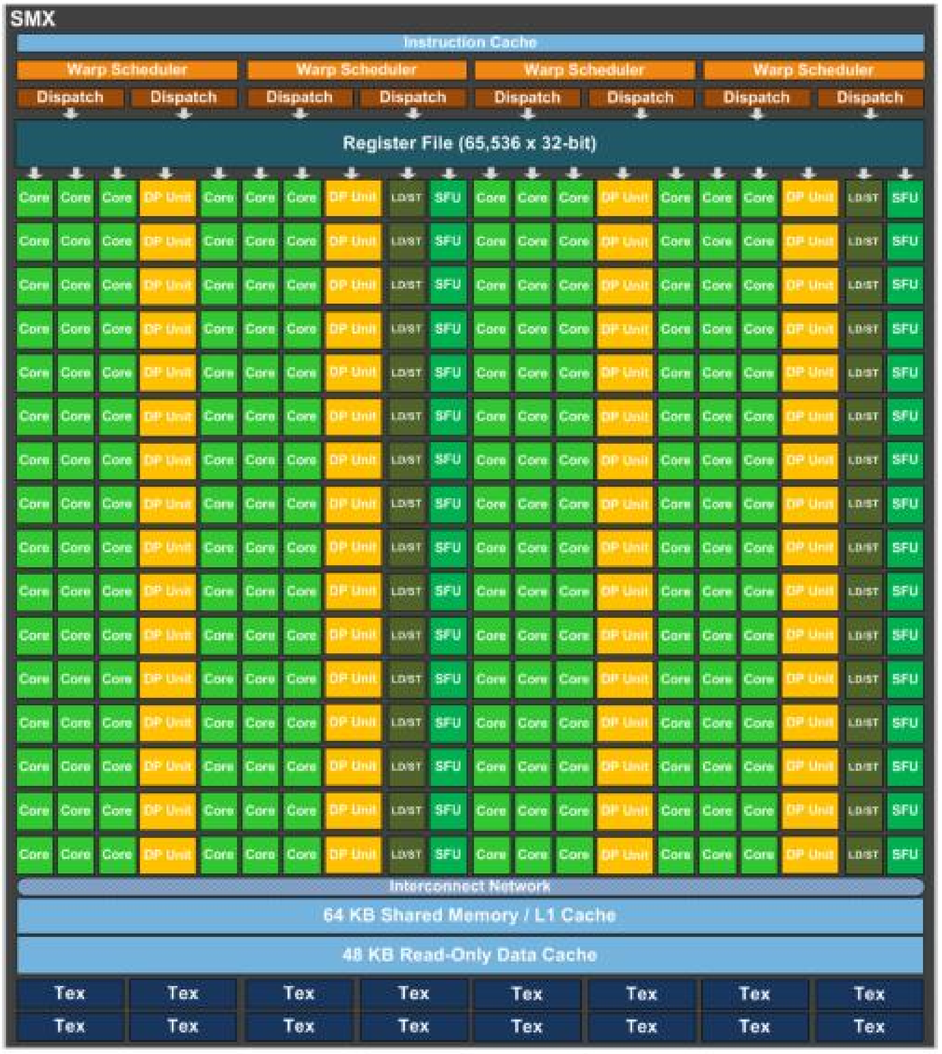
SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 3.8: Streaming Multiprocessor (SMX) Architecture [24]

Each SMX unit contains 192 single-precision CUDA cores, each of which has a pipelined floating point and integer arithmetic logic unit. Kepler has exhibited significant performance improvement as a result of double precision operations as they are at the heart of HPC applications. Each SMX schedule threads in a group of 32 parallel threads, which are called warps. Each SMX has four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Kepler's quad warp scheduler selects four warps, and two independent instructions per warp can be dispatched at every cycle. The number of threads that can be accessed by the thread has been improved 4X times compared to Fermi [24].

Kepler implements a new shuffle instruction, which allows threads within a warp to share data. Atomic operations are an integral part of any HPC application. Kepler has improved throughput of global memory atomic operations by expanding native support for 64-bit atomic operations in global memory [24].

3.2.2.2 Memory Subsystem

In Kepler GK110 every SMX has 64 KB of on-chip memory, which can be configured as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache. With Kepler, there is an additional flexibility in configuring the on-chip memory by permitting a 32KB/32KB split between the shared and L1 cache. In addition to the L1 cache, Kepler introduces a 48-KB cache for data that is known to be read-only for the duration of the function.

The Kepler GK110 GPU provides 1536 KB of dedicated L2 cache memory. The L2 cache is the primary point of data unification between the SMX units, servicing all load, store, and texture requests and providing efficient, high-speed data sharing across the GPU. The L2 cache on the Kepler offers up to twice the bandwidth per available clock in Fermi. Algorithms for which data addresses are not known beforehand, such as physics solvers, ray-tracing, and sparse matrix multiplication, especially benefit from the cache hierarchy. Filter and convolution kernels that require multiple SMs to read the same data also benefit [24].

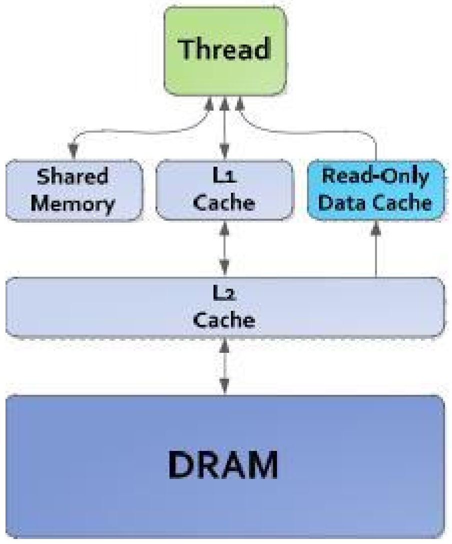

Figure 3.9: Kepler Memory Hierarchy [24]

Some of the new features in the Kepler GK110 are listed below. These features enable increased GPU utilization, making the parallel program design much simpler [24] [25].

Dynamic Parallelism - adds the capability for the GPU to generate new work for itself, synchronize results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU. This capability allows less structured, more complex tasks run easily and effectively, enabling larger portions of an application to run entirely on the GPU. In addition, programs are easier to create, and the CPU is freed for other tasks [24] [25].

Hyper-Q - enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and significantly reducing CPU idle times. Hyper‐Q increases the total number of connections (work queues) between the host and the GK110 GPU by allowing 32 simultaneous hardware‐managed connections. Hyper‐Q is a flexible solution that allows separate connections from multiple CUDA streams, from multiple Message Passing Interface (MPI) processes, or even from multiple threads within a process. Applications that previously encountered a false serialization across tasks, thereby limiting achieved GPU utilization, can realize a dramatic performance increase without changing any existing code [24][25].

Grid Management Unit – Enabling Dynamic Parallelism requires an advanced, flexible grid management and dispatch control system. The new GK110 Grid Management Unit (GMU) manages and prioritizes grids to be executed on the GPU. The GMU can pause the dispatch of new grids and queue pending and suspended grids until they are ready to execute, providing the flexibility to enable powerful runtimes, such as Dynamic Parallelism. The GMU ensures both CPU‐ and GPU‐generated workloads are properly managed and dispatched [24] [25].

3.3 AMD APUs

One of the biggest challenges in heterogeneous computing is memory management. In traditional architecture, CPUs and GPUs have different pools of memory with different hierarchies. Computation on a GPU involves movement of data from system memory to GPU memory and back from the GPU to CPU after completing the computation. One of the biggest challenges in heterogeneous computing, especially GPU computing, is the movement of data between two architecturally different systems. Sometimes this negates the advantage of computing on GPU completely and adds more overhead.

AMD has been very keen to resolve the problems associated with data movements in heterogeneous computing. To address this problem and also to address different problems related to heterogeneous computing, AMD developed the Heterogeneous Systems Architecture (HSA). This architecture involves changes to the hardware platform as well as to software runtime systems.

3.3.1 Heterogeneous System Architecture

While computing in a heterogeneous environment, some of the challenges faced by programmers are very crucial to address. The primary concern in today's world is reducing power consumption. Consumers expect more longevity in batteries, for example, for many handheld and portable devices. Even the data centers, which process and store huge amounts of data, need cooling, which increases the cost of overall maintenance. Secondly, there is always a demand for improving the performance of current technology. Consumers naturally want their devices to handle increasingly greater volumes of data with reasonable performance. Finally, the productivity of the programmer is an important factor in using evolving hardware and software technologies. It should be easy for new programmers to tap into the capabilities of new architectures without a considerable effort. One of the priorities of memory development should be the ability to run the same code on different platforms, targeting different devices without having to rewrite the code again and again for each specific platform [26].

HSA is one of the best comprehensive solutions for all the problems discussed. It promises improvement across all of the above parameters: power, performance, programmability, and portability [26].

The main goal of HSA design is to integrate the different processing elements tightly. HSA helps applications create and initialize data structures in a single unified address space to use the hardware effectively. The HSA model does not require fundamental changes to be implemented by software developers. Unified addressing used in HSA makes sharing of data between the CPU and GPU simple. The HSA architecture allows multiple compute tasks to work on the same coherent memory regions, using barriers and atomic memory operations, erasing the limitations associated with dissimilar memory systems [26].

The HSA Foundation aims to help system designers integrate different kinds of computing elements in a way that eliminates the inefficiencies of sharing data and sending work items between them. The HSA design allows multiple hardware solutions to be exposed to software through a common standard low-level interface layer, called the HSA Intermediate Language (HSAIL). HSAIL provides a single target for low-level software and tools. HSAIL is sufficiently flexible and yet low-level enough to allow each hardware vendor to map its individual underlying hardware design.. HSAIL frees the programmer from the burden of tailoring a program to a specific hardware platform – the same code runs on target systems with different CPU/GPU configurations [26].

A very important feature of HSA is its simplification of the process of running applications on the architecture. HSA brings hardware to the application programmer. The different layers of HSA include hardware, interfaces, standard runtime components, and common intermediate languages. These layers work together to maintain memory coherence and work queues synchronized [26].

HSA is all about delivering improved user experiences through advances in computing architectures by providing improved power efficiency, performance, programmability, and portability [26].

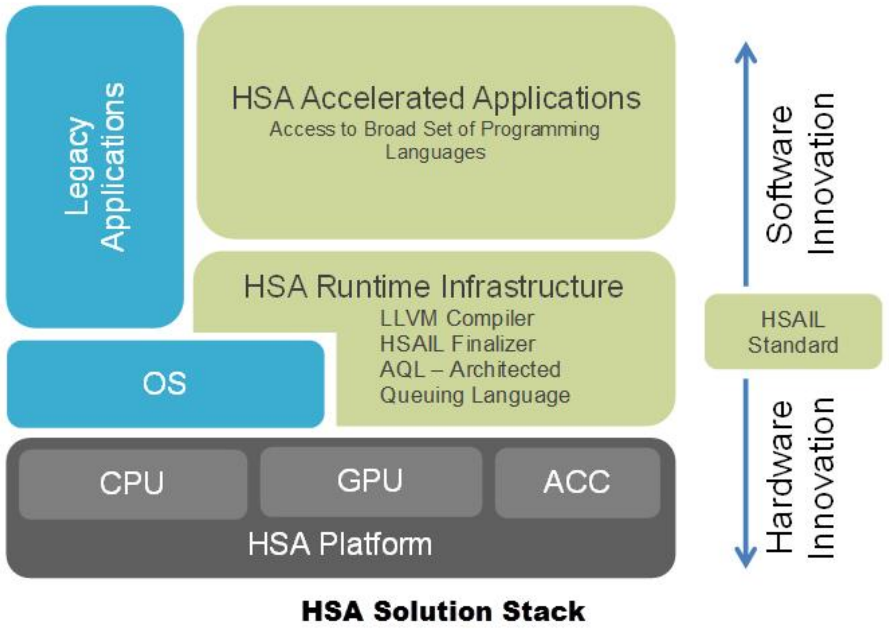

Figure 3.10: Heterogeneous System Architecture [26]

3.3.2 Kaveri Architecture

AMD’s most important leap towards the heterogeneous computing for HSA was heterogeneous Uniform Memory Access (hUMA). This architecture promises to solve this problem associated with the previous architecture. In hUMA memory space is cache-coherent which allows CPUs and GPUs use the same pointers to access the entire memory space. hUMA supports paged virtual memory, which makes it possible to work with larger data sets. This architecture allows the developer to write their applications using standard programming languages like Java, C++, and Python [28].

Kaveri is AMD's first processor to support hUMA. It has the new Steamroller CPU cores, combined with Radeon GCN (Graphics Core Next 2.0) architecture [29]. The main idea behind Kaveri is that current processors are too heavily CPU-biased. This technology utilizes the GPU core and draws on its maximum performance. Context switching between the CPU and GPU cores brings heavy overhead. Kaveri APUs are optimized to overcome this problem by increasing CPU frequencies, but these losses are buffered by a boost of almost 20% with new Steamroller cores [29].

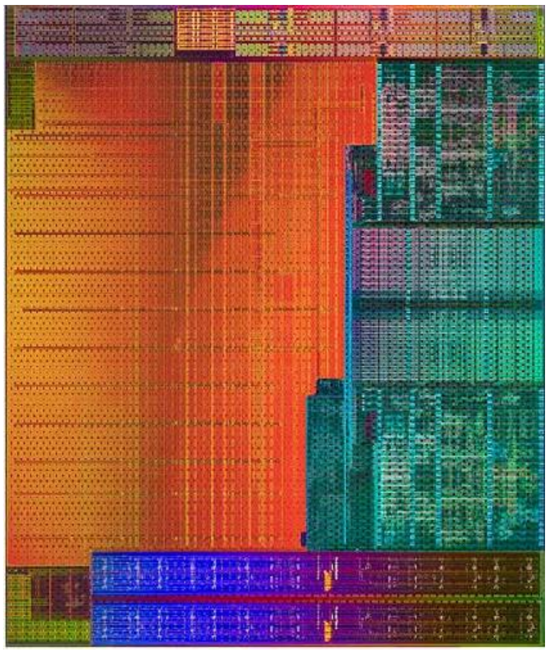
Figure 3.11: Kaveri APU [29]

In Kaveri architecture, all the cores, CPUs, and GPUs, are called compute cores. Each compute core acts as a programmable hardware block that can run processes in its context and virtual memory space. The architecture contains four multi-threaded Steamroller CPU cores and eight GCN-based Radeon GPU cores. Two very important technologies which are implemented in Kaveri and HSA for making the computing possible on the cores are hUMA and heterogeneous Queuing (hQ) [29].

In Kaveri, using the hUMA memory access pattern allows both CPU and GPU access the same memory. hUMA reduces almost all the issues associated with data movement and data management on two different memory systems, which enhances the performance of the programs [29].

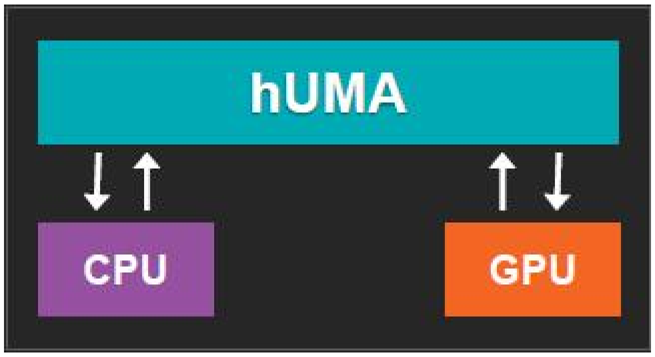

Figure 3.12: Block Diagram of Kaveri – hUMA [29]

hQ is a very important element in Kaveri, as it allows the GPU to send its tasks to the queue, which can then be dispatched to the GPU or CPU. In the previously discussed architecture, the GPUs did not have any role in an event or work dispatch. This new feature increases performance by eliminating bottlenecks associated with latency in processing, leading to less power consumption. AMD claims that as these features are implemented in the architecture, programmers no longer have to write a code specific to the GPU [29].

The new HSA architecture features support for OpenCL 2.0 as a programming standard.

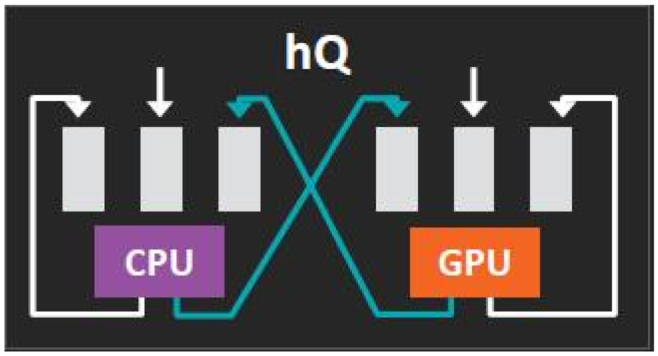

Figure 3.13: Kaveri hQ [29]

Chapter 4. Programming Models for Heterogeneous Systems

Parallel programming models are the basis for drawing extreme compute parallelism from heterogeneous systems. These programming models help map parallel applications to the hardware of the compute elements, execute applications on hundreds of these devices, and finally produce results with high performance and greater accuracy. Development of parallel programming models has become a recent priority, as we are using large heterogeneous compute elements for application processing more than ever.

How do we decide which is the best programming model for a given application? There is no "best" model, although there are certainly better implementations involving some models compared to others.

Different parallel programming models have been designed not only to target GPUs, but especially to target accelerators in common, are OpenMP, CUDA, OpenACC, OpenCL, and OpenGL. We will discuss how each of these models work in providing the best strategies for writing parallel code for accelerator architectures.

4.1 CUDA

NVIDIA introduced CUDA at the end of 2006 as a parallel computing platform and programming model for general processing units [30]. NVIDIA's GPUs use CUDA to solve many complex computational problems more efficiently than CPUs. CUDA provides a software environment for developers by allowing them to use C as a high-level programming language.

The three key abstractions provided by CUDA are a hierarchy of thread groups, shared memories, and barrier synchronization [30]. CUDA exposes all these features by providing a set of language extensions to C. These extensions allow CUDA to be a scalable programming language by running the programs on a number of multicores.

Programming Model:

CUDA provides fine-grained data parallelism and thread parallelism, which are nested within coarse-grained data and task parallelism. They also guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads. Each sub-problem is further divided into finer pieces, which can be solved cooperatively in parallel by all threads within the block. Some important elements of CUDA are discussed below.

Kernels are the main computational units that execute the code section on the GPU using CUDA C extensions. Kernels are executed N times by N CUDA threads. A kernel is defined using the __global__ declaration specifier, and the number of CUDA threads that execute the kernel is specified using a new <<<…>>> execution configuration syntax. Each thread that executes the kernel is given a unique *threadID* that is accessible within the kernel through the built-in *threadIdx* variable [30].

Thread Hierarchy is one of the important properties for CUDA programming. It specifies the level of parallelism applied to the loop. threadIdx is a three-vector component so that threads can be identified using a one-, two-, or three-dimensional thread index, forming a one-, two-, or three-dimensional thread block, respectively. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume [30].

In the following example, we add two matrices A and B of size NxN and store the result in matrix C:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{  int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{   ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...}
```

Table 4.1: Code Snippet for Matrix Multiplication using CUDA [30]

There is an upper limit on the number of threads that can be spawned per block, and these threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, the kernel can be executed by multiple equally shaped thread blocks so that the total number of threads is equal to the number of threads per block times the number of blocks [30]. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system.

The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type *int* or *dim3* [30]. Two-dimensional blocks or grids can be specified as shown in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in *blockDim* variable [30].

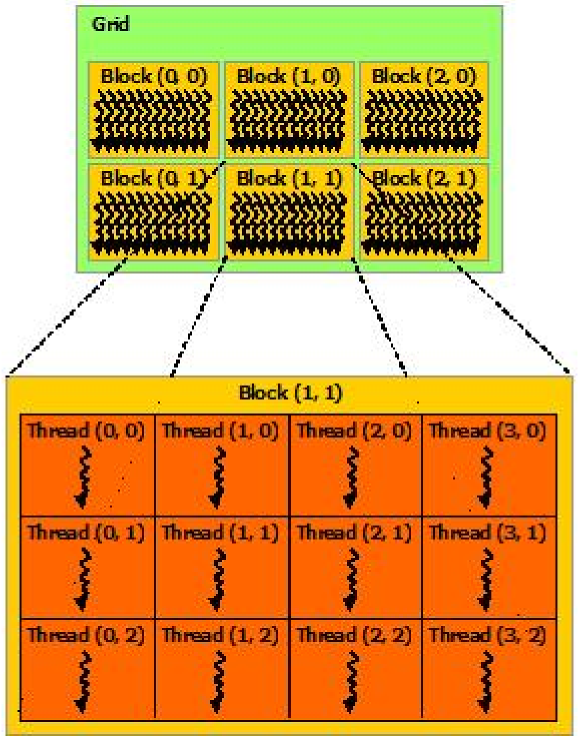

Figure 4.1: Grid of Thread Blocks [30]

Thread blocks are required to be executed independently: It must be possible to execute them in any order, in parallel or series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores. Threads within a block can cooperate by sharing data through shared memory and by synchronizing their execution to coordinate memory accesses. *__syncthreads()* acts as a barrier at which all threads in the block must wait before any of them is allowed to proceed. For efficiency, the shared memory is expected to be low-latency, and *__syncthreads ()* is expected to be lightweight [30].

CUDA threads can access data from multiple memory spaces during their execution as shown in the figure below. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All of these created threads have access to the same global memory [30].

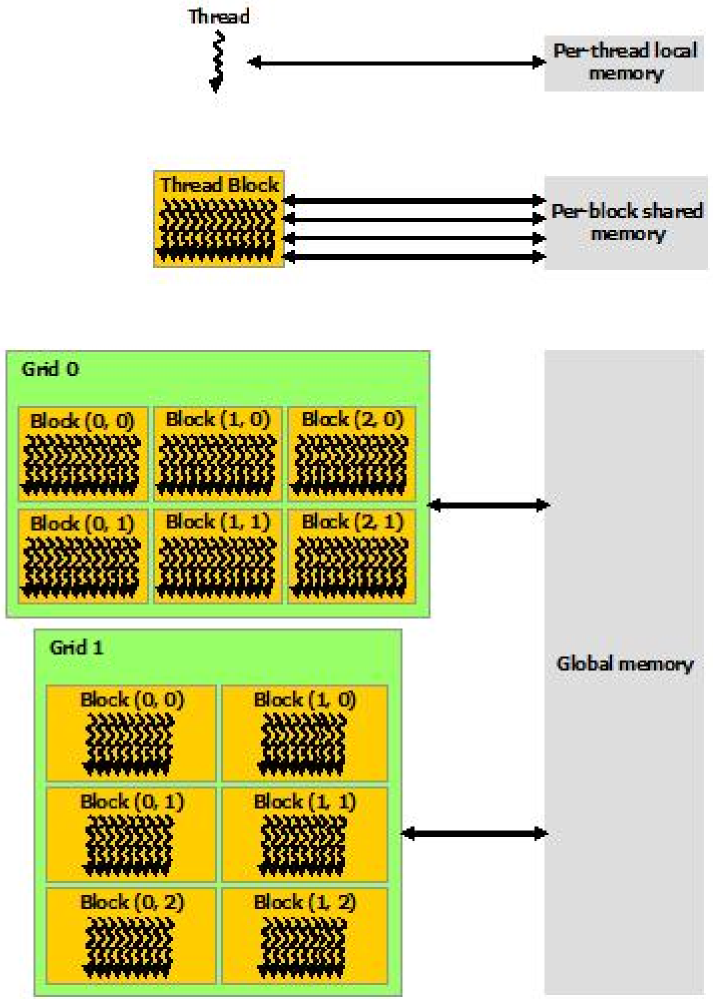

Figure 4.2: Memory Hierarchy [30]

Heterogeneous Programming

The CUDA programming model assumes that the CUDA threads execute on a separate device connected to the main CPU via interconnect. This host and device execution model will have a different memory space, which needs to be allocated, deallocated, and synchronized.

4.2 OpenACC

OpenACC is a high-level directive-based programming model for accelerators that enables scientific and technical C, C++, and Fortran programmers to easily take advantage of the power of heterogeneous CPU/accelerator systems. This model provides a set of pragmas and high-level APIs for the non-native programmer hiding all the low-level details of the programming language compared to CUDA. OpenACC provides set of pragmas which serve as hints to the compiler regarding how different portions of the code should be run.

OpenACC allows programmers to use simple compiler directives to identify which areas of code to accelerate, without requiring modification to the underlying code itself. By identifying parallel code segments, OpenACC directives allow the compiler to perform the detailed work of mapping the computation on the accelerator.

Understanding the memory model and architecture of the systems will help the programmer to achieve better performance, as it helps to map the program to the hardware more efficiently.

OpenACC Basic Syntax: OpenACC supports C/C++ and FORTRAN languages for directive-based programming. Table 4.2 gives the syntax for C/C++ and FORTRAN.

Syntax for C/C++	Syntax for Fortran
#pragma acc directive [clause [,] clause] Often followed by structured code block	*!$ acc directive [clause [,] clause]* Often followed by structured code block Paired with the matching end directive *!$ acc end directive*

Table 4.2: OpenACC Pragma's Syntax

The directives and clauses in italics can be replaced by appropriate implementations depending on the structure of the code. We can have only one directive on each line of code, which can be followed by more than one clause.

Compute Constructs of OpenACC: OpenACC API has two compute constructs namely 1. Kernels and 2. Parallel construct. Both constructs have the same goal, but they are used in different contexts.

1. Kernels Construct: When the kernels construct is applied to the loop nest, it is converted to parallel kernels by compiler to run efficiently on the GPU. This is a three-step process. The first step is to identify a parallelizable loop. The second is to map this abstract loop parallelism on to the concrete hardware parallelism. For NVIDIA GPU, this means mapping a parallel loop onto a grid-level or thread-level parallelism. In OpenACC, gang level parallelism is mapped to grid-level parallelism and vector parallelism mapped to thread-level parallelism. In the third step, the compiler has to generate and optimize the block of code to implement the selected parallelism. Hence, the kernels directive uses classical automatic parallelization to identify and parallelize the loops.

```
#pragma acc kernels
{
   for( i = 0; i < n; ++i )
    a[i] = b[i] + c[i];
}
```

Table 4.3: Code Snippet for Sample Kernel Constructs

2. Parallel Construct: OpenACC parallel construct tries to solve the same problem as that of the kernels directive. The only difference between the two is that the kernels construct is implicit, giving more freedom to the compiler to analyze the loop, find parallelism, and then map parallelism to the hardware threads. But in the case of parallel constructs, the compile command is more explicit. Whenever we use the parallel construct, it is the programmer's responsibility to analyze the loop and determine when it is legal and appropriate to parallelize the loop.

```
#pragma acc parallel
{
   #pragma acc loop
   for( i = 0; i < n; ++i )
      a[i] = b[i] + c[i];
}
```

Table 4.4: Code Snippet for Sample Parallel Constructs

Loop clause: An important observation from Table 4.4 is the use of the loop clause. Loop is a work-sharing construct. When we do not use the loop clause with the kernels/parallel directive, all the threads execute the code redundantly.

4.3 OpenMP

OpenMP is a high-level directive-based programming API for shared memory multiprocessor architecture, which uses incremental changes to the existing program to parallelize it. It includes a set of compiler directives/pragmas, library functions, and environment variables. OpenMP is sufficiently expressive and efficient for many applications. The directives used for parallelizing serve as compiler commands for C, C++, and FORTRAN languages. The main aim of the OpenMP model is improving the productivity of a programmer by accelerating the performance of the sequential code to get comparatively better performance [31]. A better understanding of the programming model and architecture of the target platform will help in implementing the right kind of optimization techniques to achieve better performance.

4.3.1 Memory Model of OpenMP

As indicated above, OpenMP is a programming model for shared memory multicore architecture.

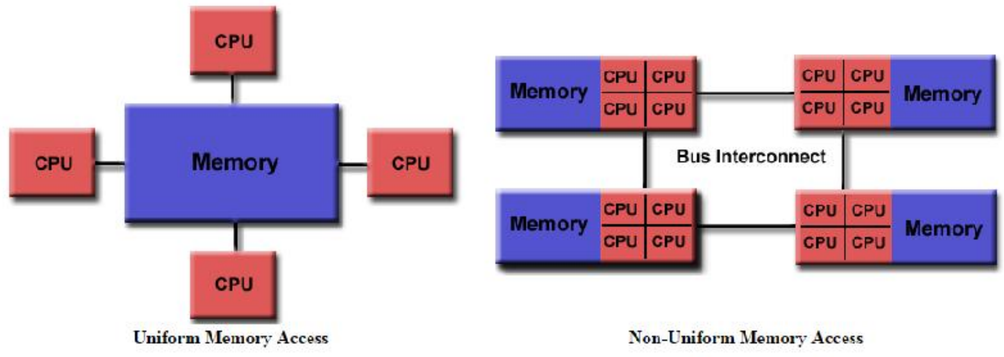

Figure 4.3: Memory Model for OpenMP [31]

In shared memory programming, each node is called a symmetric multi-processor (SMP), and these SMPs access memory simultaneously in order to provide communications among the processing elements. Compared to different heterogeneous memory subsystems, programming a shared memory system is relatively easy. This is because all SMPs share a single view of data and the communication between processors can be as fast as memory accesses to the same location. We can have systems with two different types of shared memory architecture: 1. Uniform memory access (UMA) as shown in Figure 4.3, where all SPMs share the same memory and the memory access time is the same for all cores. 2. Non-uniform memory access (NUMA) as shown in Figure 4.3 memory access time depends on memory system from which the processor is requesting the data [31].

4.3.2 Execution Model of OpenMP

OpenMP uses the fork-join model for parallel execution. As shown in the figure below, the execution of the program starts with a single thread, which is the main/master thread created by the program. When the master thread encounters the parallel construct, it creates a team of threads to perform the computation in parallel. The number of threads created by the master thread depends on the value assigned to the environment variable OMP_NUM_THREADS or the number of threads provided, which is denoted as the routine *omp_set_num_threads (number_threads)* [31].

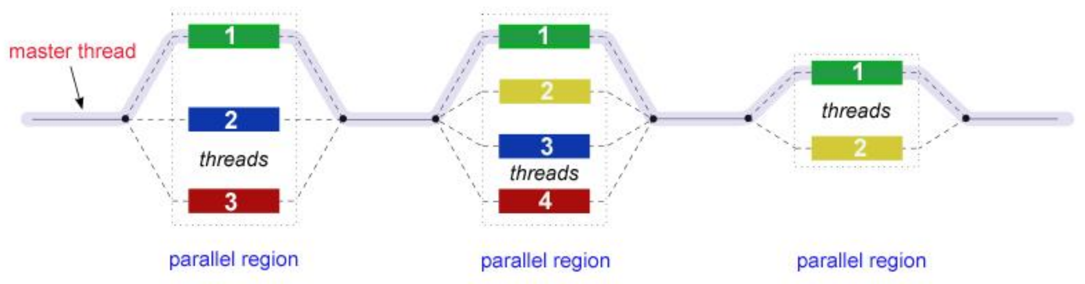

Figure 4.4: Fork-join OpenMP Execution Model [48]

4.3.3 OpenMP Programming Directives

OpenMP provides the compiler directives that can be inserted into the existing serial program. We have a slightly different syntax for using OpenMP pragmas in C/C++ and FORTRAN as shown in the Table 4.5 below.

C/C++ directive format	`#pragma omp parallel [clause ...]  newline`
	`if (scalar_expression)`
	`private (list)`
	`shared (list)`
	`default (shared | none)`
	`firstprivate (list)`
	`reduction (operator: list)`
	`copyin (list)`
	`num_threads (integer-expression)`
	
	`structured_block`

Fortran	`!$OMP PARALLEL [clause ...]`
	`IF (scalar_logical_expression)`
	`PRIVATE (list)`
	`SHARED (list)`
	`DEFAULT (PRIVATE | FIRSTPRIVATE |`
	`SHARED | NONE)`
	`FIRSTPRIVATE (list)`
	`REDUCTION (operator: list)`
	`COPYIN (list)`
	`NUM_THREADS (scalar-integer-expression)`
	
	`structured_block`
	
	`!$OMP END PARALLEL`

Table 4.5: OpenMP Directive Usage

Some important sets of directives for OpenMP API are described briefly in the following sections. The list is not exhaustive.

Parallel Region Construct:

Parallel region construct is one of the most important parts of the OpenMP programming, as this construct is responsible for creating the threads that are, in turn, responsible for parallel programming. In the program, a single thread is executed until it meets the PARALLEL construct. Once the program encounters the PARALLEL construct, it creates a pool of threads, and all these threads execute the upcoming block of code in parallel. Every thread executes each line of code in parallel until they encounter the implicit barrier at the end of the parallel clause [31]. After this point, the master thread takes over the execution as shown below.

```
int nthreads, tid;

/*  Fork,  a  team  of  threads  with  each  thread  having  a  private  tid
variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */
```

Table 4.6: Sample OpenMP Code Snippet

Work-sharing Constructs:

Work-sharing constructs are very important, as they help in dividing work among all the threads, which are created using the PARALLEL construct. Work-sharing constructs do not create more or new threads. If we do not have the work-sharing constructs, and if we have only parallel constructs for a block, multiple threads will be created, but every thread executes the entire block of code. This will degradation the performance because of the overhead associated with the construct itself. We have different types of work-sharing constructs. Do/for constructs allow data parallelism by dividing the loop iterations among the threads. Sections' construct allows functional parallelism, as they allow different pieces of code to be executed by a single thread. The single construct is used to serialize the block or section of code; that is when we want some code to be executed by a single thread, we use a single construct [31].

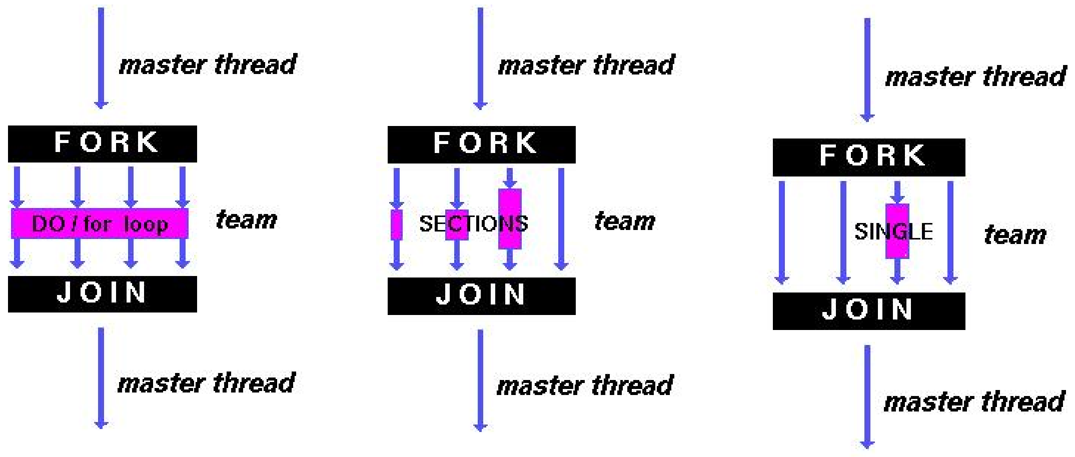

Figure 4.5: Work-sharing Constructs [31]

Nested Parallelism in OpenMP:

OpenMP allows nested parallelism. When a thread executing in a team encounters one more parallel construct, the executing thread creates a group of threads and acts as a master thread. This feature allows parallelizing recursive algorithms in a natural way [2].

Task Parallelism:

The TASK construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team. The data environment of the task is determined by the data-sharing attribute clauses. Task execution is subject to task scheduling. For more information on task scheduling and clauses, see the OpenMP 3.1 specification document [31].

Fortran

```
!$OMP TASK [clause ...]
             IF (scalar logical expression)
             FINAL (scalar logical expression)
             UNTIED
             DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
             MERGEABLE
             PRIVATE (list)
             FIRSTPRIVATE (list)
             SHARED (list)

   block

!$OMP END TASK
```

C/C++

```
#pragma omp task [clause ...]  newline
                   if (scalar expression)
                   final (scalar expression)
                   untied
                   default (shared | none)
                   mergeable
                   private (list)
                   firstprivate (list)
                   shared (list)

     structured_block
```

Table 4.7: OpenMP Task Clauses

Synchronization Constructs:

Synchronization is a kind of inter-process communication that is required to maintain order and data dependencies in the program. PARALLEL has implicit synchronization at the end and in the beginning. DO, MASTER, and SINGLE constructs have implicit synchronization at the end of the construct. OpenMP provides a rich set of synchronization constructs like ATOMIC (to perform atomic operations), and FLUSH (provides point-to-point synchronization) [31].

Data Scope Attributes:

Defining the scope of the variables is performed explicitly in OpenMP in order to ensure the correct results. As OpenMP is a shared memory programming model and many of the threads work on the same data, some variables change, and some should retain their values. Some variables should be the same for all threads, but some will have an updated value. All these conditions are taken care in OpenMP using the data scoping clauses explicitly. These overwrite the scoping provided by the programming environment / language. Data scoping is provided by PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, DEFAULT, REDUCTION, and COPYIN constructs. These are used in conjunction with the PARALLEL constructs to define the data scope for the entire block [31

4.4 OpenCL

OpenCL is a programming language that helps in writing the programs for heterogeneous platforms. OpenCL routines are called *kernels*, which are executed on accelerators. OpenCL can be used to program, NVIDIA GPUs as well as AMD APUs. One of the biggest disadvantages of OpenCL is it has a significant learning curve compared to other heterogeneous programming languages.

Host Application Development:

The first step in OpenCL programming is writing a host application. This program runs on the host and dispatches kernels to the coprocessor. This application can be programmed using C or C++, and every host application requires five data structures from OpenCL: cl_device_id, cl_kernel, cl_program, cl_command_queue, and cl_context [33].

A host application distributes kernels to devices, and this kernel is represented by a cl_kernel. The device receives kernels from the host represented by a cl_device_id. The host selects kernels from a program, which is represented by a cl_program. Each device receives kernels through a command queue. In code, a command queue is represented by a cl_command_queue [33] [32]. The OpenCL context allows devices to receive kernels and transfer data. In code, a context is represented by a cl_context.

Chapter 5. Developing an Accelerator Benchmark Suite

The main goal of developing an accelerator benchmark is to evaluate an OpenACC model for heterogeneous computing. This benchmark also helps to understand how different clauses implemented in new OpenACC2.0 can be used to improve performance compared to OpenACC 1.0, CUDA, and OpenMP.

We have different programming models and standards like OpenCL, CUDA, and OpenMP, all of which support accelerator programming. OpenACC intends to achieve better portability across different accelerator architectures.

OpenACC code interoperates well with the broader ecosystem of libraries and parallel programming languages for accelerated development. Developers may choose vast majority of their application development with OpenACC.

We compare the performances of the benchmarks created using an accelerator benchmark suite with already existing accelerator models like CUDA and OpenMP.

Portability is of major concern while programming heterogeneous systems. Although there are low-level languages, such as CUDA, which is widely used to program GPUs, it is certainly not a portable approach. The main advantage of using CUDA is that it has more control over GPU architecture and offers programmers efficient programming techniques. Using OpenACC for accelerators is a viable approach while addressing the portability factor. In this work, we have certainly achieved portability by using OpenACC, but not the same performance as with a CUDA program. For example, stencil code shows that the performance of the OpenACC code is ~77% of that of CUDA code. It may be acceptable to trade off higher productivity for lower performance as long as the performance of the high-level approach is not too low.

The OpenACC model is inspired by OpenMP; both are high-level programming model approaches. OpenACC as of today is more mature for accelerator programming compared to OpenMP. This helps in exploiting the massive capabilities of GPUs. Although the hardware used for OpenMP and OpenACC are significantly different from each other, comparative analysis between both models helps to understand how an OpenMP-like model, i.e. OpenACC, can be used efficiently for accelerators. The OpenMP 4.0 version, which was released recently, also adds support for GPU accelerators [42]. This comparison will help us in understanding how well OpenMP supports programming for GPU architecture.

5.1 Heterogeneous Benchmark Suite

There are various accelerator benchmarks created to compare and analyze the performance of applications with respect to runtime, power consumption, etc. We select the benchmarks, depending on the aim with which they are created and their usefulness with the given architecture.

5.1.1 Overview

On using OpenACC for GPU programming, "*Let me start by stating that OpenACC does not make GPU programming easy. You will hear some vendors or presenters saying that*

OpenACC or some other language, library or framework makes GPU programming or makes parallel programming easy. Do not believe any of these statements. GPU programming and parallel programming is not easy. It cannot be made easy. However, GPU programming need not be difficult, and certainly can be made straightforward, once you know how to program and know enough about the GPU architecture to optimize your algorithms and data structures to make effective use of the GPU for computing. OpenACC is designed to fill that role." - Michael Wolfe, The Portland Group [35].

Background work involves understanding a different set of benchmarks, target architecture and intent of developing. Some benchmarks are listed in the section below.

SHOC Benchmark: The Scalable Heterogeneous Computing Benchmark Suite (SHOC) is a collection of benchmark programs that test the performance and stability of systems using computing devices with non-traditional architectures for general-purpose computing and the software used to program them. Its initial focus is on systems containing Graphics Processing Units (GPUs) and multi-core processors and on the OpenCL programming standard. It can be used on clusters as well as individual hosts [36].

In addition to OpenCL-based benchmark programs, SHOC also includes a Compute Unified Device Architecture (CUDA) version of many of its benchmarks for comparison with the OpenCL version. Multiple benchmark applications are written in both OpenCL and CUDA. SHOC implements cluster-level parallelism with MPI. For multiple GPUs per node, SHOC provides node-level parallelism [37].

The SHOC benchmark suite is divided into two primary categories: stress tests and performance tests. The stress tests use computationally demanding kernels to identify OpenCL devices with bad memory, insufficient cooling, or other component problems [37]. The performance tests are further subdivided according to their complexity and the nature of the device capability they exercise.

Rodinia Benchmark: The Rodinia benchmark is published by the University of Virginia. A researcher desiring a set of GPU computing source codes to experiment with, but who does not need any kind of uniform scripting environment for code multi-versioning, collecting performance, results, or changing platform settings across all benchmarks, may find the Rodinia benchmarks to be a better fit [34]. The Rodinia benchmarks have no compilation complexity other than standard source code and makefiles to understand. The support in Parboil provides easy solutions for conducting certain kinds of research experiments at the cost of a slightly more complex compilation and execution system. Rodinia is similar in philosophy and development of applications compared to the Parboil benchmarks. They contain a similar mixture of building-block kernels and applications, and also support both CUDA and OpenCL for most benchmarks [38].

5.2 Using OpenACC 2.0

OpenACC is an emerging high-level model available for GPU programming. It gained a lot of popularity in a very short period of time. We have some of the benchmarks that have implemented OpenACC versions. Porting applications using OpenACC1.0 are complicated, as they lack some key features. As OpenACC 1.0 did not have support for separate compilation units, it also lacked support for function calls as separate compilation units [14].

The rationale behind the release of OpenACC 2.0 was to enhance the productivity of programmers with more tools to enable them to write code and exploit the underlying GPU architecture.

Adding separate compilation unit is one of the most important features of OpenACC2.0. Function calls, which are very important elements of the programming language, are missing from the OpenACC 1.0 spec, because of the unavailability of the linkers to some targets. Thus, we needed to inline all the function calls in OpenACC 1.0 [41].

5.2.1 Routine Directive

Procedure calls and separate compilations are two of the most important features of any high-level programming language, as they allow modularity and also help in adding libraries and other components. OpenACC 1.0 did not support function calls from inside the parallel or the kernels region, as it lacked linker support. Thus, programmers must manually inline the function calls in OpenACC 1.0. This was one of the main limitations of OpenACC 1.0 [35] [43].

OpenACC 2.0 introduces a new routine directive that enables calls to the routines to be made from within the compute region. This allows the compiler to build a host and device version of the function separately, allowing the device version to be called from the compute region. Using the routine directive, we can also specify the level of parallelism that should be applied to the called routine by inserting one of the gang, worker, vector, or seq clauses [43].

```
#pragma acc routine worker extern
void matvec(float *v, float *x ,
float *a, int i,int n);

#pragam acc parallel loop gang
for(int i=0;i<n;i++)
{
  v[i]+=rhs[i];
matvec(v,x,a,i,n);
//procedure call on the device
//in OpenACC 1.0 inlines the function
call
}
```

```
#pragma acc routine worker
void matvec(float *v, float *x ,
float *a,int i, int n)
{
     float xx=0;
    #pragma acc loop reduction(+:x)
     for(int j=0;j<n;++j)
           xx+=a[i*n+j]*v[j];

      x[i]=xx;

}
```

Table 5.1: Usage of Routine Directive [35]

5.2.2 Nested OpenACC Regions

Nested parallelism, which is introduced in OpenACC 2.0, allows kernel launches from within the kernels.

Dynamic parallelism and function calls complement each other. In the sense, when we have function calls on the GPUs, it is most likely that the function that is called will also have structured loops that can be parallelized. If dynamic parallelism were not supported, then function calls would not have been very useful.

```
#pragma acc routine worker extern
void matvec(float *v, float *x ,
float *a, int i,int n);

main{
..
#pragam acc parallel loop
for(int i=0;i<n;i++)
matvec(v,x,a,i,n);
..
}
```

```
#pragma acc routine void matvec(float
*v, float *x , float *a,
 int i, int n)
{
 …
    #pragma acc parallel loop
     for(int j=0;j<n;++j){..}
}
```

Table 5.2: Nested Parallel Regions

The goal of the routine calls and nested parallelism is to move as much code to the accelerator as possible. This allows decoupling of the device code from the host and reducing the data movements as much as possible.

5.2.3 Unstructured Data Regions

OpenACC 1.0 fostered structured data lifetimes, which are not always very practical because of the structure of the code. OpenACC 2.0 allows unstructured data lifetime for variables by implementing two new constructs: “acc enter” and “acc exit” clauses. Enter and exit data region clauses provide flexibility on how long the data can stay on the device or moved back to the host only when it is finally done. This will further reduce some of the unwanted data movements compared to the structured data movement and help overcome the bottlenecks associated with data transfer on the PCI bus. In C++ programming, we can relate the use of unstructured data with the use of constructors and destructors, as data is created in one function and deleted in order. Thus, we can keep the data on the device longer. If there is no “exit data”, then the lifetime of the variable continues until the end of the program [43].

```
#pragma acc enter data
copyin(a[0:n]) create(b[0:n])
..
..
//some blocks of code, need not be
structured block

#pragma acc exit data delete(a[0:n])
...
#pragma acc exit copyout
delete(b[0:n])
```

```
class Matrix {
  Matrix(int n) {
  len = n;
  v = new double[len];
#pragma acc enter data create(v[0:len])
  }
  ~Matrix() {
#pragma acc exit data delete(v[0:len])
 delete[] v;
  }

  private:
    double* v;
    int len;
};
```

Table 5.3: Unstructured Data Lifetime [35]

In OpenACC 1.0, the lifetime of the data is only defined between the structured blocks. With the unstructured data region, the lifetime of the data region is not controlled by the set of curly braces, but rather we can define the data lifetime of the variables depending on the program flow [44]. There are three types of unstructured data regions. The first one is enter and exit data constructs, which can be added anywhere in the code, but are normally added where data is created and deleted in the program respectively. Secondly, “acc_malloc” is used to allocate data on the device and “acc_free” to free data from the device. Data allocated using acc_malloc is only available on the device when no host copy exists. To indicate that only a device copy is available, we use a device pointer, which is done using "acc declare" pragma; for example: acc declare deviceptr (result).

OpenACC 2.0 also allows the creation of global variables. The need for global variables comes from the fact that OpenACC 2.0 allows the separate compilation of subprograms. We create global variables using the "Declare" directive, and these variables are shared among multiple kernels. We have different versions of this directive: 1. *Declare create* - creates static data for both the host and device. Both copies should be synchronized using the update directive. 2. *Declare device_resident* - creates data only on the device and the memory is not allocated to this variable on the host. Thus, this variable can be used only on the device for computation. For instance, if we are using some arrays that contribute to the computation as input elements, then it's better to make these variables device-resident [35] [45].

5.2.4 Atomic Directive

The OpenACC atomic directive ensures that a particular variable is accessed and/or updated atomically (no two gangs, workers, or vector threads access the location simultaneously) to prevent indeterminate results and race conditions. In other words, it prevents one thread from encroaching on other threads due to accessing a variable simultaneously, resulting in different results run-to-run [45].

```
void main() {
      ….

    if ( val and condition=true )
    {
        #pragma acc atomic capture
        Number_of_waves++;
    }
}
```
Table 5.4: Atomic Construct

Without the atomic directive, it is not possible for multiple threads to increment the value of Number_of_waves simultaneously, which will likely result in an incorrect final count. The atomic directive ensures that no two threads access Number_of_waves at the same time.

5.2.5 Tile Construct

Loop tiling is one of the compiler optimizations that partition the loop of the larger multidimensional array into smaller chunks. The aim of creating blocks of data is to ensure that data stays in the cache for loop iteration. This enhances cache reuse. Normally, the tile clause is used along with the cache clause.

`for(i=0; i<N;`
`++i)`
`{`
`  ...`
`//blocks of code`
`}`
	`for(j=0; j<N; j+=S)`
	`  for(i=j; i<min(N,`
	`j+S); ++i){`
	`    ....`
	`// S – is the tile size`
	`  //blocks of code`
	`  }`
		`#pragma acc parallel loop`
		`private(i,j) tile(8,8)`
		`for(i=0; i<rows; i++)`
		`    {`
		`     for(j=0; j<cols; j++)`
		`      {`
		`        out[i*rows + j] =`
		`in[j*cols + i];`
		`      }`
		`    }`

Table 5.5: Explaining Tile Clause Code Snippet

5.2.6 Support for Multiple Device Types

OpenACC 2.0 implements the device_type clause, and the compiler can direct the part of the code to run on a particular device that is a parameter to the clause. We can tune the program, according to the particular device architecture by using these directives [43].

```
#pragma acc parallel loop \
            device_type(NVIDIA) vector_length(256) \
            device_type(radeon) vector_length(512) \
                                vector_length(64)
for ( int i=0; i<n; ++i)
{
      sum[i] = a[i] + b[i];
}
```

Table 5.6: Syntax for Using Multiple Device Types [43]

5.2.7 Extended Runtime API

A complete set of runtime APIs added to OpenAC2.0 are shown in the table.

acc_wait_async	Enqueues wait operations on one async queue for the operations previously enqueued on another async queue.
acc_copyin, acc_present_or_copyin	Allocates memory on the accelerator device to correspond to the specified host memory, and copies the host data to that device memory if the data is not already present on the device.
acc_create, acc_present_or_create	Allocates memory on the accelerator device to correspond to the specified host memory if the data is not already present on the device.
acc_copyout	Copies data from the device memory back to host memory and deallocates the device memory.
acc_delete	Deallocates the device memory corresponding to the specified host memory.
acc_map_data, acc_unmap_data	Maps and unmaps previously allocated device data to the specified host data, respectively.
acc_deviceptr	Returns the device pointer associated with a specific host address.
acc_hostptr	Returns the host pointer associated with a specific device address.
acc_is_present	Tests whether a host variable or the array region is present on the device.
acc_memcpy_to_device, acc_memcpy_from_device	Routine copies data to and from local memory to device memory, respectively
acc_update_device, acc_update_self	Copies data from device memory to the corresponding host memory.
acc_wait, acc_wait_all instead of acc_async_wait and acc_async_wait_all	Wait routine waits for completion of all associated asynchronous operations.

Table 5.7: New OpenACC 2.0 Clauses and Directives for Data Management [45]

OpenACC 2.0 includes some of the run-time library routines to enhance data management. Acc_deviceptr and acc_hostptr are used to access the device and host copies of the variables that are allocated using the declare directive. acc_copyin is used to allocate memory on the device and then copy the data, which is specified in the clause to allocate device memory. The acc_create clause creates memory on the device, which is similar to using "acc data enter" with the create clause [43].

5.3 Testbed Setup

Features of the testbed used for measuring the performance of the benchmarks are as listed:

GPU details: NVIDIA Tesla K20c (GK110GL), with width of PCI bus: 64-bit

CPU details: Intel (R) Xeon (R) E5520@ 2.27GHz, 2 sockets, 4 cores per socket, two hyper threads per core

Operating system: Linux 2.6.18-398.el5, 64-bit

Compiler: PGI 14.9

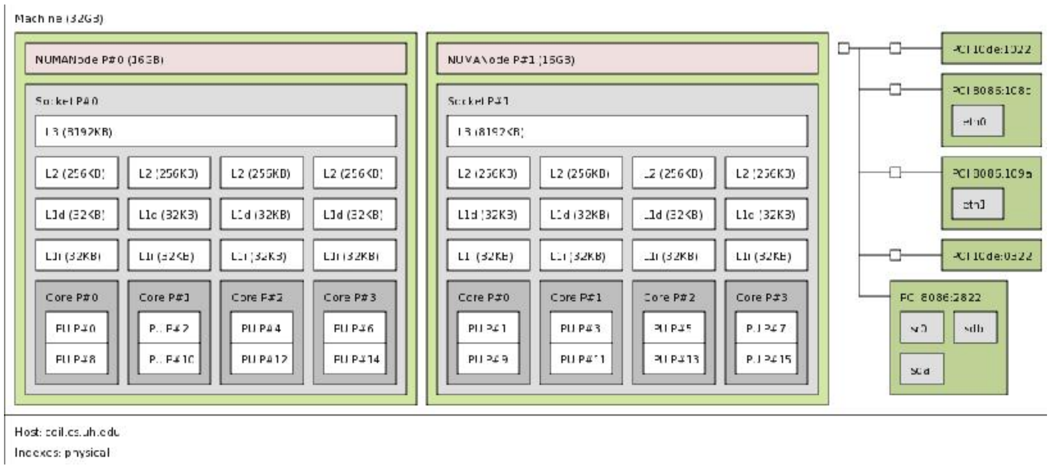

Figure 5.1: Architecture of the Coil System

5.4 Parboil OpenACC 2.0

The Parboil benchmark has multiple scientific and commercial applications including image processing, bimolecular simulation, fluid dynamics, and astronomy, which are written in C and C++ applications. As of today, this benchmark has been used by many researchers to implement the latest versions of the applications for different programming architectures and programming languages, permitting a performance comparison over the other already present accelerator versions. Parboil has been implemented for Serial, OpenMP-base, OpenCL-base, CUDA-base, CUDA-Fermi, and CUDA-generic in the latest release, which targets multicore CPUs and GPUs [34].

The goal of the Parboil benchmark is to: 1) Represent the emerging scientific and commercial applications of throughput computing; 2) Represent application algorithms exhibiting low algorithmic complexity; 3) Scale to very wide parallel architecture, anticipating the potential for exponential growth in machine width and choosing algorithms that will remain applicable for several years; and 4) Provide application implementations and a benchmarking infrastructure that will support the work of various research interests, at minimum encompassing architecture, microarchitecture, compiler, language, and programming environment research [34].

The benchmarks below are ported to the GPU architecture using the OpenACC programming model. We have also used advanced compute directives and constructs provided by OpenACC 2.0, which enhances the portability and also to get better performance results for the benchmark.

5.4.1 Benchmark Porting and Result Analysis

Stencil: Stencil computations are fundamental to many of the large applications in scientific computing. Stencil computations are used for solving partial differential equations, which are the building blocks for solving the Jacobi kernel, the Gauss–Seidel method for image processing, etc. The computationally intensive nature of a stencil algorithm makes it the best candidate for accelerator computing.

Parboil uses stencil code to represent the iterative Jacobi solver of the heat equation on a 3D structured grid [34]. In stencil code, we perform sequential iterations through a given array. In every iteration array elements are updated with the new values using neighboring array elements with the specified pattern. The pattern in which new values are computed is repeated for the entire computation, hence the highly parallel nature of the code. Computations in the stencil are tightly coupled to neighboring points, making it memory-bound and, thus, increasing computing time [48].

In the code snippet, line 1 is where the OpenACC kernel is placed, which has a parallel computation. When the compiler encounters a “pragma acc kernels” keyword, it launches multiple threads that perform the following computation in parallel. On the same line, we also have copy construct with A0 [0:nx*ny*nz] and Anext[0:nx*ny*nz] attributes. This construct is shorthand for present_copy, which first confirms if the variables are already present on the GPU for computation. If yes, then the data that are already on the GPU is used for the computation, ignoring the copy command. And if the data are not present on the GPU, then the compiler copies the data to the GPU, which will be followed by the computation in the loop. One important thing to observe is while performing computations on the GPU, the data are moved back and forth between the host and GPU, which definitely degrades the performance.

```
1     #pragma acc kernels pcopy(A0[0:nx*ny*nz], Anext[0:nx*ny*nz])
2     {
3           #pragma acc loop independent vector
4           for(i=1;i<nx-1;i++)
5           {
6               #pragma acc loop independent gang vector
7                   for(j=1;j<ny-1;j++)
8                   {
9                      #pragma acc loop independent gang vector
10                          for(k=1;k<nz-1;k++)
11                          {
12                              Anext[Index3D (nx, ny, i, j, k)] =
                                (A0[Index3D (nx, ny, i, j, k + 1)]
13  +
14                              A0[Index3D (nx, ny, i, j, k - 1)] +
15                              A0[Index3D (nx, ny, i, j + 1, k)] +
16                              A0[Index3D (nx, ny, i, j - 1, k)] +
17                              A0[Index3D (nx, ny, i + 1, j, k)] +
18                            A0[Index3D (nx, ny, i - 1, j, k)])*c1
19                             - A0[Index3D (nx, ny, i, j, k)]*c0;
20                          }
21                  }
22          }
```
Table 5.8: Stencil Code Snippet

One of the best optimization techniques that programmers can implement for accelerator codes is to place a data construct to cover the larger scope of the program. This is important when we have more than one kernel and all the data computation is predominantly performed on the main array for the program, on the arrays used as input, and on some arrays that are constantly updated. On lines 3, 6, and 9, we have an acc loop clause; these are the work-sharing constructs. If we do not place these clauses on these lines, then all the threads end up performing all the computations. The consequence of this is much worse when compared to serial code. This is because we are creating more overhead for the program to run with threads, but without distributing the work across threads.

On lines 3, 6, and 9, we also have an independent clause, which informs the compiler explicitly to consider the array elements as independent with respect to the previous iterations. If we do not specify this clause, then the compiler warns the user by raising an error flag, stating that a loop carried dependency exists between the loop iterations.

On lines 3, 6, and 9, we have vector, gang vector, and gang vector clauses. By using these clauses, we tell the compiler explicitly which level of parallelism should be applied for the loops that follow. Specifying these, we can achieve fine-grained parallelism.

Stencil gives the best performance with OpenACC with respect to CUDA, which exhibit nearly identical results. For the default problem size [total number of elements: 6777216 (nx = 512 : ny = 512 : nz = 64)], CUDA and OpenACC exhibit good performance gains over OpenMP. This indicates that as the problem size increases, we can see better performance in CUDA and OpenACC as they are best suited for large sets of data that are highly parallel in nature. The performance gain of OpenACC with respect to OpenMP for the default size is attributed to the fine-grained parallelism applied to the main computational loop of the stencil program.

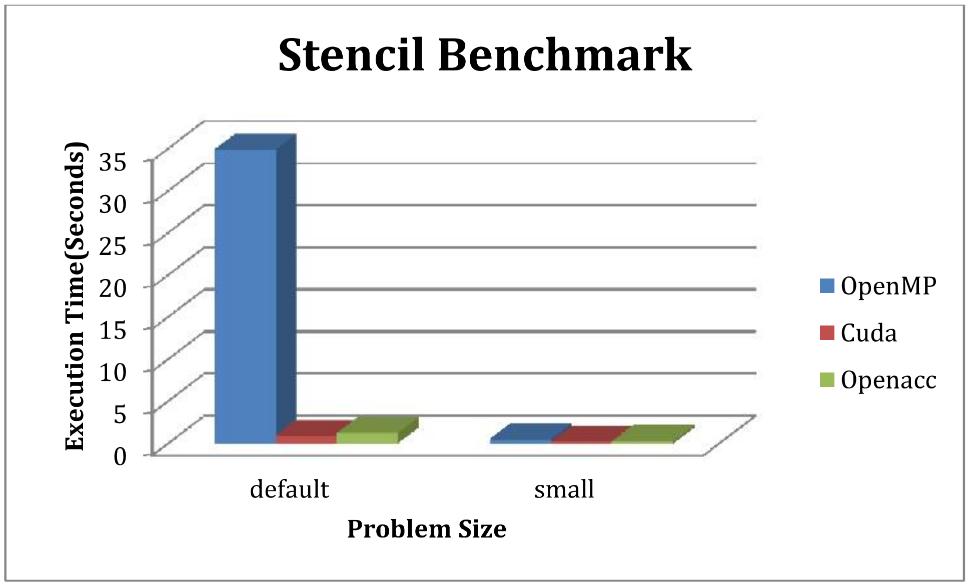

Figure 5.2: Execution Time Graph – Stencil

Histogram (histo): The histogram input set follows a Gaussian distribution in the input. The histogram benchmark is a straightforward histogramming operation that accumulates the number of occurrences of each output value in the input data set. The output histogram is a two-dimensional matrix of char type that saturates at 255. The dimensions of the histogram (256 W × 8192 H) are very large, yet the input set follows a roughly Gaussian distribution, centered in the output histogram. Recognizing this high concentration of contributions to the histogram's central region, the benchmark optimizations mainly focus on improving the throughput of contributions to this area [34].

For creating an output of histogram operations, we need to check if each and every pixel value is less than 255. If yes, then we have to update the internal value associated with the image. Updating the histo value is atomic, as only one thread should be updating this value at any given time. Thus, we use the "acc atomic update" feature provided by OpenACC 2.0. This feature was not available in OpenACC 1.0. Atomic is one of the most important features in thread programming, as it allows the programmer to have a lock on the variable so that only one thread can access the variable at any given point. We use atomic update on line 7, which locks the histo array to be accessed by only one thread.

```
 1  #pragma acc parallel loop
 2  for (i = 0; i < img_width*img_height; ++i)
 3  {
 4     const unsigned int value = img[i];
 5     if (histo[value] < 255)
 6     {
 7
 8        #pragma acc atomic write
 9        {
10             histo[value]=histo[value]+1 ;
11        }
12     }
13  }
```

Table 5.9: Use of Atomic Update in Histogram

Atomic is one of the most important implementations provided by OpenACC 2.0. Without this directive, we would not be able to parallelize any part of the computation.

We can also use the new data management features "acc enter data" and "acc exit data" provided by OpenACC 2.0. We move data to the GPU once data are allocated on the host and then free the data just before the data is completely utilized. This is one of the most powerful features provided by the new OpenACC implementation because it allows the porting processes to follow the natural program flow.

```
unsigned int* img = (unsigned int*) malloc
(img_width*img_height*sizeof(unsigned int));
unsigned char* histo = (unsigned char*) calloc
(histo_width*histo_height, sizeof(unsigned char));

#pragma acc enter data
create(histo[0:histo_width*histo_height],img[0:img_width*img_height])
```
```
pb_SwitchToSubTimer(&timers, outputStr, pb_TimerID_IO);
#pragma acc exit data copyout(histo[0:histo_width*histo_height])
delete(img)
```

Table 5.10: Unstructured Data Lifetime Feature

The histogram benchmark localizes atomic operation of the main loop while calculating values for the output histogram. The OpenMP version does not run for large problem size, as it fails to allocate enough memory. The execution time of OpenACC is greater when compared to the CUDA version. This is because of the atomic region. CUDA, which is a low-level language, has more control of the architecture, maximizing the performance of the architecture. In OpenACC, atomic regions are not as mature as CUDA. When we have an atomic region inside the main loop, which has parallelism, it makes the part of the code sequential. This adds overhead and, hence, increases the runtime of the program.

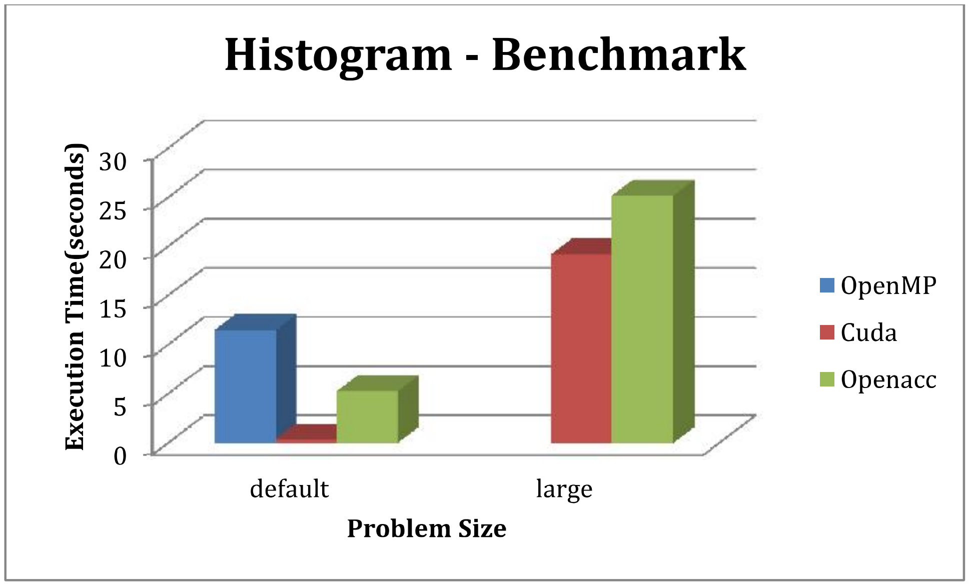

Figure 5.3: Execution Time Graph - Histogram

Cutoff-limited Coulombic Potential (CUTCP): Some molecular modeling tasks require a high-resolution map of the electrostatic potential field produced by charging atoms distributed throughout a volume [17]. Cutoff-limited Coulombic Potential (CUTCP) computes a short-range component of this map, in which the potential at a given point comes only from atoms within a cutoff radius of 12 A [34].

This program contains the definition and usage of struct elements, which are structure for atom, lattice, and vec3. OpenACC 1.0 versions did not support computation of complex structure elements, but the newer version facilitates the use of complex structures. In this application, we traverse the grid, which is spatially distributed, and then find the closest grid point with a position less than or equal to the atom. Next, we identify the extent of the surrounding box of grid points, trim the box edges so that they are within the grid point lattice, and then finally loop over the surrounding grid points.

```
#pragma acc parallel data copy(next[0:size1], first[0:ncell])
#pragma acc loop gang, vector private(atom)
```

Table 5.11: Code Snippet for CUTCP

We use acc parallel to start a kernel, and then use loop gang to divide the computation among the gang of threads. We make the atom as private so that every thread can have its own copy of the atom and process it independently, as sharing an atom among the threads will lead to inconsistent results. Analyzing these features in the programs is very important to ensure accuracy of results while performing the optimization techniques.

The optimized CUTCP application is compute-bound. Unlike the other compute-bound benchmarks, this kernel achieves high computational throughput partly at the cost of performing redundant computation.

The percentage of redundant computation is the primary performance limiter for the hardware configuration. The reasonable performance of an Opencc version of the CutCP application for large problem size is due to the fine-grained level of parallelism.

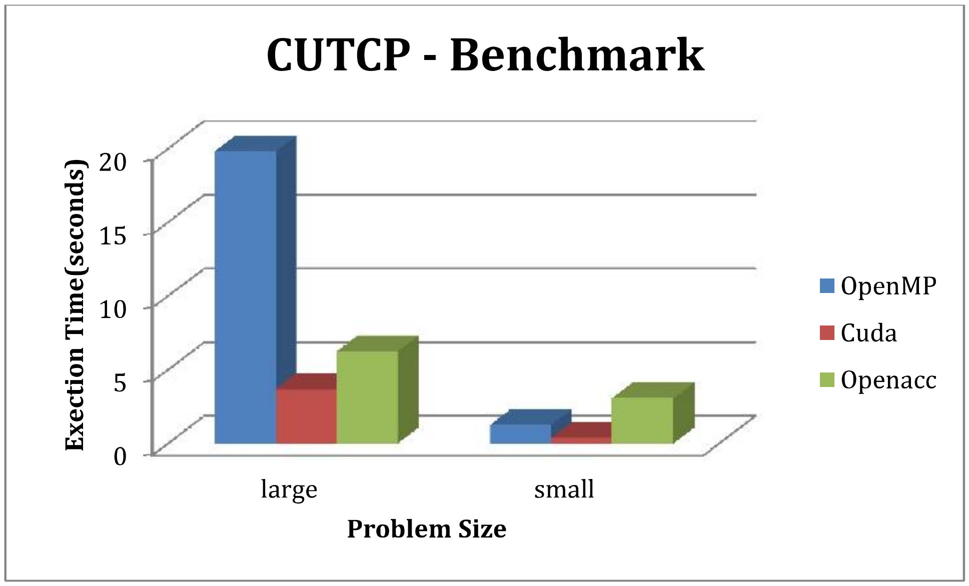

Figure 5.4: Execution Time Graph – CUTCP

Sparse Matrix-dense Vector Multiplication (SpMV): Sparse matrix-vector multiplication is the core of many iterative solvers. SpMV is memory-bandwidth-bound when the matrix is large. Thus, most optimization efforts have focused on improving the memory bandwidth for both regular and irregular access. In this program, we store sparse matrix data in the Jagged Diagonal Storage (JDS) format. JDS works with parallelism of finer granularity more easily than others. Accessing data in the JDS format naturally results in stride-one access to the sparse matrix elements. Preaching is also applied to hide more memory latency when high thread-level parallelism is not sufficiently available to hide latency alone [34].

SpMV computes the product of a sparse matrix with a dense vector. The sparse matrix is read from files in a coordinate format, converted to the JDS format with configurable padding, and translated for distribution to different devices. As the problem size increases, OpenACC clearly delivers the best performance among all three languages.

```
#pragma acc data copyin(h_nzcnt,h_perm,h_ptr,h_indices,h_data,h_x_vector),
copy(h_Ax_vector)
{
	#pragma acc loop gang, vector
	for(p=0;p<50;p++)
	{
		 #pragma acc loop gang, vector
		for (i = 0; i < dim; i++)
		{
			sum = 0.0f;
			bound = h_nzcnt[i];
			 #pragma acc loop seq
			for(k=0;k<bound;k++ )
			{
				j = h_ptr[k] + i;
				in = h_indices[j];

				d = h_data[j];
				t = h_x_vector[in];

				sum += d*t;
			}

			h_Ax_vector[h_perm[i]] = sum;
		}
	}
}
```

Table 5.12: Code Snippet for SPMV Compute Loop

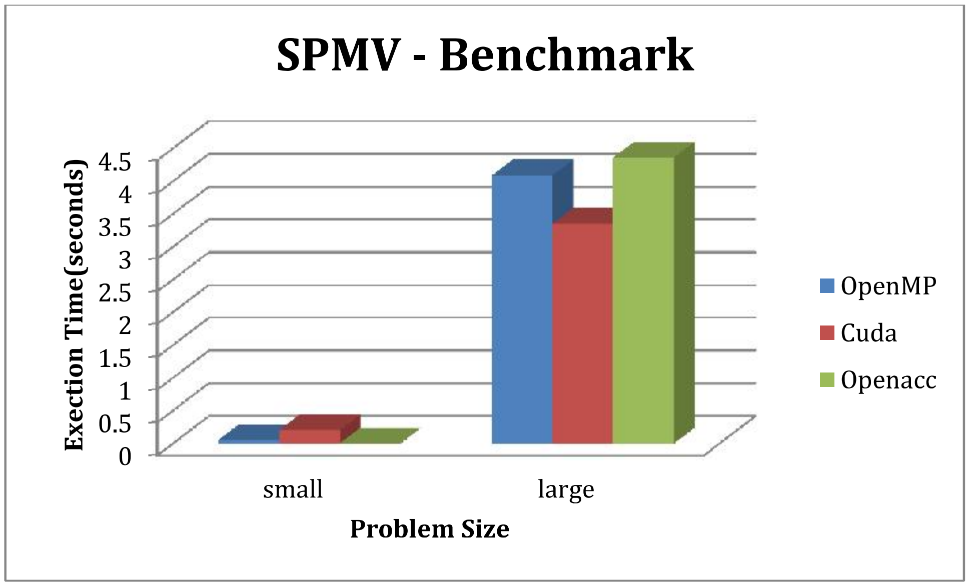

Figure 5.5: Execution Time Graph – SPMV

Lattice-Boltzman Method Simulation (LBM): The LBM is a method of solving the systems of partial differential equations governing fluid dynamics [49]. Its implementations typically represent a cell in a lattice with 20 words of data: 18 represent fluid flows through the six faces and 12 edges of the lattice cell, one represents the density of fluid within the cell, and one represents cell type or other properties. In a timestep, each cell uses the input flows to compute the resulting output flows from that cell and an updated local fluid density [34].

The major difference between LBM and a stencil application is that no input data is shared between cells; the fluid flowing into a cell is not read by any other cell. Therefore, the application has been memory-bandwidth-bound in current studies, and optimization efforts have focused on improving achieved memory bandwidth [34].

Functions	Computation performed
LBM_allocateGrid	Allocates grid points, which will be used for simulation.
LBM_freeGrid	Deallocate the memory after the computation is done.
LBM_initializeGrid	Set the grid with initial values like position, velocity, and time.
LBM_swapGrids	Swaps the grids to initiate a simulation.
LBM_loadObstacleFile	This file reads the collision objects in the grid space to create a scenario for elastic collisions.
LBM_initializeSpecialCellsForLDC	Initializes special cells for accelerators.
loadValue	Loads the present state of each grid element.
LBM_initializeSpecialCellsForChannel	Sets the IN_OUT_FLOW for the channel to be true or false by calculating the position of the grid elements and obstacles.
LBM_performStreamCollide	Creates a destination grid, which is created after the collision on the source grid by applying multiple sweeps against obstacles.
LBM_handleInOutFlow	This function is very important in LBM. This handles the simulation by handling the rate at which we have influx and outflux of the particles.
LBM_showGridStatistics	Gives statistics for grid molecules like position, mass, velocity, obstacle cells, and fluid cells.
storeValue	Stores the present state of each grid element.
LBM_storeVelocityField	Every time there is a collision, the velocity of the grid molecule changes. This function stores the new velocity.
LBM_compareVelocityField	Compares velocity of molecule before and after collision.

Table 5.13: LBM Functions

GPU version of the LBM benchmark uses the most logical layout for software engineering: a large array of cell structures. Data layout transformation results in an optimized GPU version using a tiled structure-of-arrays [34].

The LBM simulation is a problem in the fluid dynamics simulation for an enclosed, lid-driven cavity. This is one of the standalone applications with an execution time of 170 seconds in OpenMP for long problem size. This program is heavily nested, requiring fine-grained parallelism for better performance. We achieve reasonable performance in LBM as we applied fine-grained parallelism to the innermost loop in the LBM_initializeSpecialCellsForChannel and LBM_performStreamCollide functions.

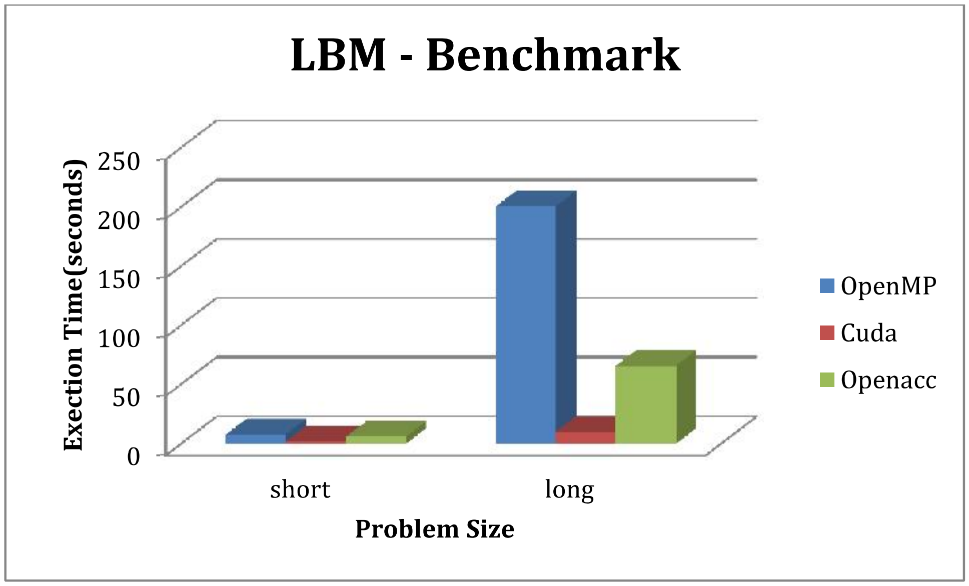

Figure 5.6: Execution Time Graph – LBM

MRI non-Cartesian Q matrix calculation (MRI-Q): One of the original Parboil benchmarks, MRI-Q, computes equation 3 in the GPU-based MRI reconstruction paper by Stone et al. [50], and is based on the implementation used to publish their work. The algorithm examines a large input data set representing the intended MRI-scanning trajectory and the points that will be sampled. Each element of the Q matrix is computed by a summation of contributions from all trajectory sample points. Each contribution involves a three-element vector dot product of the input and output 3D location and a few trigonometric operations. The output Q elements are complex numbers, but the inputs are multi-element vectors. An output element (and its corresponding input denoting its 3D location) is assigned to a single thread. To ensure that the thread-private data structures exhibit good coalescing, a structure-of-arrays layout was chosen for the complex values and physical positions for a thread's output.

The shared input data set, however, is cached using GPU constant memory or some other high-bandwidth resource, and elects an array-of-structures implementation to keep each structure in a single cache line [34].

```
#pragma acc parallel loop
for (indexK = 0; indexK < numK; indexK++) {
  for (indexX = 0; indexX < numX; indexX++) {
    expArg = PIx2 * (kVals[indexK].Kx * x[indexX] +
                     kVals[indexK].Ky * y[indexX] +
                     kVals[indexK].Kz * z[indexX]);

    cosArg = cosf(expArg);
    sinArg = sinf(expArg);

    float phi = kVals[indexK].PhiMag;
    Qr[indexX] += phi * cosArg;
    Qi[indexX] += phi * sinArg;
  }
}
```

Table 5.14: Code Snippet for ComputeQ CPU Function

MRI-Q is a fundamentally compute-bound application, as trigonometric functions are expensive and the regularity of the problem allows for easy management of bandwidth [34]. Therefore, once tiling and data layout removes any artificial bandwidth bottleneck, the most important optimizations where the low-level sequential code optimizations improving the instruction stream efficiency by applying loop unrolling.

The MRI-Q application computes a matrix Q, representing the scanner configuration for calibration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space. This benchmark uses complicated structure data types as program variables that are not supported in OpenACC 1.0. As seen in the code snippet above, calling the trigonometric functions from inside the parallel region increases overhead somewhat.

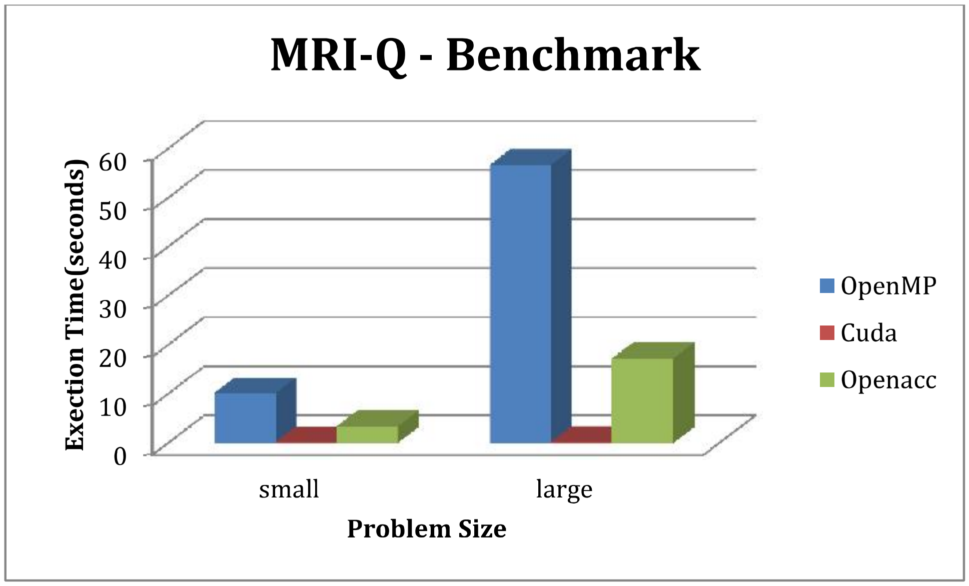

Figure 5.7: Execution Time Graph – MRI-Q

SGEMM: SGEMM is a dense matrix multiplication application, which is a building block of many linear algebra equation systems. Some of the popular implementations of this algorithm from major vendors are CUBLAS (from CUDA), MKL, and ACML. We use the C++ version of the application's inside-out code to implement the OpenACC version. This application optimizes the input matrix into a vector array; this provides the first level of very important optimization. Matrices are two-dimensional or multidimensional in nature. When we use matrices, especially when the matrices are dense and are very large, accessing elements will pose the biggest challenge and result in a bottleneck, thus reducing the performance of the program.

We read column major matrix rows and columns and then convert them into vector elements in the application. This helps improve the data locality and cache performance significantly because of their access pattern. These features improve computational performance as the size increases.

```
  int matArow, matAcol;
  int matBrow, matBcol;
  std::vector<float> matA, matBT;

  /* Read command line. Expect 3 inputs: A, B and B^T
     in column-major layout*/

  // load A
  readColMajorMatrixFile(params->inpFiles[0],
      matArow, matAcol, matA);

  // load B^T
  readColMajorMatrixFile(params->inpFiles[2],
      matBcol, matBrow, matBT);

// allocate space for C
  std::vector<float> matC(matArow*matBcol);
```

Table 5.15: Code Snippet for Vectorising Column Major Matrixes to Vector Array

The code snippet shows how the input matrices are converted into vectors, and then these vectors are used in GPUs for performing computations.

Sgemm contains one parallel OpenACC kernel – Sgemm_kernel, which is called from the main function. Below is the code snippet showing how the computation is parallelized using OpenACC.

```
#pragma acc data copyin(A[(m*k)],B[(k*n)]) copy(C[(m*n)])
copyin(alpha,lda,ldb,beta,ldc)
{
      #pragma acc parallel loop gang, vector (8)
      for (int mm = 0; mm < m; ++mm)
      {
         #pragma acc loop gang, vector (8)
         for (int nn = 0; nn < n; ++nn)
         {
            c = 0.0f;
            #pragma acc loop  seq
            for (int i = 0; i < k; ++i)
            {

                 a = A[mm]+I * lda];
                 b = B[nn + i * ldb];
                 c += a * b;
         }
         C[mm+nn*ldc] = C[mm+nn*ldc] * beta + alpha * c;
      }
}
```

Table 5.16: Code Snippet for SGEMM – sgemm_kernel

The first step in parallelizing any loop using OpenACC is taking care of data movements. In the program, vectors A and B are input elements that move to the GPU space for computation, and C is the vector array where we store the results. The important thing to observe here is we use copying (A,B) and copy (C), so A and B are copied to the GPU, and once we complete the computation, vector C is copied as a result array back to the host. Thus, after exiting the parallel compute region, A and B are deleted from GPU memory, making space for the new kernels to be launched later in time. The data constructs help maintain the data for the entire loop iteration time, which reduces the overhead involved in moving data back and forth for every iteration, which is the case in the absence of the data clause.

Further down the function, we apply acc parallel on the outermost loop, creating the threads needed for the parallel region. We split the computation of the outermost loop into gangs and vectors of length. The second inner loop also spreads the computation across the gang and vector, and finally the acc loop seq executes the innermost loop using a sequence of threads. We have applied a fine-grained level of parallelism to the compute kernel. Applying fine-grained parallelism is architecture- and compiler-dependent. Thus, when we change the architectures, we should change the level of parallelism and determine for every change how the performance varies. Implementing different levels of parallelism help understand how different architectures react to specific levels of parallelism.

One of the most widely and intensely studied benchmarks, this application performs a dense matrix multiplication using the standard BLAS format. We apply loop unrolling for realizing the best performance. Moreover, usage of efficient data management constructs optimizes results.

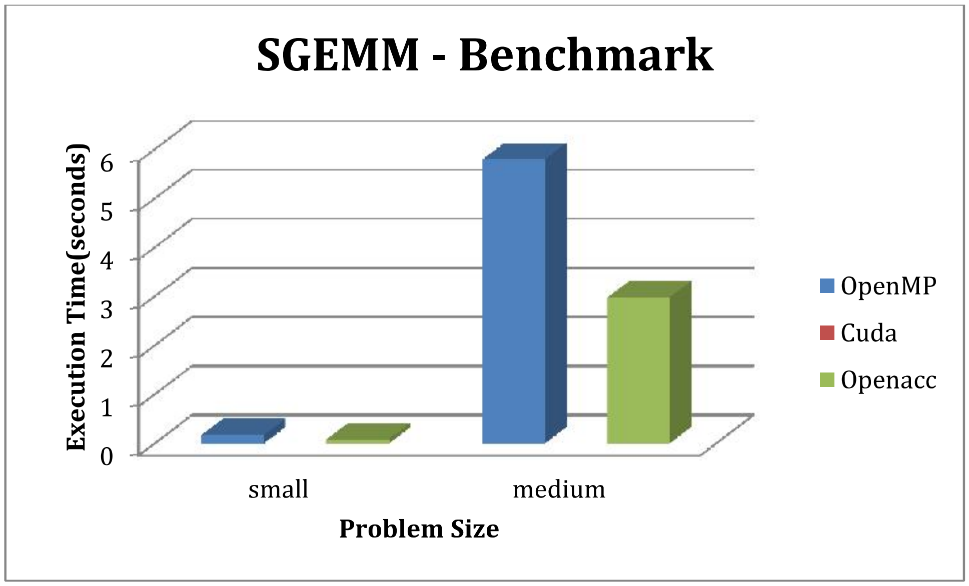

Figure 5.8: Execution Time Graph - SGEMM

Breadth-first Search (BFS): The Breadth-first Search algorithm is a graph algorithm used to search for a particular node in a given graph. We can also identify the shortest path between two nodes using BFS. The Parboil benchmark uses the code from Luo et al. [52] for implementing the BFS application. In BFS, we use the queue data structure to store the data for the visited node. Then, the node is checked for conditions if the current node is the one that we are looking for, and then we end the algorithms and return to the main program. If the node does not match the requirement, then we enqueue the neighboring nodes. If the node that we are searching for is not found, then we end the search. Suppose we have V vertices and E edges in the program. Then, the complexity of the program when we have to search the entire graph will be O(|E|+|V|).

```
        std::deque<int> wavefront;
        wavefront.push_back(source);
        color[source] = GRAY;
        int index;
        while(!wavefront.empty()){
               index = wavefront.front();
               wavefront.pop_front();
#pragma omp parallel for
               for(int i=h_graph_nodes[index].x;
                      i<(h_graph_nodes[index].y +
        h_graph_nodes[index].x); i++){
                            int id = h_graph_edges[i].x;
                            if(color[id] == WHITE){
                                   h_cost[id]=h_cost[index]+1;

#pragma omp critical
        wavefront.push_back(id);
color[id] = GRAY;                      }
                      }
                      color[index] = BLACK;
        }
```

Table 5.17: Code Snippet for BFS OpenMP Version

This implementation does not work right now in OpenACC 2.0 because we do not have "critical" incorporated into the script.

Openacc 2.0 does not support section functionality yet compare to OpenMP. Here we tried using the "acc atomic" clause around the push_back function because when we are pushing an element or retrieving an element from the queue, it is always an atomic operation. Applying "acc atomic" does not work because in the code above, because the atomic function exists inside the push_back function; thus, the surrounding wavefront.push.back will not work. These changes should be performed inside the push_back function because we are using a queue implementation from the std library.

In the section above, we measure the performance of Parboil benchmarks, which are successfully ported using the OpenACC 2.0 specification. The performance parameter used for the graphs is execution time. In general, we observe that the execution time for the CUDA version is the best. This is because CUDA is a low-level language used for GPU computation. The level of optimization provided by low-level languages is always better than higher-level languages. This is because of the overhead associated with implementing the pragma directives and then converting them to machine code. CUDA implementation is very close to the machine level/assembly level implementation. As OpenMP is a high-level language for shared memory programming, the number of threads used here are less comparable to the number of threads spawned in the GPU.

5.5 NAS Parallel Benchmark

NASA developed the NASA Parallel Benchmarks (NPB), which are designed for high-performance computing. NPB has issued a newly created set of parallel-aware algorithms. These benchmarks are architecturally neutral, address generic problems, and readily distributable. The best feature of NPB is accommodating newly emerging systems with increased power and extended new software implementations [39].

NASA Advanced Supercomputing (NAS) is a benchmark suite used to evaluate the performance of highly parallel HPC computing systems. This benchmark mimics the complex scientific computational, communication, and data movement characteristics of large-scale computational fluid dynamics (CFD) applications [40].

NPB has five kernel benchmarks, namely Integer Sort (IS), Embarrassingly Parallel (EP), Conjugate Gradient (CG), irregular memory access and communication, Multi-Grid (MG) on a sequence of meshes, and discrete 3D fast Fourier Transform (FT), which has all-to-all communication. Three pseudo-applications that are part of NPB are Block Tri-diagonal solver (BT), Scalar Penta-diagonal solver (SP), and Lower-Upper Gauss-Seidel solver (LU).

The NPB benchmark has a different class of problem size for measuring and comparing performances. These classes are Class S (small for quick test purposes); Class W (90s workstation size; now likely too small); Classes A, B, C (standard test problems), ~4X size increase going from one class to the next; Classes D, E, F (large test problems, ~16X size increase for each of the previous classes) [39].

We have identified some NAS OpenACC 1.0 implementations in [46], for prospective porting to the OpenACC 2.0 accelerator benchmark. We have focused mainly on three benchmarks: Embarrassingly Parallel (EP) benchmark, Fast Fourier Transfer (FFT), and Multi Grid (MG). The reason for choosing these benchmarks is that they have potential applications in a variety of fields given their usability and parallelism with respect to accelerators. All of these benchmarks require significant computational resources [47]. All of the above-mentioned benchmarks have many function calls inlined, as there was no support for the routine calls in OpenACC 2.0. All these applications inline many of the function calls manually, thus running all of these functions in sequential mode. The main aim of porting these applications is to understand the best way for implementing function calls using OpenACC 2.0 features and to understand the parameters that affect the performance of the applications.

Without the facility of function calls, porting applications required significant code restructuring, as function calls are inherent features of any high-level programming languages. Programmers have to sacrifice some level of parallelism because of a lack of function calls from inside the parallel constructs. By including function calls inside the parallel or kernel constructs, we move a number of lines of the computation to the GPU, which improves the millions of operations per second (MOPs) for the application. Improving MOPs implies the better utilization of computing power and, thereby, an improvement in the performance of the benchmarks.

Let us explore the different optimization techniques and the new features used for the above-mentioned benchmarks one by one.

Embarrassingly Parallel (EP): These are the problems that are inherently parallel in nature. They require much less communication between the intermediate results. This kind of program helps in exploiting as much parallelism as the hardware can offer. These applications help in understanding how the architecture best supports parallelism [39].
NAS implements an algorithm that generates independent "Gaussian random variables" using the Marsaglia-polar method. This function generates random variables, which will have different probability distributions [47].

The "embarrassingly parallel" kernel provides an estimate of the upper achievable limits for floating point performance, i.e. the performance without significant inter-processor communication. The only requirement for communication is generating the sums at the end of the program. This typically follows Monte Carlo simulation applications [47].

```
#pragma acc routine
double randlc_ep1( double *x, double a )
{
  double r;
#pragma acc data pcopy(x , a, r)
 double t1, t2, t3, t4, a1, a2, x1, x2, z;
  t1 = r23 * a;
  a1 = (int) t1;
  a2 = a - t23 * a1;
  t1 = r23 * (*x);
  x1 = (int) t1;
  x2 = *x - t23 * x1;
  t1 = a1 * x2 + a2 * x1;
  t2 = (int) (r23 * t1);
  z = t1 - t23 * t2;
  t3 = t23 * z + a2 * x2;
  t4 = (int) (r46 * t3);
  *x = t3 - t46 * t4;
  r = r46 * (*x);
  return r;
 }
}
```

Table 5.18: Randlc_ep Function

We apply the loop unrolling optimization technique for many of the functions, as OpenACC does not support nested if-else statements inside a parallel region. The EP benchmark has a routine “double randlc_ep(double *x, double a)”, which is inlined in OpenACC 1.0. This routine returns a uniform pseudorandom double precision number in the range (0, 1) by using the linear congruential generator given by the formula - x_ {k+1} = a x_k (mod 2^46). This routine is called approximately (100+ BLKSIZE [1792] + MK [16]) times from inside the parallel loop.

The computation related to this function is run on the GPU using the routine directive. While using the routine directive, we have to understand the data flow between parts of the subprogram. If the data are not synchronized between the GPU and CPU, then there is a significant deviation from the actual result. By making this a function call on the GPU, we reduce the context switch, which is required when a host version of the function is called. This also reduces the data movement between the functions for moving the computed result back to the parallel loop. Using the acc routine directive enables all these calls to be performed on the GPU. With the addition of the routine feature to the entire program, Mops increases drastically by improving the execution time. Though there is improvement in the execution time, it is not as drastic as expected. This is because the code in the routine does not have much parallelism. When we have code that is not parallel on the GPU, sometimes it degrades the performance by adding more overhead in computation.

As shown in Table 5.9, the function call on lines 373 and 375, though they are called blk+100 times, do not exhibit much performance because of the “if” clause surrounding the function call. This serial nature of the code frustrates the ideal performance even after running the code on the GPU.

```
#pragma acc kernels loop independent
private(t1,t2,kk)reduction(+:sx,sy)
  for (k = 1; k <= blksize; k++) {
    kk = k_offset + k + koff;
    t1 = S;
    t2 = an;
    for (i = 1; i <= 100; i++) {
      ik = kk / 2;
      if ((2 * ik) != kk) t3 = randlc_ep(&t1, t2);
      if (ik == 0) break;
      t3 = randlc_ep(&t2, t2);
      kk = ik;
    }
...............
}
#pragma acc kernels loop independent private(t1,t2,kk)
reduction(+:sx,sy)
```

Table 5.19: EP Code Snippet

Fast Fourier Transfer (FFT): Fast Fourier transfer is an algorithm used to compute the discrete Fourier transfer (DFT) and it's inverse. It is one of the most important numerical algorithms, as it converts them to a frequency and vice versa by factorizing the FT matrix into a product of sparse factors. FFT transfers are widely used in many applications in engineering, science, and mathematics [47].

The FT application is time-bound. It does show some improvements in runtime, but it also increases the communication time very significantly. The 3D FFTs are key parts of certain CFD (computational fluid dynamics) applications, notably large eddy turbulence simulations. The 3D FFT steps require considerable communication for operations such as array transpositions [47].

In the FT benchmark, we apply the routine directive by removing the inline function. And this routine has a gang-level parallelism. Because of the routine call support, we restructure the code to run more computations on the GPU [47].

```
#pragma acc routine gang
void return_complex_abs(dcomplex *z1, dcomplex *z2, double *err,int nt)
{
int i;
#pragma acc data copyin(z1,z2) copy(err)
{
#pragma acc loop gang
       for (i = 1; i <= nt; i++)
              {
                     dcomplex z3;
               z3 = (dcomplex){z1[i].real-z2[i].real,z1[i].imag-z2[i].imag};
                     double a = z3.real;
                     double b = z3.imag;
                     double c = z2[i].real;
                     double d = z2[i].imag;
                     double divisor = c*c + d*d;
                     double real = (a*c + b*d) / divisor;
                     double imag = (b*c - a*d) / divisor;
                     dcomplex result = (dcomplex){real, imag};
       err[i]= (sqrt(result.real*result.real)+result.imag*result.imag);
    }
  }
}
```

Table 5.20: FT Code Snippet for return_complex_abs Routine

As shown in the code, we copy the double complex number object dcomplex to the GPU and perform the computation and then calculate the result, which is stored in the result variable. In the non-GPU version of the code, the result was computed in the return_complex_abs function, which was inlined, and then the result was returned to the calling function. In the revised OpenACC version, we not only put the entire function computation on the GPU, but we go one step further to compute the absolute error in the calculated results. As we want to return the results of these computed errors to the host, we have a copy clause in front of the err variable, which takes care of moving err back to the host once we exit the parallel region.

The performance and Mops for this application increase as input size increases. We observe the best performance for CLASS B, which improves the Mops from 3841 to 3912, which, in turn, enhances the performance of the overall application.

Multigrid (MG): MG is a simplified 3D multigrid kernel. This benchmark requires highly structured long-distance communication and tests both short- and long-distance data communication. MG approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method [23]. MG methods in numerical analysis are a group of algorithms for solving differential equations using a hierarchy of discretization [47].

This benchmark contains almost 1800 lines of code. This can be compared to more real-world applications. The MG benchmark contains three functions that are inlined in OpenACC 1.0 versions. The functions that were inlined are bubble, randlc, and vranlc.

```
#pragma acc routine(bubble) gang

void bubble(double ten[][2], int j1[][2], int j2[][2], int j3[][2],int
m, int ind)



#pragma acc routine vector

double randlc( double *x, double a )



#pragma acc routine(vranlc)

void vranlc( int n, double *x, double a, double y[] )
```

Table 5.21: MG Inline Functions Replaced by Routine Functions

Bubble function implements a bubble-sort algorithm for the grid elements, which compares each element with its neighbor and bubbles out the smallest/biggest element if necessary by swapping; traversing the entire list until it is sorted. Bubble sort has an average and a best-case complexity of O (n2). This function is inherently not so parallel because of significant branching in the algorithm. As shown in the code below, multiple if-else statements and loop unrolling are applied in order to ensure code parallelism. We apply gang-level parallelism to each unrolled loop, which distributes the computation across the gang of the thread block.

```
#pragma acc loop seq

  for ( i = 0; i < n; i++ ) {

    t1 = r23 * (*x);

    x1 = (int) t1;

    x2 = *x - t23 * x1;

    t1 = a1 * x2 + a2 * x1;

    t2 = (int) (r23 * t1);

    z = t1 - t23 * t2;

    t3 = t23 * z + a2 * x2;

    t4 = (int) (r46 * t3) ;

    *x = t3 - t46 * t4;

    y[i] = r46 * (*x);

  }
```

Table 5.22: Vranlc Function Snippet

```
if (ind == 1) {
   #pragma acc loop gang
   for (i = 0; i < m-1; i++) {
        if (ten[i][ind] > ten[i+1][ind]) {
        temp = ten[i+1][ind];
        ten[i+1][ind] = ten[i][ind];
        ten[i][ind] = temp;

        j_temp = j1[i+1][ind];
        j1[i+1][ind] = j1[i][ind];
        j1[i][ind] = j_temp;

        j_temp = j2[i+1][ind];
        j2[i+1][ind] = j2[i][ind];
        j2[i][ind] = j_temp;

        j_temp = j3[i+1][ind];
        j3[i+1][ind] = j3[i][ind];
        j3[i][ind] = j_temp;  }  }
#pragma acc loop gang
for (i = 0; i < m-1; i++) {
      if (ten[i][ind] <= ten[i+1][ind]) {
              return;  }    }  }
  if(ind!=1)
  {
     #pragma acc loop gang
     for (i = 0; i < m-1; i++) {
      if (ten[i][ind] < ten[i+1][ind]) {
        temp = ten[i+1][ind];
        ten[i+1][ind] = ten[i][ind];
        ten[i][ind] = temp;

        j_temp = j1[i+1][ind];
        j1[i+1][ind] = j1[i][ind];
        j1[i][ind] = j_temp;

        j_temp = j2[i+1][ind];
        j2[i+1][ind] = j2[i][ind];
        j2[i][ind] = j_temp;

        j_temp = j3[i+1][ind];
        j3[i+1][ind] = j3[i][ind];
        j3[i][ind] = j_temp;
      }
       }
  #pragma acc loop gang
    for (i = 0; i < m-1; i++) {
      if (ten[i][ind] < ten[i+1][ind]) {
        return;
      }  }  }  }
```

Table 5.23: Bubble Function in MG

The Vranlc function has the computations where most of the lines in the function have dependency over the previous lines. Because of these dependencies, we direct the code to execute in sequence. Launching threads is not costly when compared to CPU thread creation. Thus, there is no harm in creating the threads even if the code is not highly parallel. The final code will still be faster than the normal computation.

The randlc function is called zran3, also referred to as the vranlc function. This function is executed in sequence because of the dependencies.

In this application, we transfer extensive computation to the GPU. Though all three functions put on the GPU are not highly parallel, there is still an increase in the number of maps for the execution, thereby increasing the performance in the OpenACC 2.0 when compared to the OpenACC 1.0.

Class	OpenACC 1.0 (Mops)	OpenACC 2.0 (Mops)
S	970	1102
A	7894	8310
B	8078	8202

Table 5.24: Mops Comparison between OpenACC 1.0 and OpenACC 2.0

5.6 Analysis and Observations

When we are parallelizing the code to run on the GPU, as observed in the previous benchmarks, the number of lines of code in the program tends to increase. But this definitely does not complicate the code structure. Rather, we simplify the code by vectorising it and extracting the best results on the GPU. This is because when compared to CPUs, GPUs are structurally very simple. They do not have mechanisms for handling complex situations like context switching or using shared memory. But they have a very powerful computational capacity. This can be best harnessed using simple and vectorizable loops. The OpenACC 2.0 routine directive does not support library function calls. If a loop that is reasonably vectorizable has a simple function call to the library, then we cannot parallelize the loop.

Using routine directives for function calls improve portability of the application. This directive helps retain the natural program structure, which reduces the time required for porting the application.

We use the routine directive only when we have a function that is called from inside the parallel loop. In the absence of a routine directive, we use code inlining. Inlining is easier from the normal programming perspective, as it does not need much analysis; the code can simply be inserted into the function or code space.

Using the routine directive, the routines are compiled separately for each GPU version and linked later using the linker provided by the OpenACC runtime. The routine directive gives the highest performance when we are calling the function from inside the highly parallel loop, and even the called function has more computation and the loops inside the routine are vectorizable.

In NAS, the performance of the benchmarks does not increase drastically after inlining the functions because the functions that are inlined do not contribute heavily towards the execution time of the entire program. But using the routine directive has increased the number of MOPs, which is one of the performance parameters indicating improved resource utilization. We also have more lines of code executed on the GPU. As kernels/parallel constructs allow only one 'if' statement inside, the number of lines of code increases to preclude many if-else statements.

Chapter 6. Conclusion and Future Work

6.1 Conclusion

In this thesis, we develop an accelerator benchmark suite that provides an extensive overview of its scientific applications. These applications are selected from the NPB and Parboil benchmarks, both of which are widely used in research areas. Of all of these applications, we concentrate on the regions of code with computationally intense loops and also present some of the best approaches for mapping these loops to the GPU architecture using OpenACC parallel programming API. We also discuss optimization techniques like loop unrolling, loop scheduling, and cache optimization. We convert multidimensional matrixes to vectors for better cache utilization and for the improvement of memory access. We use an unstructured data lifetime of the variables on the GPUs, which will reduce unnecessary data movement between the host and device. Data management is always one of the biggest challenges in GPU computing. Applying all the mentioned optimization techniques will help to align the performance of OpenACC very closely to that of CUDA. These benchmarks help evaluate the programming model itself and its suitability to accelerators.

OpenACC 2.0 has implemented some of the most important features like the routine directive, atomics directive, unstructured data lifetime clauses, etc. All these features were missing in OpenACC 1.0. These are some of the very high-level features implemented for programmers.

Sometimes implementing these features poses challenges, as many are not very straightforward due to the lack of different use cases describing these directives. For example, not all the directives work when used for the first time. They need different levels of architectural understanding.

Understanding and porting applications to GPUs using OpenACC is not only challenging, but becomes overwhelming sometimes. We do not see one-to-one mapping between the problems in porting the application and the directives for handling a specific problem. For instance, applying fine-grained parallelism to a loop looks simple once we have the code running, but much effort is required to check which level of parallelism best suits the different combinations of clauses and directives. Many sections of these optimizations are the result of trial and error by using different clauses.

The developed accelerator benchmark will highlight many of the difficulties we face in porting applications by addressing how to overcome some of the challenges encountered in validating the effectiveness of certain working applications.

This research aims to provide feedback to the OpenACC committee to help improve the model by using analysis of performance during creation of the implementation with respect to particular clauses.

6.2 Future Work

Although the results presented in this thesis are very promising and cogent, there are some areas that can definitely be improved.

As a topic of future research, we should explore new optimization technologies for the benchmarks to further enhance performance. As GPU computing is an emerging field, and as there are not many benchmarks implemented for OpenACC, it will be a good idea to develop a set of applications involving computational complexity and to create a new brand benchmark for accelerators. This benchmark suite can be used for performance comparison between different architectures like NVIDIA's GPU, Intel Xeon Phi, AMD's GPU, APU, etc.

OpenACC has not yet implemented the concept of tasking as we have in OpenMP. Tasking will help in parallelizing the applications, which will have recursive calls. Coordinating the implementation of tasks for OpenACC and then porting the applications from the Barcelona OpenMP Tasks Suite (BOTS) [53] would also be an interesting topic of future research.

Bibliography

[1] Rainer Buchty, Vincent Heuveline, Wolfgang Karl, Jan-Philipp Weis, "A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators, Special Issue Paper, *Concurrency and Computation: Practice and Experience*", Volume 24, Issue 7, pages 663–675, May 2012.

[2] D.M.Kunzman, "International Symposium on Parallel and Distributed Processing Workshops, *Programming Heterogeneous Systems*", Page(s):2061 – 2064, Issue 16-20 May 2011

[3] Amar Shan. (2006, Jan). "Heterogeneous Processing: a Strategy for Augmenting Moore's Law, *Linux Journal*". [Online]. Available: http://www.linuxjournal.com/article/8368

[4] CUDA Parallel Computing Platform,[Online]. Available: http://www.NVIDIA.com/object/CUDA_home_new.html

[5] Stefan Möhl, "FPGAs in HPC: *Part 1, Introduction to Hybrid Computing*".[Online]. Available: http://www.vidqt.com/id/Vfhy1pKPggs?lang=en

[6] GPU (Graphics Processing Unit).[Online]. Available: http://www.ubergizmo.com/what-is/gpu-graphics-processing-unit/

[7] Steven W. Smith. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. [Online]. Available: http://www.dspguide.com/

[8] Chunhua Hu, David Bell, "*KeyStone Memory Architecture*",Texas Instruments.

[9] Zoran Nikolic, Gaurav Agarwal, Brooke Williams, Stephanie Pearson, "*IT Gives Sight to Vision-enabled Automotive Technologies*",Texas Instruments.

[10] Grzegorz Budzyn, *Advanced Micro controllers*, *Lecture 11: Digital Signal Controllers & Digital Signal Processors.* Wroclaw University of Technology.[Online].Available: http://www.ue.pwr.wroc.pl/advanced_microcontrollers/adv_m_11.pdf

[11] Oliver Mattes, Wolfgang Karl, "*Targeting Self-organizing Memory Management in Future Silicon Photonics System Architecture*", 2014.

[12] Rainer Buchty, Vincent Heuveline, Wolfgang Karl, Jan-Philipp Weis, "A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators, Special Issue Paper, *Concurrency and Computation: Practice and Experience*", Volume 24, Issue 7, pages 663–675, May 2012.

[13] Wm.A.Wulf, A.Sally, McKee, "Hitting the Memory Wall: Implications of the Obvious, *Computer Architecture News*", Volume 23, Issue 1, March 1995

[14] Andrew, David, Targhetta,(2008, April), *Heterogeneous Parallel Computing*,

[Online]. Available: https://cs.nmt.edu/~cs451/lectures/grad/targhetta.pdf

[15] Peter Messmer, *GPU Architecture Overview and CUDA Basics,* (NVIDIA). [Online]. Available:https://www.youtube.com/watch?v=nRSxp5ZKwhQ

[16] Mantha Anil Srimanth, Naveen Jawalkar, Ashwini Deshmukh, ”State Of Art Technologies In Graphics Processing Unit (GPU) Architectures, *Global Journal of Advanced Engineering Technologies*”, Vol 3, Issue3- 2014,ISSN: 2277-6370

[17] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, Jianmin Chen, “Architecture Comparisons between NVIDIA and ATI GPUs: *Computation Parallelism and Data Communications, Workload Characterization*”, 2011,E-ISBN: 978-1-4577-2062-8,Pages: 205 – 215.

[18] GPU Accelerated Computing, NVIDIA, [Online]. Available: http://www.NVIDIA.com/object/what-is-gpu-computing.html

[19] “GPU Programming, *NVIDIA Tesla GPU Servers*”, Thinkmate NVIDIA, [Online]. Available: http://www.thinkmate.com/systems/gpu

[20] *GPU- Accelerated Libraries*, NVIDIA, [Online]. Available: https://developer.NVIDIA.com/gpu-accelerated-libraries

[21] *OpenACC, CUDA Zone,* NVIDIA, [Online]. Available: https://developer.NVIDIA.com/OpenACC

[22] *About CUDA, CUDA Zone*, NVIDIA, [Online]. Available: https://developer.NVIDIA.com/about-CUDA

[23] *While AMD goes for Brains, NVIDIA goes for Brawn*,[Online]. Available: http://www.destructoid.com/while-amd-goes-for-brains-NVIDIA-goes-for-brawn-210726.phtml

[24] NVIDIA's Next Generation CUDA Compute Architecture - Kepler GK110, whitepaper, NVIDIA.

[25] Jen-Hsun Huang, “The Kepler GPU Architecture, *GPU Technology Conference*”, San Jose, CA, 2012 , May 14- 17,

[27] John Morris, A closer look at AMD's Heterogeneous Computing, [Online].Available:http://www.zdnet.com/a-closer-look-at-amds-heterogeneous-computing-7000014840/

[28] *AMD Kaveri APU Architecture Detailed* – Next Generation APU Featuring Steamroller and GCN Cores, [Online].Available: http://wccftech.com/amd-kaveri-apu-architecture-detailed-generation-apu-featuring-steamroller-gcn-cores/

[29] Hank Tolman, *AMD Kaveri APU Architecture Overview,* [Online].Available: http://benchmarkreviews.com/11622/amd-kaveri-apu-architecture-overview/

[30] CUDA C Programming Guide, NVIDIA, Developer Zone, [Online].

Available:https://www.clear.rice.edu/comp422/resources/CUDA/html/CUDA-c-programming-guide/#programming-model

[31] Blaise Barney, *OpenMP tutorial,* Lawrence Livermore National Laboratory, [Online]. Available: https://computing.llnl.gov/tutorials/OpenMP/

[32] Matthew Scarpino, *A Gentle Introduction to OpenCL*.[Online]. Available: http://www.drdobbs.com/parallel/a-gentle-introduction-to-OpenCL/231002854

[33] Overview about OpenCL and Parallel Processing, [Online]. Available: http://www.cmsoft.com.br/OpenCL-tutorial/overview-OpenCL-parallel-processing/

[34] John A.Stratton, Chistopher Rodrigues, I-Jui Sung, nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, Wen-mei W.Hwu, "Parboil : A Revised Benchmark Suite for Scientific and Commercial Throughput Computing, *IMPACT Technical Report*", March 2,2012; Revised : March 19,2012.

[35] Michael Wolfe,(2014), "OpenACC 2.0 and the PGI Accelerator Compilers, *GPU Technology Conference*".The Portland group

[36] A.Danalis, G.Marin, C.McCurdy, J.Meredith, P.Roth, K.Spafford, V.Tipparaju, J.Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processors*", March 2010.

[37] Md. Rezaur Rahman, *The Scalable Heterogeneous Computing Benchmark Suite (SHOC)* for Intel® Xeon Phi™, Intel, April 9, 2013.

[38] M.Boyer, Jiayuan Meng, D.Tarjan, J.W.Sheaffer, Sang-Ha Lee, Skadron, K.Shuai Che, "Rodinia: A Benchmark Suite for Heterogeneous Computing, Workload Characterization", Page(s):44 - 54 ,Austin, Texas.

[39] D.H.Bailey, E.Barszcz, J.T.Barton, D.S.Browning, R.L.Carter, L.Dagum, R.A.Fatoohi, P.O. Frederickson3, T.A.Lasinski1, R.S.Schreiber3, H.D.Simon2, V.Venkatakrishnan2 and S.K. Weeratunga2, "The NAS Parallel Benchmarks, *Journal of Supercomputer Applications*", vol. 5, no. 3 (Fall 1991).

[40] Geyong Min, Beniamino Di Martino, Laurence T.Yang, "Frontiers of High Performance Computing and Networking, *International Workshops*", FHPCN, XHPC, S-GRACE, GridGIS, HPC-GTP, PDCE, ParDMCom, WOMP, ISDF, and UPWN, Sorrento, Italy, December 2006, Proceesings.

[41] James Beyer, OpenACC 2.0 Elucidated, Cray. [Online]. [Available: http://blog.cray.com/?p=6455

[42] David Wallace, *OpenACC for HPC Accelerator Programming*, Cray.[Online].Available: http://blog.cray.com/?p=6371

[43] Jeff Larkin, *7 Powerful New Features in OpenACC 2.0*, NVIDIA.[Online].Available: http://devblogs.NVIDIA.com/parallelforall/7-powerful-new-features-OpenACC-2-0/

[44] Mark Ebersole, *The How to GPU series, CUDA Casts Episode #17,Unstructured data lifetimes in OpenACC 2.0*, .[Online].Available:https://www.youtube.com/watch?v=dsffyNx7m5Q

[45] The OpenACC™ Application Programming Interface, Specification Version 2.0, NVIDIA, June, 2013, Corrected, August, 2013

[46] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yonghong Yan, Barbara Chapman, "OpenACC Parallelization and Optimization of NAS Parallel Benchmarks" , 2014, S4340.

[47] D.Bailey, E.Barszcz, J.Barton, D.Browning, R.Carter, L.Dagum, R.Fatoohi, S.Fineberg, P.Frederickson, T.Lasinski, R.Schreiber, H.Simon, V.Venkatakrishnan and S.Weeratunga, "The NAS Parallel Benchmarks", RNR Technical Report RNR-94-007, March 1994, [Online], Available:http://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf

[48] Sloot, Peter M.A. et al. (May 28, 2002), "Computational Science – ICCS 2002: *International Conference*", Amsterdam, The Netherlands, April 21–24, 2002. Proceedings, Part I. Page 843. Publisher: Springer.

[49] Y.H.Qian, D.D'Humieres, and P.Lallemand. "Lattice BGK Models for Navier-Stokes Equation, *Europhysics Letters*" Pages:479–484, 1992.

[50] S.S.Stone, J.P.Haldar, S.C.Tsao, W.W.Hwu, Z.Liang, and B.P. Sutton. "Accelerating Advanced MRI Reconstructions on GPUs. *In International Conference on Computing Frontiers*", pages 261–272, 2008.

[51] V.Volkov and J.W.Demmel."Benchmarking GPUs to Tune Dense Linear Algebra. *In Supercomputing*", pages 1–11, Piscataway, NJ, USA, 2008.

[52] L.Luo, M.Wong, and W.m.Hwu. "An Effective GPU Implementation of Breadth-first Search, *Design Automation Conference*", pages 52–55, June 2010.

[53] A.Duran, X.Teruel, R.Ferrer, X.Martorell, E.Ayguade,"Barcelona OpenMP Tasks Suite", Page(s):124 - 131 , 22-25 Sept. 2009.

[54] Mark Ebersole, "Accelerate Your Programming or Science Career with GPU Computing: *An Introduction to Using CUDA*".[Online]. Available: https://www.youtube.com/watch?v=KfGnLltyRH4

Appendix A

Compiling and Running Benchmarks

A1. Compiling and Running Parboil Benchmark

1. Download the benchmark suite.
2. Use module load command to load the compiler. The compiler used for the development is pgi/14.9 from PGI.

```
$ module avail

$ module load pgi/14.9 // for enabling the pgi compiler
```

3. Navigate inside the PARBOIL parent folder, you should find a '*parboil*' executable file. Now run the bellow commands to compile and run the different benchmarks, with different problem size.

```
$./parboil command benchmark version problem_size    //syntax for executing benchmarks
```

```
Examples :

$./parboil compile stencil OpenACC // to compile , problem_size is optional

$.parboil run stencil OpenACC  // to run, problem_size is mandatory
```

Benchmark folder contains all the benchmarks that are implemented and the *src* folder contains different versions of the benchmark; we have folders for each benchmark inside the *src* folder.

Dataset folder contains different problem sizes for each benchmark. We have to choose this problem size of the folder dataset->{benchmark} ->{problem_size} ->input.

A2. Compiling and Running NAS Parallel Benchmark

1. Download the benchmark suite.
2. Use module load command to load the compiler. We can use either pgi or hmpp compiler.

```
$module avail

$module load hmpp/3.4.1 // for hmpp compiler

$module load pgi/14.9 // for pgi compiler
```

3. Navigate to the benchmark you want to compile from the parent directory. When you navigate for the first time in the benchmark directory, before you hit the same child directory inside, create '*bin*' and '*temp*' folders if they are not already present.

4. Example, if we want to compile FT benchmark, then follow the steps below:

```
$cd FT/FT
$make CC=pgcc CLASS=S // case sensitive commands, this will
compile FT benchmark
```

5. Now from the current directory, navigate back one step, where you have created the '*bin*' directory. After compilation, the executable files are pushed into the bin directory according to the make rules. Execute the benchmark that you just compiled for a specific class.

```
//from current directory, where we left off in the last section

$cd ..
```

```
$cd bin

$./ft.S.x                 // the executable file created after compilation

from the previous step.

                          // This step will execute and produce the results
```

Appendix B

Explanation of Abbreviations

Table B. 1: Abbreviations used in this document

Abbreviation	**Acronym**	**Meaning**
OpenACC	Open Accelerators	Accelerator programming standard for CPU/accelerator system for scientific computing.
OpenMP	Open Multi-Processing	A programming model for multi-platform shared memory multiprocessing platform.
CUDA	Compute Unified Device Architecture	NVIDIA's parallel computing architecture for GPUs.
OpenCL	Open Computing Language	API for heterogeneous programming.
HPC	High Performance Computing	Area of parallel computing, which delivers high-performance for most complicated scientific and commercial applications.
GPU	Graphics Processing Unit	Electronic circuit capable of performing rapid calculations, especially for graphics processing.
CPU	Central Processing Unit	The electronic circuitry within a computer that carries out the basic arithmetic, logical, control, and input/output operations.
APU	Accelerated Processing Unit	Set of 64-bit microprocessors from AMD designed to act as a CPU and graphics accelerator (GPU) on a single chip.
NAS	NASA Advanced Supercomputing	The NASA Advanced Supercomputing (NAS) Division is enabling advances in high-end computing technologies.
NPB	NAS Parallel Benchmarks	Set of performance benchmarks for highly parallel architectures.

Abbreviation	Acronym	Meaning
Mops	Million Operations per second	Measurement of computer performance using number of operations performed per second.
API	Application Programming Interface	Standard set of routines and/or protocols that will help to communicate with different software components.