Experimental Evaluation and Enhancement of Optimizations of Annotation-Based and Automatic Parallel Code Generators for GPUs

BY

Anas Ali Almousa

A Dissertation Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

January-2017

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by ANAS ALI ALMOUSA under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.**

Dr. Mayez Abdullah Al-Mouhamed
(Advisor)

Dr. Adel Fadhl Noor Ahmed
Department Chairman

Dr. Shokri Zaki Selim Hassan Zaki
(Member)

Dr. Salam A. Zummo
Dean of Graduate Studies

Dr. Mohamed Elnasir Salaheddin Elrabaa
(Member)

25/4/17
Date

Dr. Tariq Ahmed Helmi Albusyuni
(Member)

Dr. Moataz Ali Kamaleldin Ahmed
(Member)

All praise goes to Allah alone, whose blessings we cannot enumerate.

I dedicate this dissertation to my Father **Ali Almousa**, who did the most support, help and encouragement for me during my pursuit of higher education and this degree. To my mother **Najwa Almuradi** whom without her and my father I would never be even close to who I am today. To my great, loving and supporting siblings, especially my brother **Hassan Almousa** who patiently taught me and answered my questions about programming and electronics when I was a teen. To **Tareq Nashawati** who was a role model for me as the computer expert when I was a kid.

ACKNOWLEDGMENTS

I would like to thank my Advisor and Committee members Mayez Al-Mouhamed, Shokri Selim, Mohamed Elrabaa, Tariq Albusyuni and Moataz Ahmed for their guidance, support and valuable feedback that helped in shaping the dissertation.

Also I would like to thank King Abdullah University of Science and Technology for providing access to GPU workstations in the early stages of my investigation of this study.

Moreover, I would like to thank Information Technology Center and College of computer sciences and Engineering at King Fahd University of Petroleum and Minerals for facilitating access to HPC systems for this research work.

In addition, I would like to thank my colleague Ayaz Khan for sending me his work on global synchronization for comparison.

Finally, my sincere thanks goes to my colleges and friends Ayham Zaza, Yousef Elarian, Yazan Rawashdeh, Ayman Hroub and many more who were really supportive and caring friends during my PhD and helped me through hardships.

TABLE OF CONTENTS

LIST OF TABLES

TABLE OF LISTINGS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	:	Application Programmer Interface
AWG	:	Automatic Wrapper Generator
GPU	:	Graphics Processing Unit
BLAS	:	Basic Linear Algebra Subroutines
HPC	:	High performance computing
LDD	:	Least Dependence Distance
SAXPY	:	Single Precision Scaled-X Plus Y
SMR	:	Symmetrical Multiprocessor Residency
RBSM	:	[Number of] Resident Blocks of a Symmetrical Multiprocessor
RBTGPU	:	[Number of] Total Resident Blocks of a GPU
NSM	:	Number of Symmetrical Multiprocessors in a GPU

ABSTRACT

Full Name : Anas Ali Almousa

Thesis Title : Experimental Evaluation and Enhancement of optimizations of Annotation-Based and Automatic Parallel code generators for GPUs

Major Field : Computer Science and Engineering

Date of Degree : January 2017

GPUs have gained a lot of attention in the HPC community lately. Since that, a lot of research was done on creating and optimization language models that enable programming these devices. OpenACC standard has emerged to standardize the effort of creating a high-level directive based language extension for several conventional programming languages to enable easy programming of these devices. In a recent change to the draft, extra directives were introduced to allow software library component reuse. This theoretically enabled the use of some established libraries that are implemented on GPUs. However, practically, there are difficulties in calling libraries that separate implementation details and/or use hidden data structures; expecting them as arguments. In this dissertation, we propose a systematic approach to enable the use of a variety of GPU libraries from within OpenACC compute regions. when applying our approach to CuBLAS, which is a library for performing Basic Linear Algebra operations, we found that our approach enhances performance by up to thirty-two times over OpenACC alone and up top 2.52 over using CuBLAS alond. In addition, we achieved a reduction to code size down to fifty-two percent of the original OpenACC code. Our approach also opens the opportunity to call those libraries with any parallel granularity desired (thread/ block

of threads/ all threads); eliminating the need for ending the GPU execution region in order to call those libraries from the CPU.

In GPUs, synchronization of different gangs of threads is only possible through the host systems processor. Hence, it is only possible to implement synchronous iterations through repeated launches of GPU kernels for each iteration step. Some techniques were proposed to reduce that overhead using shared variables. However, these are prone to deadlocks and are error prone when implemented without correct enforcement of memory coherency. In this dissertation, we provide a comprehensive analysis of the reasons and parameters involved in the deadlock-inducing behavior in such synchronization techniques, where we present crisp upper limits for parameters involved to avoid deadlocks in these techniques. We also analyze the scheduling behavior for different gangs inside the GPU. Based on the scheduler analysis, we have introduced a new novel method for implementing synchronized iterations and any one-way producer-consumer methods without the need synchronization through the host CPU and without suffering limitations for the parameters involved in the other methods proposed in literature. Our analyses show that the performance of our method has a speedup of about 1.38 over inter-block synchronization methods used in literature. We also introduce analysis for the cases and parameters involved for omitting the explicit shared variable synchronization all together and gaining performance similar to asynchronous algorithms via arranging for an implicit scheduling-based synchronization for such cases; where we also see a gain in speedup nearing 1.8 for applications where blocks do not share a balanced load among them. We also show that for a large majority of the cases where an explicit synchronization is needed, host-based synchronization is the best choice.

ملخص الرسالة

الاسم الكامل: أنس علي الموسى

عنوان الرسالة: تقييمٌ وتعزيزٌ تجريبيان للتهيئات التحسينية للشيفرات البرمجية المتوازية المستهدِفة لوحدات معالجة الرسومات التي تطبق بواسطة المترجمات التلقائية او المستندة الى حواش

التخصص: هندسة و علوم الحاسب

تاريخ الدرجة العلمية: ربيع الآخر 1438 هـ

لقد اكتسبت وحدات معالجة الرسومات الكثير من الاهتمام في الحوسبة فائقة الاداء مؤخرا .منذ ذلك الحين، تم إجراء الكثير من البحوث على صنع وتحسين نماذج اللغة التي تمكن برمجة هذه الأجهزة .وقد برز معيار OpenACC لدمج الجهد المبذول في صنع امتدادات توجيهية عالية المستوى لعدد من اللغات البرمجية المألوفة بهدف تمكين عملية برمجية سهلة لهذه الأجهزة الرسومية .وفي تغيير حدث مؤخرا على مشروع OpenACC ، أُدخلت توجيهات إضافية للسماح بإعادة استخدام مكونات المكتبات البرامجية .نظريا ، مكن ذلك من استخدام بعض المكتبات الراسخة التي تستهدف وحدات معالجة الرسومات . مع ذلك، عمليا، هناك صعوبات في استدعاء المكتبات التي تفصل بين تفاصيل البرمجة و واجهة الاستدعاء او بشكل عام تلك التي تستخدم هياكل البيانات المخفية في واجهة الاستدعاء .في هذه الأطروحة، نقترح منهجية لتمكين استخدام مجموعة متنوعة من المكتبات البرمجية المستهدفة لوحدات معالجة الرسومات من داخل مناطق الحوسبة في .OpenACCعند تطبيق نهجنا ل CUBLAS، وهي مكتبة لأداء عمليات الجبر الخطي الأساسي، وجدنا أن نهجنا يعزز الأداء بنسبة تصل إلى ثلاثين مرة على OpenACC وحدها وحتى أعلى 2.52 أكثر من استخدام CUBLAS وحدها. وبالإضافة إلى ذلك، حققنا خفضا لحجم الرمز البرمجي وصولا الى اثنين وخمسين في المئة من الرمز الاصلي المعتمد فقط على لـ OpenACC. نهجنا ذاك يفتح الفرصة لاستدعاء تلك المكتبات مع أي تقسيمات موازية مطلوبة (all threads / block /thread)؛ مما يلغي الحاجة لوضع نهاية لمنطقة التنفيذ الرسومي بهدف استدعاء تلك المكتبات البرمجية عن طريق وحدة المعالجة المركزية. في وحدات معالجة الرسومات، يمكن مزامنة جماعات مختلفة من الـ threads فقط من خلال معالج الأنظمة المضيف. لوحدة الرسوميات وبالتالي، فمن الممكن فقط تنفيذ المزامنة الـ مجموعات مختلفة من الthreads في الخوارزميات التكرارية من خلال تكرار اطلاق وحدات تنفيذية على وحدة المعالجة الرسومية لكل خطوة تكرارية تحتاج مزامنة .تم اقتراح بعض التقنيات لتقليل تكلفة المزامنة تلك باستخدام المتغيرات المشتركة .ومع ذلك، تبقى هذه التقنيات عرضة لانسداد الطريق. وتكون عرضة للخطأ عند تنفيذها دون الفرض الصحيح للانسجام في الذاكرة .في هذه الأطروحة، ونحن نقدم تحليلا شاملا للأسباب و والمعاملات اللمعنية في الوصول لانسداد الطريق في مثل هذه التقنيات لتزامنية، حيث نقدم الحدود العليا هش للمعاماات المعنية لتجنب انيداد الطريق في هذه التقنيات .نحن أيضا نحلل السلوك المعمول به في جدولة ال blocks المختلفة داخل وحدة

الرسوميات .استنادا إلى تحليلنا ذاك، أدخلنا طريقة جديدة لتنفيذ تكرارات متزامنة أو تنفيذ أي تزامن باتجاه واحد بين منتج و مستلك، دون الحاجة للمزامنة من خلال وحدة المعالجة المركزية المضيفة ودون أن تعاني من قيود على عدد المجموعات المشاركة في الطرق الأخرى المقترحة في المواد المطبوعة. وتُظهر تحليلاتنا أن أداء أسلوبنا لديه سرعة تصل إلى حوالي 1.38 على أساليب التزامن بين المجموعات.الموجودة في المواد المطبوعة. ونقدم أيضا تحليلا للحالات والمعاملات التي ينطوي عليها حذف التزامنا لصريح المشترك مما يسبب واكتساب أداء مشابه للخوارزميات غير المتزامنة من خلال ترتيب التزامن القائم على جدولة ضمنية في مثل هذه الحالات؛ حيث نرى أيضا مكاسب تقترب من 1.8 للتطبيقات حيث المجموعات التي تعاني من عدم تساوي الحمل فيما بينها .وتبين لنا أيضا أنه بالنسبة إلى الغالبية العظمى من الحالات التي يلزم فيها التزامن الصريح، فإن المزامنة المستندة إلى المضيف هي الخيار الأفضل.

CHAPTER 1
INTRODUCTION

For many years since the making of microprocessors, the majority of software development(in domains other than the high performance community) has been able to ignore concurrency[1]. Software developers have been able to benefit from architectural enhancements in processors, memory, and I/O to improve their software in what had been described as the "Free lunch" for software developers[2]. As architectural enhancements that improve the performance of serial code is slowing down considerably; that "free lunch" will eventually end [1]. Software developers have to ride the wave of parallelism if they do not want their software to be left behind regarding performance enhancements and new features that become feasible because of high performance.

In the past decade, the field of high performance computing has been rapidly changing. From one point, hardware accelerators had been pushing the boundaries of computational discoveries due to their characteristics that incorporate low-cost, energy efficiency, and performance[3]. Graphics processing devices, when designed and operated to target general purpose computing had demonstrated impressive advances in computations science. Some experts believe that those systems would be the building blocks for future high performance computing platforms[4]. However, the programming of these devices is yet to mature enough to a state that allows developers easy access to the performance of these devices. That is why developing established, efficient and an easy programming model is vital to the success and longevity of these architectures.

There has been a growing research for reducing the difficulty of programming these devices[5][6][7]. The programming model of both Compute Unified Device Architecture (CUDA) [8] and the standardized Open Computing Language (OpenCL) [9] provide an interface with less hassling programming experience.still, when compared to parallel programming for CPUs using standards such as the OpenMP[10], this programming

experience still bears complexity. This motivated the creation of several directive based programming interfaces such as OpenMPC[11] and HiCUDA[12] from the academia and (OpenACC [13] , PGI Accelerator [14] , OpenMP for Accelerators[15], and R-Stream[16] from the industry. These models provide different details of abstraction and usage cost for achieving optimized and restructured code.

On the other hand, the use of software libraries has recently become an essential part of software design. When appropriately applied, the use of libraries would reduce development cost since it reduces expertise needed by developers, reduces domain knowledge related to the problems solved by such libraries and reduces the overhead in software testing.

This concept takes particular importance in the domain of numerical computations and simulation since computations in these domains share a lot of well-known algorithms and methods. Enhancing the performance of numerical computations needs a deep understanding of algorithms and computer architecture involved. Using renowned and production-grade libraries in this domain usually have a significant reduction in effort, or even achieve performance and goals that are not achievable in certain environments.

Parallel numerical computational libraries on Accelerator devices are gaining much attention lately [17] [18][19] due to the help they provide in achieving performance or the ease of use they are providing. Because of the large number of parameters involved in programming accelerator devices, many of these libraries are tuned programmatically for best performance per the specifics of the problem instance related to the problem domain (e.g. matrix size, symmetry, and aspect ratio for matrix related algorithms).

OpenACC [13] is a standard designed for simplifying targeting GPU systems and heterogeneous systems(CPU/GPU). OpenACC standard allows a compiler to use automated parallelization techniques against serial code. The standard allows the user to leave all key choices for the compiler to decide. Its main focus is offloading loops to target accelerates because loops are structures that usually have opportunities for data parallelism. A collection of standardized compiler directives directs the compiler to loops

and regions of code to be compiled for an accelerator device. OpenACC's design objective is portability across CPUs, operating systems, and accelerators, including APUs, FPGAs, GPUs, and many-core coprocessors.

"The directives and programming model defined in the OpenACC API document allow programmers to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown.

These details are implicit in the programming model and are managed by the OpenACC API-enabled compilers and runtimes. The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on the mapping of loops onto an accelerator, and similar performance-related details.**"**[1]

In a similar way to OpenMP programming mode, the user can use directives and functions to direct the compiler of C, C++ and Fortran towards regions to execute on an accelerator device. For both OpenACC and OpenMP 4.0 and newer; the code can be accelerated over both the CPU and GPU.

As OpenACC depends on compiler technology to target accelerator devices by annotating serial code; the output of such programs heavily depends on compiler technology. In many instances, guaranteeing that automated compiler optimizations do not break correctness hinders the automatic enhancement of performance of such programs. Domain knowledge of the application, on the other hand, allows a human expert to apply such optimizations when they are considered safe. Moreover, domain knowledge allows tuning specific parameters for best performance. That is the reason why software libraries have an advantage in performance over annotated programs in some applications. Using such libraries would help alleviate the overhead of testing.

[1] http://www.openacc.org/About_OpenACC

Moreover, algorithms covered by such libraries need not be programmed and annotated for parallelization.

Recent additions to OpenACC standard, namely version 2[13], allowed function calls from within OpenACC compute regions, which are regions to be executed on the device. These additions gave OpenACC programmers, in addition to modular programming, the ability to call components already parallelized for the device. Libraries which have a device interface now can be called, allowing modularity to be extended to thread and gang level granularity.

However, using such libraries introduces challenges when their use is needed inside code that needs to be automatically parallelized in frameworks such as OpenACC. To help alleviate one class of difficulty associated with this problem, we propose the implementation of a wrapper generator with hidden abstract data type management for C-based device library interfaces. We also implement such generator for one renowned parallel GPU library (CuBLAS). We show that the approach of automatic wrapper generation would be useful both regarding performance and ease of programming.

1.1 GPU Hardware architecture overview

Since GPGPU started getting focus from GPU users and manufacturers, GPU architectures and their configurations had gone through some changes in the last decade, and are still expected to keep going through such changes as long as there is room for improvements.

Elements of single CPU systems interfaced to GPUs can be illustrated by the diagram in Figure 1, while Figure 2 shows a symmetric multi-CPU system interfaced to a GPU. The PCI Express (PCIe) bus consists of lanes. Each lane provides a certain bandwidth, and peripherals can have a configuration of one, four, eight or 16 lanes where the bandwidth of communications add up. For example, each lane in a PCIe 2.0 bus provides a bandwidth of 500 MB/s, which adds up to provide 8 GB/s.

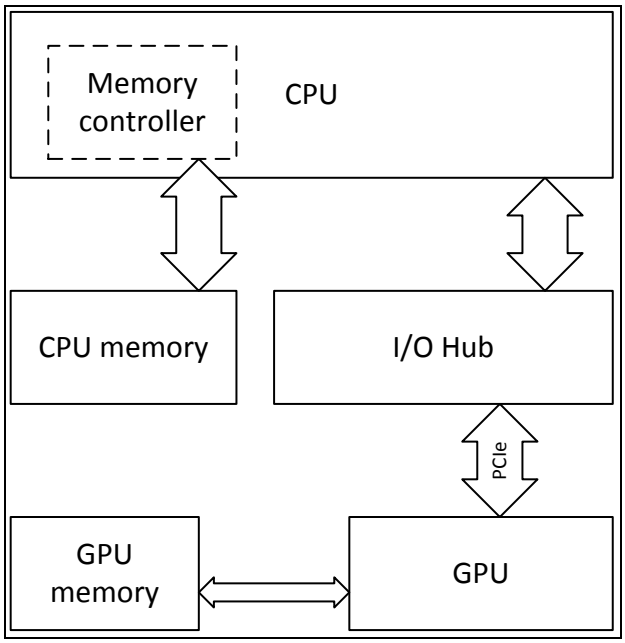

Figure 1: (Redrawn up from [20, Fig. 2.4]) An illustration of a single GPU - single CPU system

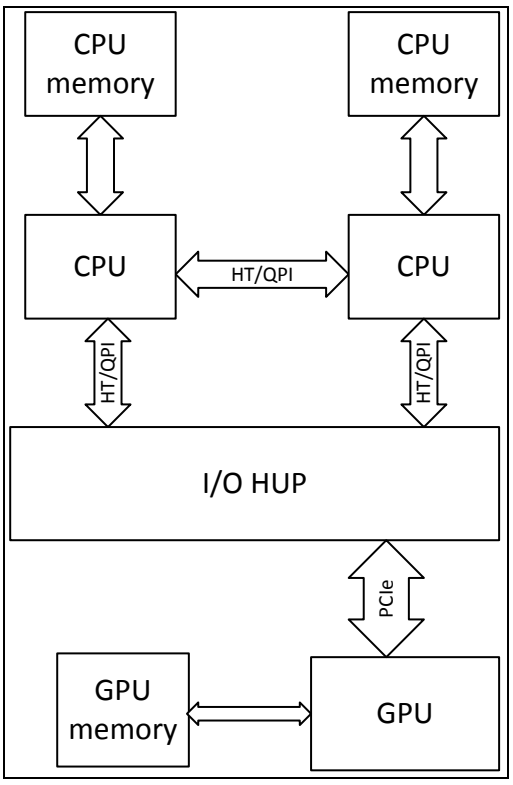

Figure 2: (Redrawn up from [20, Fig. 2.5]) A modern multi-CPU, single GPU system configuration

1.2 GPU programming model

In the past decade, the field of high-performance computing has been rapidly changing. From one point, hardware accelerators have been pushing the boundaries of computational discoveries due to their characteristics that incorporate low-cost, energy efficiency, and performance[3]. Graphics processing devices, when designed and operated to target general purpose computing have demonstrated impressive advances in computations science. Some experts believe that those systems would be the building blocks for future high-performance computing platforms[4]. However, the programming of these devices is yet to mature enough to a state that allows developers easy access to the performance of these devices. That's why developing mature, efficient and easy-to-use programming models are vital to the success and longevity of these architectures.

Some languages and programming tools targeted GPUs to make GPGPU accessible to developers. Extensions to programming languages were introduced such as CUDA[8], OpenCL[9] and BSGP [21] made GPU programming accessible to developers.

Most commonly used hardware accelerators that are used for general purpose computing use multithreading to put up with operations of time-consuming latency, which are mainly memory operations. As a result, application writers need to expose more degree of parallelism for this model to fulfill its intended performance goals.

Current Accelerator devices targeting high performance computing are constructed as IO devices, where their physical memory is separate from the host`s physical memory. That makes memory management an important part of achieving high performance[22].

Figure 3 below shows an illustration of the programmer's view of a machine with a hardware accelerator. It illustrates memory separation between CPU and GPU, the existence of multiple cores in the hardware accelerator, and multi-threading in each core. The illustration shows the programmers view of a machine containing a hardware accelerator, where a PE is a processing element. Software cache is a manually controlled cache (scratchpad) memory while the accelerator's hardware automatically manages the

hardware cache. Notice that there is a possibility of adding a second level cache, which is indeed the case of recent generations of GPU architectures; we incorporate awareness of the second level of hardware cache in Chapter CHAPTER 4.

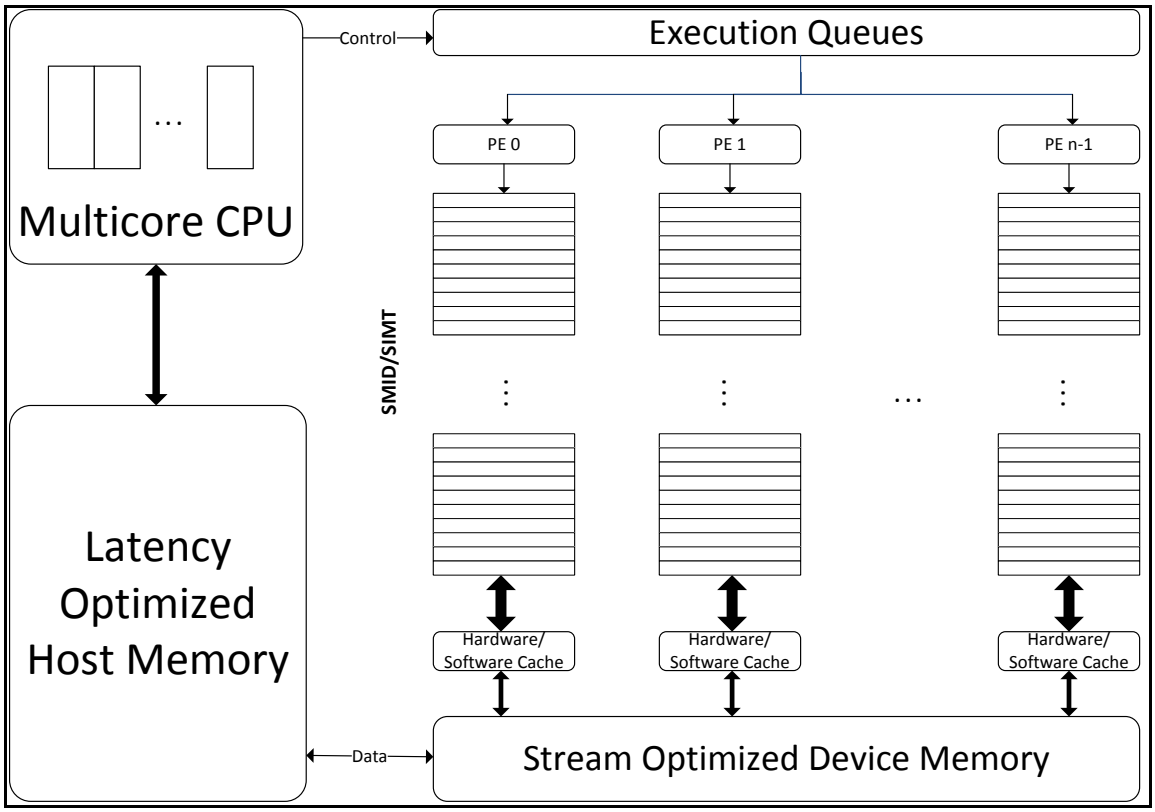

Figure 3: (Redrawn up from [22]) An illustration of abstract machine model of a hardware accelerator.

Usually, programming models for GPU accelerators present an abstraction that groups multiple threads together sharing scratchpad memory and a processing element`s cache. These groups are called gangs (blocks in CUDA and gangs in OpenCL).

In CUDA, a multiprocessor is referred to as an SM, while a gang of threads is usually called a block. The whole combined structure of blocks/threads is known as a grid. A subgroup of threads that executes concurrently is called a Warp. A block could be made from multiple Warps.). Blocks are expected to be larger than a single warp to tolerate latency operations via switching among these WARPS for each Multiprocessor. Nevertheless, multiple blocks can be assigned to each SM if the resources permit. In the

rest of this work, we will use CUDA terminology as we are working using CUDA and related technologies.

Because of the architecture and design of GPU hardware accelerators, they are most suitable for data parallel applications [23]. This suitability is because GPU hardware accelerators lack a direct path between processing elements to share data between them.

GPUs are a form of throughput based high performance computing. It hides non-computational latency via fast context switching among threads. Each processor in a GPU accelerator device is a usually capable of running a massive number of threads. NVidia calls these processors as SMs or SMXs, depending on generation. GPUs usually have a parallel stream optimized memory shared among GPU processors, which NVidia calls global memory. A much faster, smaller and parallel memory also usually is present, which is shared between threads (cores) within each single GPU processor (called shared memory in NVidia terms).

CUDA is an extension to C programming language that makes it able to target NVidia GPUs. Execution on GPU starts when a section called the kernel is launched during CPU code execution. Kernels define the number of threads and blocks to be executed in that launch instance. The hardware scheduler assigns a block or multiple blocks to GPU multiprocessors, Context switching between warps inside blocks that are assigned to each processor is hardware based and fast.

From a programmer's point of view, A CUDA kernel is usually initiated by the code running on CPU by calling a function preceded by the __global__ CUDA keyword. The code written inside the function is similar to an ordinary CPU code in the sense that each thread executes the same function. However, since each thread has its own private space in register file and memory, the same variable name could have different value for each thread. A thread index stored in a register is usually used for data decomposition of arrays over different threads.

Threads in a single block are executed in groups that NVidia calls Warps; some processors are capable of pipelining operations from several warps together. Operations in a single warp of threads happen at the same time. In the case of threads in a single warp are assigned different operations, those operations are serialized in a way that makes threads stall while other threads in the same warp execute an operation that stalling threads were not assigned to do.

Threads launched in a kernel are grouped into cooperative thread arrays called warps that are executed collectively as a unit in a GPU processor. Threads within a warp have the ability to synchronize and communicate with each other in the fastest manner using CUDA system calls. Several warps of threads are grouped into a block and are assigned to the same processor, which does contect switching between them. A GPU processor could be able to manage several blocks of threads concurrently (using warp context switching across different blocks). Synchronization between threads within a single block is possible using CUDA system calls but with a slight additional performance penalty. However, threads within a kernel or different kernels cannot synchronize or communicate using CUDA system calls. The CUDA call used for synchronization is the *__syncthreads*() system call. however, each thread in the block have to individually or collectively (with other threads) reach this synchronization function call and execute it in the control flow of the program. This is especially important when the control flow conditions depend on thread indices.

1.3 Problem description

Since programming hardware accelerators are considered complex and error prone [11]; Several solutions to abstract and ease their programming model have been developed[11], [24], [25]. However, there is still a performance gap between the compiled code of these abstractions and manually tuned parallel code, which would cause underutilization of the performance capabilities of these hardware accelerators.

There have been several attempts to alleviate this problem. However, these approaches vary in their way of implementing the specific optimizations targeting GPUs. It is tough to compare their performance and their level of optimization: the user is left without real information of the effectiveness of compiling his code using a specific compiler. Users should be adequately informed to select an optimized approach for programming these machines. There are many fundamental issues in the optimization of code targeting these devices that are hard to implement and to access even in recent GPUs. Our work aims at a thorough (systematic) exploration and performance comparison of major proposed compilers addressing (covering) a wide set of optimizations within the framework of the iterative Linear Algebra Solvers, and extended to other algorithms as fit. Through this study, we will propose methods for enhancing the optimizations applied by tools and compilers that deal with directed automatic parallelization of such serial code.

1.3.1 Goals

We base our work on the experimental analysis of the output of high-level constructs for automated compilers on GPUs like the OpenACC®. We analyze and introduce solutions that lessen the performance gaps between the compiled applications of hand tuned parallel programs of famous and well-known libraries (such as CuBLAS) and benchmarks (such as the EPCC benchmark suite) versus the compiled code of the abstracted languages and language extensions.

The strategy is to find out cases for which Directive constructs and automated compilers does not perform well on published results for libraries and benchmarks and our own optimized accelerated programs. Then to identify optimizations that will alleviate the weaknesses of these implementations. **The action includes proposing a set of practices and enhancing optimizations, the study of their effects on performance, and proposing corresponding directives when applicable.**

The following list defines the objectives of this research:

Objective-1: Scope of Applications - Compare the performance of OpenACC programs to hand-optimized programs, automatic or annotations-based generated programs for following applications GEMM, AXPY. Matrix addition, Matrix transpose, and some typical Linear Algebra Solvers.

Objective-2: Use of Analysis Tools - Extensively use run-time profiling techniques and parallelization reports for:

(1) optimizing application programs
(2) identifying underlying execution techniques in GPU and their details
(3) Analyzing and determining the conditions for maximizing run-time performance. This analysis is helpful to determine strengths, weaknesses, and un-optimized implementations in the studied subject programs.

Objective-3: Experimental Analysis - Analyze run-time application performance as described in objective 1 to evaluate and assess currently implemented optimizations in both CUDA using annotation-based programming as well as automatically generated code transformations. This includes the performance comparison of available code transformation systems versus the applied optimizations. It also involves the use of a few programming tools which are locally or remotely accessible.

Objective-4: Propose/Enhance Optimizations – Propose detailed enhancements to existing optimization techniques and developing new optimizations for improving the performance of application programs based on experimentally evaluating the efficiency of some existing programming practices.

CHAPTER 2
Literature Review

2.1 Introduction

While being a Ph.D. student at Stanford, Ian Buck introduced Brook as a language to ease the use of GPUs for general purpose programming[5] [26]. Buck demonstrated the language and wrote several applications with it, comparing performance to CPUs and among several GPUs. It did not take long since then for him to be hired by NVidia to lead the development of CUDA. CUDA and the later emerging standardized OpenCL (Khronos), allowed programmers to program the GPU without having to deal with actual graphics programming. However, they still put the burden on the programmer to exploit the architectural design of the underlying hardware to gain notable performance improvements.

GPU programming is introduced to help alleviate this problem. In the next section, we review the work done in that regard. In the section after that, we examine the history of parallel numerical libraries. Then, we review some of the work done optimizing the programs of GPU devices and on integration between directive programming and other technologies.

2.1.1 GPU directive programming overview

[27] Analyzed and evaluated several directive programming models including their own. In this overview, we will have a spotlight focus on their work due to excellent quality of their methodology and presentation. We also go through the authors' analysis and evaluation of their work and other directive programming models.

[23] Investigated what program optimizations can be done on a CUDA program to enhance performance. They have done an exhaustive search on optimization space for the kernel launch parameters. They have found configurations that are 74% faster than what was thought optimal before. The authors suggested manual methods for optimizing CUDA code, some of which became standard in teaching GPU programming.

The high burden of correctly exploiting the architecture of memory hierarchy for performance gains in GPUs; motivated [28] to introduce CUDA-lite. CUDA-lite takes a simple CUDA code as input, which would view the memory as an abstract solitary entity rather than a hierarchical one. The naïve CUDA code could be annotated with proposed extensions to maximize the efficiency of the transformation. CUDA-lite performs analysis on the annotated naïve CUDA code to observe opportunities to conserve memory bandwidth and reduce latency.it was implemented as a source to source translator that still produces CUDA code. Several annotations serve the programmer in CUDA-lite. CUDA-lite does not affect parallelization decisions. It only operates under the state of how the program got parallelized. CUDA-Lite performs the transformations such as inserting shared memory variables, loop tiling coalescing memory accesses and caching global variables on shared memory, which yielded 2 to 17 speedups in the author's experiments. CUDA-lite only focuses on memory access pattern. So, it is not a complete solution to optimizing CUDA code.

[12] [24] introduced HiCUDA also as a language based on directives for programming NVidia GPUs. HiCUDA stands for high-level CUDA. The authors intended an abstraction that closely matches CUDA model. They wanted a CUDA with new and simpler directives set. The goal for HiCUDA is not to automate optimizations; rather it is to make it easier for the programmer to program CUDA. For example, it provides simple directives to ease data transfers between CPU and GPU. Explicit optimizations are required for utilizing hardware features such as the constant memory or shared memory. HiCUDA does very few implicit optimizations. Namely, it tries to minimize the size of shared memory used based on the lifetime of shared memory variables. Also, they constrain the distribution scheme for loops over threads to be cyclical for memory

coalescing purpose. The lack of abstraction could sometimes be thought as a limitation due to a learning curve that is likely to be longer than platforms that provide implicit optimizations such as the later OpenACC. However, it also can be an advantage when the application needs optimizations specific to a certain underlying architecture [27].

[11] had proposed a set of directives to be added for OpenMP set of directives [29]. Their work would translate such annotated code into CUDA. OpenMPC suggests and implements an extension to the OpenMP [10] interface that targets CUDA GPUs. It naturally offers the programmers a degree of abstraction similar to the one provided by OpenMP. Thus, seamless porting is expected for most existing OpenMP programs into GPUs. The author's implementation is constructed on using the Cetus compiler framework[30]. Data regions are used in OpenACC and PGI model for setting up the order (boundary) of memory allocation and freeing. In contrast, environment variables are employed in OpenMPC for setting functions (or the whole program) as a data boundary. After that, data management is automatically done using context-sensitive, inter-procedural analyses. One feature of OpenMPC is that it allows low-level tuning for a set of optimizations, execution arrangements in addition to parameters specific to CUDA such as shared memory. The framework offers tuning tools suitable assistance for users to generate a wide set of optimization variants without the need for deep knowledge of the GPU architecture and programming models. OpenMPC also does optimizations via auto tuning. The auto tuner is a prototype that the authors built to analyze the program and optional user settings automatically. The implementation by the authors also included profile based optimizations, where the programmer provides manually the list of optimizations to be done; along with their parameters. Authors of OpenMPC asserts that a unique feature of their framework is the ability to do reduction operations on arrays in addition to scalars[27]. Array reductions are usually implemented in OpenMP via critical sections. OpenMPC detects automatically such behavior while transforming OpenMP code and creates a critical section free array reductions for the GPU. They also allow arrays in addition to scalars in explicit reduction statements. Indeed this feature is considered unique until this day as far as we know for annotation based compilers that work on C language.

The PGI Accelerator programming model [14] The PGI Accelerator was one of the bases that OpenACC standard was built upon. It allowed minimal directives marking a code region to be inserted into a serial code to transform it into a CUDA-enabled GPU. It also allowed other directives to be added such as ones to specify array bounds (which is mandatory in C's pointer-based arrays) in addition to directives guiding the choice of a loop's parallelism type to be applied. A limitation in PGI's accelerator model considered in the literature [27] is the fact that programmers have little control over optimizations applied during transformation. In addition, the implementation of PGI limits the number of loop nests to a maximum of three. Other limitations viewed in the literature include the dependence only on automatic detection of reduction clauses instead of allowing an explicit one; which limited the capabilities of reduction operations to basic ones. Another limitation was disallowing pointer-based arithmetic. OpenACC is a standardized set of directives based on par with PGI's model. OpenACC adds other directives that were not available in PGI's one, such as the parallel construct the forces transformation of a code region into a single GPU kernel. OpenACC has three levels of parallelism while PGI had only two. Moreover, OpenACC had an explicit reduction clause. Since its introduction in 2012, OpenACC standard is being constantly enhanced. The last version at the time of writing is OpenACC standard version 2.5. lee et al.'s [27] view on the limitation of OpenACC is the same for PGI's: it does not allow much control over various optimizations and does not provide the architecture specific features and ambiguity in handling the distribution of levels of parallelism into loop nests.

There are several implementations of OpenACC, namely there was an implementation from CAPS built upon their own HMPP [31], it supported many targeted architectures and output languages; such as OpenCL [9] and intel's MIC [32]. There is the accULL[25] for which the authors reported very promising results [33]and is providing it for free. However, it is still not completely implementing the OpenACC standard.

A comparison between various implementations of OpenACC was made by [25]. Although these implementations are rapidly being enhanced and new versions have been released after the particular time. However, this comparison would serve as a guide.

Another model that is based on directives is the HMPP [31]model. It was built and implemented by CAPS Company. It targeted CUDA, OpenCL and MIC architectures. It allowed simple auto tuning using a user provided search spaces for certain variables. One outstanding feature of the HMPP model is targeting based on calling context and that it allowed abstract control for architecture specific arguments related to memory structures and control over loop transformations and mappings. However, optimizations in the HMPP model were only explicit; Meaning they needed some knowledge to be applied. It shall be noted, however, that CAPS®, the company that owns the HMPP model has vanished in the year 2014, making the future of the model to be uncertain.

A comparative study where made in the literature between HiCUDA, PGI, and accULL[33] .all code used is manually converted from OpenMP. PGI clearly sets the record for larger problem sizes. Something hard to understand is why HiCUDA would perform worse at large problem sizes since HiCUDA has the least abstraction of CUDA (i.e. lowest level).

R-Stream [16] is a high-level, architecture-independent programming model that is centered around the polyhedral model[34]. It extends its targeting beyond CUDA GPUs into several architectures such as Tilera CPUs. In order to be mapped into GPU, a region of code must operate on a dense matrix or array structure with affine boundaries and affine loop index. A user only has to define valid regions to be transformed to the GPU. The fully automatic capabilities for R-stream are impressive since even data transfer optimizations do not need a directive to operate correctly. Also, R-stream provides strong tiling optimizations. Finally, R-stream allows seamless portability across different architectures, including ones other than hardware accelerators. This seamless portability only needs a user provided architecture description without any changes to the code region markings or the serial code. During the time R-stream was created, it was compared to CuBLAS 2.0 using the matrix multiplication routine and achieved 72% of CuBLAS' performance.

Stream also provides the user with the capability to control various optimizations for loops and data movement for valid regions. However, the condition for requiring matrix or array operations with affine indexing and boundaries is too restrictive. Another drawback is data cannot reside in GPU across the regions. When added to the previous restriction, this means a lot of separate kernels with data movement for each, Which would be a drawback for performance.

CUDA-CHiLL[35], is a framework for performing source to source transformations to generate parallelized code optimized to run on GPUs. CUDA-CHiLL is built on top of CHiLL[36] also based on the polyhedral model [37]. CUDA-CHiLL accepts a transformation recipe (a sequence of commands). This sequence of commands provides an abstract strategy for code transformation. CUDA-CHiLL recipes have a level of abstraction that is higher than that of CHiLL as well as allowing the lower level CHiLL commands.[38]. Usually, a CUDA-CHiLL command is a fixed collection of several CHiLL commands abstracted in that single higher level command. The concept of transformation strategies is discussed in[39], and is also similar in concept to the annotations introduced in X language [40]. However, the Transformation recipes in CUDA-CHiLL are isolated from code; meaning that the code would be clean from changes and dependencies on specific architectures[38]. A mix of transformation recipes (strategies) is provided, such as tile-by-index; which tiles a loop using a given size. Another example is copy-to-shared; which caches array values that are accessed from a starting loop level in the on-chip memory in GPU. CUDA-CHiLL inherits from CHiLL An important feature [38]; which is the ability of the framework to target a wide variety of architectures. This is done using an architecture description provided to the code generation subsystem. Although CUDA-CHiLL depends on command based transformations commands, applying these transformations still needs some optimization heuristics. For memory hierarchy optimizations, the authors used adaptations from [41], in addition, The authors expressed that they don't use constant or texture memory; following [42]. In order to use CUDA-CHiLL, one must understand the necessary optimizations needed. The author of CUDA-CHiLL discusses the limitations in his thesis [35], he describes transformation recipes as "flat sequences of commands and

parameters" for the transformations. He describes the necessity for targeting specific architectures to offer diverse layers of abstraction for automatic handling of transformations for different input scenarios.

Khan et al.[43] [38] proposed and implemented a compiler that automatically transforms serial code into optimized CUDA code. It builds up on CUDA-CHiLL[35], by constructing a transformation strategy generator (TSG) and an automatic tuner that chooses the best tuned from a set of resulting variants. The TSG generates scripts for CUDA-CHiLL. The compiler takes as input a serial Code in C; and outputs GPU code and corresponding finest transformation strategy for CUDA-CHiLL. The optimization space that this work explores is the same commands that CUDA-CHiLL offers. The compiler system is composed of two components: a component that composes recipes (TSG), and the auto tuner. The system uses two subsystems (CHiLL) and CUDA-CHiLL. The author first points the impressive performance results of the optimization space pruner.it evaluates a maximum of 335 recipes, for the benchmarks. The pruner is successful in sparing the system the evaluation of about 97 percent of the data optimization and placement search space. The author also compared the performance of code generated by his system with manually tuned benchmarks and libraries for BLAS, multimedia, imaging and scientific domains. The author reported comparative performance results for various benchmarks against manually tuned Blas; and against CUBLAS 3.2 and MAGMA libraries. The generated code was outperformed in rare times by MAGMA and CuBLAS. However, authors had a performance speedup up to 1.9X from their work over CuBLAS 3.2. Overall, TSG generated code was reported to have comparable performance to other libraries and benchmarks.

Zhang & Mueller [44] introduced HiDP. They noticed that some annotations introduced in directive based approaches limit the optimization space and are applicable to only specific sets of algorithm types. HiDP allows users to mark a code in terms of task parallelism and data parallelism. The system then uses autotuning to find suitable switching points for the target application. HiDP performs memory optimizations to exploit shared and constant memory, loop unrolling and auto tuning to choose

experimentally among several variants of generated code. Authors' tests show that their system is capable of achieving better performance in Matrix-matrix multiplication over CuBLAS 4.2 for matrix sizes up to a certain size limit. Their system achieved better performance in several benchmarks such as stencil computations and very close results to the CUDA SDK counterparts in quick sort and particle simulation. It lagged behind the CUDA SDK version of the Bitonic sort in terms of execution time. The authors mentioned that HiDP is active in development, and the promised to release it as open source.

One common theme of the above mentioned models is that they all abide by the api limitations of the underlying programming GPU programming model (CUDA in case of NVidia architecture). For example, we did not see any of the models implementing a GPU-based synchronization that overrides the usual limitation of doing synchronization using the host CPU.

As the research for automatic parallelization and directive based approaches is ramping up rapidly, there are many approaches and implementations being introduced. This is similar to the pre-MPI and pre-OpenMP eras where there were needs for the solutions they provided. The research community and industry are introducing transformation approaches, language designs and different implementations for the task of automatic parallelization.

Other proposed designs and implementations that were not introduced in the survey for reducing space might equally promise. To name a few, there are GPU extensions[45] to unified parallel C[46], GPU extensions[47] [48] to Chapel[49] , Microsoft's AMP++ in visual studio that extends C++ for accelerators and MATLAB GPU computing toolbox.

2.1.2 Parallel numerical libraries

The use of libraries was proposed officially since the year 1968 as a part of the solution for software crises management [50]. It was felt that such software reuse would be useful

in numerical software among other types. The exact words of the authors about the workflow for using such components are listed below:

> *"He [the user] will consult a catalogue offering routines in varying degrees of precision, robustness, time-space performance, and generality. He will be confident that each routine in the family is of high quality - reliable and efficient. He will expect the routine to be intelligible, doubtless expressed in a higher level language appropriate to the purpose of the component, though not necessarily instantly compliable in any processor he has for his machine." [50]*

Now, after more than 40 years, this workflow is mostly the case how programmers are using third party libraries.

The need for efficient, reliable, and high-quality software components is even more important in parallel computing, as it requires more effort from the programmer to produce parallel software with such features. The current variety of parallel technologies contribute to this difficulty[51]; since porting parallel software between technologies is mostly not straightforward.

A lot of work and effort have been done to create parallel numerical libraries; many of which are freely available. Another very good job had been done to list freely available numerical libraries and compare their features on the web[2].

2.1.3 Optimizations

There has been a lot of work on enhancing automatic parallelization frameworks for GPUs.

---

[2] By Jack and Ahmed, 2015, http://www.netlib.org/utk/people/JackDongarra/la-sw.html

Liu et al. [52] Described applying a general data layout optimization strategy to improve the performance of CUDA code. Their method considers the data access patterns presented by the program. Their applied transformations are called data localization and data relocation. They claim a performance enhancement of 4.3X for their method when applied solely, and an enhancement of 6.1X when combined with loop optimizations from literature.

the target optimizations defined by the authors were Minimizing channel skewing in device memory (i.e. balance access to the device`s memory channels, reduce bank conflicts in shared memory); where accesses by threads in the same block that are executed in the same step are aligned and contiguous in device memory.

The authors apply two optimization steps: localization and data relocation. Data localization step is there to ensure that data accesses to data executed on a processing unit are confined to a part of memory rather than spread all over it. However, they state that their strategy won't eliminate all cases where multiple processing units access one data block.

The idea of data localization is grouping data in array in a way that instructions a processing unit (thread block on the course grain optimization level; or a thread on the fine grain optimization level) access a single group of data. The authors represent the localized data layout of an array as a set of data hyperplanes. They also represent loop nests as a set of parallelization hyperplanes, where they map parallelization to data hyperplanes through an equation (3.4 in their dissertation).

At data relocation phase, the strip-mining transformation is applied; where for each array index a_i, the new index in the transformed array is calculated using an equation. Array index permutation along with padding is then applied to consider the interleaved physical mapping from linear address to the memory bank.

For their work, the authors used PLUTO automatic parallelization framework. At compile time, they generate new access patterns discussed. Actual layout transformation

is performed on the fly at runtime. They applied 15 benchmarks from Pluto and Rodinia (listed in their table 3.1) using the default input sizes.

The benchmarks that benefited the most from applying all the layout optimizations were ADI and Seidel among other benchmarks. Not all benchmarks benefited from all optimizations due to the nature of these kernels (e.g. some of them already have a coalesced data access pattern). The average speedup was at 4.3X for applying all optimizations (ranging from 1.1X to 9.1X). Their optimization reduces the shared memory bank conflicts by 30.5%.

The work of Cohen et al. [53] tackles the optimization of tiling. Authors claim that their work is capable of tiling multi-statement stencils over time based and parallel loops; achieving excellent multi-level exploitation of the parallelism and local memory resources of a modern GPU. One approach to tiling is split tiling; which subdivides tiles into a sequence of trapezoidal steps of computation. This technique enhances data locality without limiting inter-tile parallelism. The authors propose a generic algorithm to calculate index-set splitting that enabled them to perform tiling for locality and synchronization avoidance. Their algorithm is able to make split tiling for any number of dimensions. In that paper, the authors present an original polyhedral approach for generating split-tiled code for GPUs. The authors Optimization target is enabling the use of shared memory by minimizing synchronization and increasing the amount of available instruction level parallelism (authors consider assigning multiple statements to 1 thread as ILP). Authors' idea here is to generate specialized code for full tiles as well as for tiles that intersect with the iteration space boundary (i.e., partial tiles). the authors extended the polyhedral GPU code generator PPCG [54] with the split-tiling technique.

Elteir et al. [55] quantified the impact of atomic operations provided by the system on AMD GPUs. They provided a novel software based method for atomic add that works better on memory-bound kernels by a 67-fold speedup. Their method consisted of using three arrays in a master-slave fashion and uses a wavefront method for updating calculations. However, their method is limited by the fact that the number of

groups to be executed shall not exceed the number workgroups that the system is able to execute concurrently.

Duarte et al. [56] proved that measurement of power and execution time shall be done over the whole applications rather than just on kernels, in order to give accurate comparative results. Their work focuses on comparisons between CPU-only systems and GPU systems. Using full application measurements per the study, one can conclude which systems are better suited for certain applications. They use two metrics for quantification, namely Energy-delay product metric and Performance per effort hours (PPEH) metric.

One useful listing in their work is a table showing applied optimizations in applications studied. We extract GPU-applicable ones and put them in Table 1 below.

Table 1: GPU software optimizations used in kernels studied by Duarte [56]

Optimization Name	Explanation
DST	Transformation of data structures; usually to improve the performance of memory accesses. One famous example is transforming AoS to SoA (data structure transformation)
PPB	Eliminating thrashing in memory copies (ping-pong buffering)
AP	Aligning matrix rows (in row-major storage) into appropriate memory or cache locations through padding rows with redundant data (Array padding).
SMC	Shared memory caching to reduce memory access
TC	Using texture cache for random access data
FM	Using optimized math functions such as fused add-multiply, though it

	reduces accuracy sometimes (fast math)
LT	Stores functions output in a look-up table to eliminate redundant calling.

Yang et al. [57], [58]demonstrated their work on constructing a source to source compiler based on Cetus [30] framework. For GPU optimization, they used vectorization optimization to group data accesses into a vector type access (although they claim that vectorization has no effect in NVidia GPUs), thread block remapping or address-offset insertion, tiling and unrolling, memory coalescing and data prefetching optimization. The produced code is claimed to achieve very high speed up over the processed input naïve code.

Qasem [59] introduced a framework to automatically apply an optimization called unroll and jam[60]. In addition to auto tuning parameters, optimizations in their work include thread coarsening, loop interchange, multi-level loop fusion and scalar replacement and distribution. Authors promised that more specialized optimizations are being added such as array contraction, iteration space splicing, and array padding on Fermi architecture. One important aspect of the work is using GPU heuristics to determine the profitability of applying mentioned optimizations, such as determining tile size and shape. They also allow the use of directives in the code for manual enforcement of optimization parameters. One important aspect of the work is using GPU heuristics to determine the profitability of applying mentioned optimizations, such as determining tile size and shape. They also allow the use of directives in code for manual enforcement of optimization parameters

Wu et al. [61] tested the operation of kernel fusion on database applications. They expected kernel fusion to result in increasing the compiler's optimization scope and smaller memory footprint since it eliminates the need for temporary variables for data flow between kernels: the first result would cause a reduction in Memory Accesses,

temporal Data Locality, reduction in PCIe Traffic, Larger Input Data. The second result would cause Common Computation Elimination and Improved Compiler Optimization.

The fusion operation happens through preserving data flow between the individual threads of kernels. They achieved some speedup in queries and proposed to extend the testing into other application domains.

Abdel Fattah et al. [62] achieved better performance than CuBLAS and MAGMA libraries on NVidia Fermi GPUs in symmetric matrix-vector multiplication kernels, which have the SYMV BLAS name. They proposed that thread blocks scan the matrix vertically for the transposed case of the multiplication (BLAS interfaces usually provide an option to transpose the matrices during certain operations). They also implemented their own thread-level buffering and prefetching strategy.

Leback et al.[22]Compares three accelerator architectures from the prescriptive of a compiler writer. Namely, they compare NVidia tesla, AMD Radeon, and Xeon phi. They expect that the OpenACC standard has the criteria necessary for long term success. The criteria explained by them is its interoperability with major parallel frameworks (OpenMP and MPI), allowing programmers to control memory transfer, virtualization of the parallel architecture and finally its design that allows portability of implementation and performance among different architectures. Authors provide a comparison between architectures that we found the best way would be to be summarized in Table 2, where we construct a comparison of the programming paradigms of accelerators available for consumers. The table sheds light on various important architectural aspects of the most common hardware accelerators and how the respective programming models manage these aspects. There are three options for a programming model to take advantage of features in the architecture, either the programming model virtualizes the feature, uncover it, or hide it. Automatically managing the feature is needed to hide it; while virtualizing a feature could be done by uncovering its presence but providing virtualized details. Uncovering a feature means exposing the user to the features details; which requires tuning and optimizing its parameters. Notice that uncovering a feature paves the road for

the user to increase performance, while feature virtualization provides the user with reduced cost and better portability. The compared models are Microsoft's C++AMP [89],FORTRAN, OpenCL [88], Intel's offload directives [29] (same as the OpenMP 4.0 target directives [90] that got introduced later) CUDA C, and OpenACC [91].

The authors provided an abstract descriptive illustration of the CPU+accelerator machine model which was already shown in Figure 3, as it demonstrates the abstract view of a heterogeneous machine architecture.

The authors conclude with the criteria for success of a programming model, which is being high level enough to allow targeting several different architectures, while still meeting important aspects of parallelism listed below:

- Portability of efficiency.
- Allows viewing the platform as an integrated CPU+accelerator platform.
- Being built around the standard C/C++/Fortran (in contrast to non-standard) without compromising much performance.
- Integration with MPI and OpenMP.
- Permits incremental porting and optimization of existing applications.
- Does not inflict much disturbance on the usual development cycle.
- Interoperability with low level accelerator programming models (CUDA, OPENCL).

Table 2: A comparison constructed for the various means to program for today's accelerators as discussed in [22].

Common Accelerator	CUDA	OpenCL	C++ AMP	Xeon phi	OpenACC
Memory management	EXPLICIT allocate and copy	Same as CUDA, but hides the exact placement of the buffer during different execution points.	Uses *views* and *arrays* of data. A **view** hides the exact placement of the buffer during different execution points. The programmer has the option to be exposed to explicit arrays	Provides the option for a completely hidden memory management. Also, provides a statements to allocate and transfer memory between host and accelerator	
Parallelism Scheduling	3 levels of parallelism (grids, blocks, threads) Mapping of blocks to hardware cores is hidden		Single- flat level of parallelism (with an option for explicit tiles)	SIMD is implemented with loop vectorization	3 levels of parallelism Explicit mapping possible, but compiler-implicit mapping is also possible

Common Accelerator	CUDA	OpenCL	C++ AMP	Xeon phi	OpenACC
Multithreading	Encourages over subscription (creating more threads than the number of cores). scheduling Parallelism among the blocks is hidden		Completely hides how parallelism is scheduled.	The programmer is exposed to the requirement to occupy the device with enough threads, and adding extra factor of them for latency hiding	Addressed via worker parallelism Can also use oversubscription
SIMD operations	Hidden from user			Classical SIMD vectorization within a thread.	multicore plus vector parallelism
Memory Strides	Programmer must deal with memory access pattern for consecutive threads		Mapping is virtualized, programmer can guess what iterations will execute together when the loop has one dimension	Only needs a stride-1 access.	Programmer should use stride-1 access

Common Accelerator	CUDA	OpenCL	C++ AMP	Xeon phi	OpenACC
Caching and scratchpad memory	Scratchpad memory Managed explicitly		No explicit scratchpad but exposes a tile optimization	No scratchpad memory	Implicit via compiler Also, there is a cache directive.
Portability	Reasonable portability across NVidia devices Bad portability for multicores	Designed for language and functionality portability	Performance portability	Offload model does not intend to provide portability to other targets	Intended to provide performance portability

2.2 Concluding remarks

In this chapter, we reviewed GPU programming model and some significant optimizations proposed in the literature. It is of note that Compiler Engineering's priority is Correctness of the generated code, the aspect of optimized code comes from extensive knowledge of target architecture combined with wide knowledge of code optimizations related to the problem domain. Moreover, different hardware generations of the same architecture need different optimization parameters and sometimes even need different optimization strategies. While extensive knowledge of the target architecture is usually best left for the compiler engineers, it is not the case for extensive domain knowledge; which is in the hands of developers. That's why it takes a considerable amount of time for optimizing compilers to perfect strategies for best code optimizations for new technologies. Moreover, compiler designers rarely consider optimizations that do not yield decent performance increases relative to effort spent.

Sometimes we find that it would be hard to automatically come out with perfect parameters for optimizations that would boost performance. For example, until recently, PGI OpenACC did not perform tiling optimizations on OpenACC programs. Recently, however, a tiling statement that is found in the openACC standard is implemented but need to be explicitly invoked along with tile size parameters.

In remaining chapters, we discuss code enhancements and optimizations that could be incorporated in automatic parallelization frameworks.

CHAPTER 3
Enhancing OpenACC Performance with Enablement and Use of Device Routines

3.1 Background

Porting libraries between different technologies, or using them in a code that needs to be compiled with compilers other than the ones they were intended to be compiled with or simply using them with older or newer versions of their intended compiler might introduce some challenges. For the reasons above, reusing components in GPU programming is necessary to reduce the effort and cost needed for designing software that utilizes this technology.

An important aspect of software design is interoperability; which is the ability of software components to interact with each other; especially those which are written in different programming languages [63]. In this work, however, we address the interoperability of various technologies in the same programming language. An example of interoperability in this context is the interoperability between OpenACC and CUDA technologies.

OpenACC is a technology to automate the process of parallelization and targeting of GPU parallel architectures. It is a standard that allows a compiler to use automated parallelization techniques to parallel architectures using serial code. The standard allows the user to leave all key choices for the compiler to decide, which can be accomplished in C/C++ programming language version of the standard by preceding a code block with the kernels directive. This process leaves the choice of parameters for the parallelization to the compiler; where it provides the most warranty of correctness for the parallelized code. The efficiency of parallelized code, however, is not guaranteed to be accomplished as the parallelization process is completely automated. Compilers give priority to guaranteeing

correctness over efficiency. Losing efficiency in this approach is expected as the compiler lacks the domain knowledge of the problem being programmed; which is usually held by the programmer. The performance of directive-based programming like OpenACC can be as good as the compiler and auto-parallelization technology could withstand. This is particularly the case for the newly emerging GPU-based parallel architectures as they have many parameters to tune. As a result, manual tuning and optimization are always preferred for gain the most performance, especially for critical applications if the time allows.

The use of software libraries has recently become an essential part of software design since they reduce development cost, expertise needed by developers, needed domain knowledge related to the problems solved and the overhead in software testing. This concept takes a particular importance in the domain of numerical computations and simulation, as computations in these domains share a lot of well-known algorithms and methods. Enhancing the performance of numerical computations needs a deep understanding of algorithms and computer architecture involved. Using renowned and production-grade libraries in this domain would usually yield better performance for a lower development and maintenance cost.

Based on the above discussion, it is advisable to use software library components whenever it is available if it does not contradict the goals of the programmer (such as introducing an unacceptable penalty in cost).

One example of production grade libraries is NVidia's® CuBLAS library [18], which we will use here as a demonstration case. CuBLAS is NVidia's implementation of BLAS [64] on GPU's. BLAS (Basic Linear Algebra Subroutines or Subprograms) are a collection of well-defined routine interfaces that provide to perform basic vector and matrix operations. They consist of several levels of work complexity to perform scalar, vector-vector, matrix-vector and matrix-matrix operations. BLAS operations are commonly used in the development of linear algebra software[65]. CuBLAS library is a library manually programmed by experts in the field of numerical algorithms and parallel

programming for the GPU architecture. Some parts of it are also having the parallelization parameters automatically tuned for the best performance possible. CuBLAS library is one of the renowned ports of the BLAS library into NVidia GPUs. Observers of computationally oriented research in any discipline would notice that an increasing number of researchers either comparing their work to CuBLAS performance or using its components. Since CuBLAS is a production-grade library with reputability; it is a good idea to reuse its components when needed.

In CUDA [8], constructing functions or routines involves a choice of making a routine callable from code executing on CPU, or from code already executing on GPU. There are certain keywords used to instruct the compiler to restrict the client of a function to be either a CPU code, GPU code or any.

CuBLAS[18], along with few other libraries, provide an additional interface called the device interface that allows calling library routines from within code segments that are already being executed on the GPU device (compute-regions). This concept allows any thread or a block of threads that need services offered by the library to call these libraries from within the device code. Such calls cause a new kernel launch with new set of parameters to be launched in a notion that NVidia calls “Dynamic parallelism” [66]

Rennich et al. [67] Found that when using CuBLAS for batched double precision matrix multiplication, better performance can be achieved for a batch of 8192 matrix-matrix multiplications by dividing the batch into four equal batches and executing them concurrently on the device. This type of decomposition would be straightforward when used from within automatic parallelization framework such as OpenACC. However, mixing library usage with automatically parallelized GPU code is not always a simple process; as it is customary for commercial libraries to require arguments referring to data structures with hidden implementations.

We noticed that many libraries follow a scheme in their interface implementation that prevents them from being called from within OpenACC compute regions. For example, CuBLAS host interface is callable from within OpenACC, but the calls cannot be made

through the device interface from within an OpenACC compute region at least up to the version we were using (version 14.10 of PGI). Hence, using CuBLAS in this scheme introduces the penalty of going back to a CPU-executed region to be able to call a library component; which might force unnecessary synchronization overhead. Moreover, using libraries such as CuBLAS in such a scheme with OpenACC introduces the penalty of not being able to choose freely the granularity of parallelism in the client calling the code and hence, the decomposition scheme. As the granularity level gets smaller, more independent tasks are created, hence enabling better scheduling and load balancing. For further explanation, if a decomposition scheme requires a call for vector addition by each participating block, such call will not be able to be accomplished as CTAs cannot call such libraries from within OpenACC Compute regions. In such cases, the data decomposition scheme shall change in a way that a single or multiple calls for that step to be accomplished from the CPU to CuBLAS' host interface. Thus, this would be considered a loss that hinders performance and increases programming complexity as we will demonstrate later in this work.

Rennich et.al [67] tested for the difference in execution time when using the device interface of CuBLAS versus using the host interface. Their graph [67, Fig. 5] shows the result of an experiment comparing the iterative use of CuBLAS host interface versus CuBLAS device interface for the GEMM matrix multiplication routine. This experiment was done using CUDA for small matrices and incorporated memory copies for the host interface timeline. Using device interface had shown superiority in performance regarding execution time due to both the need for memory copies and the overhead of communicating kernel launches through the CPU. This result is absorbing for iterative applications that need to iterate over some BLAS operations.

Another test done by the authors[67], was for concurrent batched kernels on GPU for small matrices. The authors tested calling CuBLAS' batched matrix multiply for a single batch of 8192 small dense matrices; then they tested calling the function four times concurrently for four batches of 2048 separate batches. Their test shows superior performance the concurrent run of four batches as shown in their graph[67, Fig. 6]. The

authors reported a speedup of 34X for 64 concurrent batches of 128 matrices each versus a single batched call for 8192 matrices in the double-precision matrix multiply case. The reason was the lake of multiprocessor utilization in the single-batch implementation.

The first version of the OpenACC standard did not allow branching into a code region that is outside of compute regions, which limited the scope of software component reuse to routines that could be inlined by the compiler inside parallel regions. The success of routine inlining depended on considerations by the compiler such as the existence of function calls inside the inlined function.

In the recent OpenACC standard [13], the routine directive was introduced. The routine directive is described in the OpenACC specifications as follows:

"*The routine directive is used to tell the compiler to compile a given procedure for an accelerator as well as the host.*"

That directive would tell the compiler to parallelize a routine to target the GPU device. Hence it should be possible for implementations of that standard to call device routines from within OpenACC' compute regions. This ability, theoretically allows the programmer to call any device routine from within compute regions; including libraries that provide a device interface. As CUBLAS library one of the very few libraries that provide a device interface when applying this concept to the CUBLAS library.

The OpenACC standard also introduced the ability to use *seq*-clause along with the routine directive. When the ***seq*** clause follows the routine directive, the compiler understands that it does not need to parallelize the routine. In such case, the calling thread will sequentially execute the routine.

Rennich's findings are significant in our context as OpenACC supports distributing a loop over several groups of threads (gangs). Hence a developer can easily write a loop that calls the device interface of CuBLAS for the batched matrix multiply as an example by writing a loop that loops through smaller batches of dense matrices and calls batch

matrix multiply on them by the device interface; which will cause simultaneous calls for them.

The behavior defined for the *seq*-clause is useful in our context as the libraries that we are focusing on either support launching new kernels from called routines, or provide services that are sequential in nature to be called by each thread.

3.2 Enabling the use of CuBLAS device interface from within OpenACC

In this work, we focus our on CuBLAS library for demonstration. Interested readers can see the conventional way of using OpenACC with CuBLAS' host interface in [20]. For our demonstration, we will use the same example function call; the SAXPY operation [21]. SAXPY scales a vector and adds it to another vector type; SAXPY is an abbreviation for single precision vector scaled addition – an operation very common in numerical computations. Suppose we want to add two vectors using a CuBLAS function. In version 1 of CuBLAS interface; it would theoretically be straightforward. Unfortunately, using CuBLAS through the first version of the interface is only possible from host code. To call CuBLAS GPU interface from a GPU CUDA code region (not an OpenACC one), one needs to call the second version of CuBLAS interface the same way as it would from within a CPU code region. However, CuBLAS second version of the interface introduced the use of handles to enable concurrency. Handles in such a case were not possible to be used from within the implementations tested by us of OpenACC due to it being an opaque data structure in most libraries. To allow such a call to happen from an OpenACC compute region, the use of handle data type shall be eliminated. One proposed solution for elimination the handle use from inside OpenACC is to introduce an intermediate layer as shown in Listing 1 below. : In the first cell, we show the code that would theoretically work according to the OpenACC standard; but it does not due to the reason explained in the paragraph above. Lines 7 -17 show a manual wrapping

mechanism to enable calling CuBLAS' GPU device interface from within OpenACC. Lines 18-21 show how to call CuBLAS' GPU device interface using the new method

Handle usage usually has a consistent pattern among each library. In the example of CuBLAS library; handle usage is consistent among functions in a way that makes it possible to generalize a solution and automate it. Well-known widely spread libraries usually have the advantages of being validated and efficient not to mention enabling fast prototyping and boosting program performance. Our work aims at promoting the use of such capabilities provided by libraries by users of automatic parallelizing compilers in the most performing methods.

Currently, the only mature implementation of OpenACC available is the PGI compiler; which will be considered. In this chapter, we introduce AWCUBLAS, an interface for the famous CuBLAS library that enables the use of this library's device routines from within OpenACC compute regions from versions of OpenACC implementations that does not implement CuBLAS device functionality. Thus, libraries such as CuBLAS that provide a device interface needing opaque parameters shall be easily callable from OpenACC' compute regions using OpenACC routine directive.

The remainder of the chapter is organized as follows: Section II explains the methodology. Section III provides performance and results, and then we conclude the chapter in Section IV.

Listing 1: Code example showing how to call CuBLAS device interface from within OpenACC compute region.

```
                    The client code that should theoretically work
01  #pragma acc parallel num_gangs(1)
02  {
03        cublasCreate(&cnpHandle);
04        cublasSaxpy(cnpHandle, n, ptr_alpha, x, 1, y, 1);
05        cublasDestroy(cnpHandle);
06  }
```

```
        The Solution proposed for a wrapping mechanism to act as a device routine
07  extern "C"
08  __device__ void invokeDeviceCublasSaxpy(int *returnValue,
    int n, const float *const d_alpha, const float *const d_X, float *const d_Y ){
09      cublasHandle_t cnpHandle;
10      cublasStatus_t status = cublasCreate(&cnpHandle);
11      if (status != CUBLAS_STATUS_SUCCESS){
12          *returnValue = status;
13          return;                          }
14      /* Perform operation using cublas */
15      status = cublasSaxpy(cnpHandle, n, d_alpha, d_X, 1, d_Y, 1);
16      cublasDestroy(cnpHandle);
17      *returnValue = status;           }
                          The client code that works correctly
18  #pragma acc parallel num_gangs(1) pcopy(x[0:n])
19  {
20      int status;
21      invokeDeviceCublasSaxpy(&status, n, ptr_alpha, x, y);
22  }
```

3.3 Methodology

Grillo and Reyes [22] compared the performance of several OpenACC compilers on two generations of NVidia GPUs. They also compared OpenACC implementations versus hand programmed CUDA on Kepler architecture. The comparison used EPCC benchmark suite [23] for the OpenACC code which we also took a subset of for our own comparison. In our work, we compare OpenACC implementations to CuBLAS instead of our own CUDA code.

Our method consisted of four steps. First, we benchmark OpenACC implementations with respect to the hand-programmed CuBLAS library. Some selected microbenchmarks were used to assert that well-done handmade coding performs better than auto-parallelized code. Second, we manually programmed a layer for the SAXPY [21] micro benchmark as a demonstration of the ability to accomplish the calls to CuBLAS from OpenACC compute regions through our layer. After that, we automated the process of generation for the whole AwCuBLAS layer from header files. Automation would make the work useful for future CuBLAS versions and opens up the possibility of generality to other libraries that have this problem. Lastly, we tested the layer for two applications to show the improvement regarding execution time and code size.

To explore the applications that might benefit from using GPU device libraries; we focused on benchmarking the performance of a subset of the algorithms that CuBLAS library implements versus the same algorithms coded using OpenACC. These experiments would allow us to test for possible benefits we would get from using a library such as CuBLAS over the automatically parallelized code. For the popularity of certain matrix operations in scientific applications, we chose the micro-benchmarks of matrix transpose, matrix addition, and general matrix-matrix-multiplication. For the OpenACC implementation, we adapted the code from the EPCC benchmarks [23]. Experiments in this section were done using PGI 13.2 OpenACC implementation, accULL 3.0 OpenACC implementation [24] and CuBLAS library`s version 5.5 implementation. We run multiple experiments on servers having NVidia Kepler k20c. Although we used accULL version 3.0 [24] as one implementation of OpenACC in the benchmarking phase to compare performance, we did not proceed with it in other experiments as we need an implementation that supports version 2 of the OpenACC standard to support the *routine* directive. Moreover, we discovered that accULL did not always compute correct results when we tested it on experiments unlisted in this work such as traditional Jacobi linear system solver and matrix-vector multiplication.

Benchmarking experiments were conducted on square matrices, repeated ten times each. The results were calculated by averaging the timing of the ten runs.

The first benchmarking experiment was matrix transpose. Transposing a matrix is the operation of switching matrix elements around the main diagonal. The OpenACC implementation for a square matrix transpose is straight forward and done by copying rows in source matrix into columns in destination matrix as shown in Listing 2 below.
Line 03

Listing 2: OpenACC implementation of matrix transpose.

```
Input: Square matrix A of size n×n
Output: square matrix C of size n×n

01  #pragma acc data copyin(A[0:n*n]), copyout(C[0:n*n])
02      {
03  #pragma acc kernels
04  #pragma acc loop independent
05          for (i = 0; i < n; i++) {
06  #pragma acc loop independent
07              for (j = 0; j < n; j++) {
08                  C[i * n + j] = A[j * n + i];
09              }
10      }
11  } /* end_data */
```

The second experiment in benchmarking was matrix addition; where two matrices are added together and stored in a third matrix location. The OpenACC code used is shown below in Listing 3.

Listing 3: OpenACC implementation of matrix addition.

```
Input : square matrices A and B of size n×n

Output: matrix C of size n×n

01  #pragma acc data copyin(A[0:n*n],B[0:n*n]), copyout(C[0:n*n])
02      {
03  #pragma acc kernels
04  #pragma acc loop independent
05          for (i = 0; i < n; i++) {
06  #pragma acc loop independent
07              for (j = 0; j < n; j++) {
08                  C[i * n + j] = A[i * n + j] + B[i * n + j];
09              }
10          }
11      } /* end_data */
12
```

The third experiment was the GEMM matrix multiplication [68]. It is a known operation in BLAS [65] where two matrices are multiplied together, and then the result is added to the old values of the matrix to be used to store the result. Both operations (multiplication and addition) allow scaling parameters to be involved. OpenACC code is shown in Listing 4 below.

.

Listing 4: OpenACC implementation of GEMM BLAS. The code was adapted from EPCC benchmark suite [69]

Input: square matrices A and B, of size n×n

Output: square matrix C of size n×n

```
01  #pragma acc data copyin(A[0:n*n],B[0:n*n]), copyout(C[0:n*n])
02      {
03          double temp;
04  #pragma acc kernels
05  #pragma acc loop independent
06          for (i = 0; i < n; i++) {
07  #pragma acc loop independent
08              for (j = 0; j < n; j++) {
09                  temp = 0;
10                  for (k = 0; k < n; k++) {
11                      temp += A[i * n + k] * B[k * n + j];
12                  }
13                  C[i * n + j] = temp;
14              }
15          }
16      } /* end_data */
```

The next step was to prototype a solution for enabling the use of CuBLAS from PGI. We needed to do this because up to the versions that we used of PGI (14.10) it was unable to call CuBLAS device routines from with OpenACC compute region for a reason we explain shortly in the next paragraph. PGI promised to solve this problem for CuBLAS in specific in coming versions. However, the obstacle holding device interface of libraries from being called from within OpenACC compute regions remained the same when the condition discussed next applies. The method used here can be applied to those libraries as well. We chose a simple example; namely the SAXPY CuBLAS routine as we consider it the simplest CuBLAS call to test and profile.

The problem of calling a library that already has a GPU interface arises from the fact that some parameters used and passed to the interface are opaque pointers. For our example CuBLAS library, this parameter is the *CublasHandle* parameter, which is used to differentiate between multiple CuBLAS Library contexts [70]. This parameter is needed for all functions provided by the CuBLAS library hence preventing developers from calling those functions from within an OpenACC compute regions.

An initial test template code was created to implement desired behavior similar to the example shown in the figure above (wrapper example saxpy in the introduction.). This implementation was tested and compiled with calls successfully done from OpenACC. Later, we developed a solution called ***(libimorph)*** that takes library CuBLAS header files as an input and generates a middle layer GPU interface per desired specifications.

For our SAXPY routine example, the header library defines the function prototype as shown Listing 5 below.

Listing 5: CuBLAS library`s SAXPY function header prototype for the float data.

```
cublasStatus_t cublasSaxpy(cublasHandle_t handle, int n, const float *alpha,
const float           *x, int incx, float                *y, int  incy);
```

As seen in the prototype above, The first argument required is the CublasHandle, which has a setup and usage scenario addressed in manuals [71].

To make SAXPY callable from an OpenACC compute region, we will eliminate the CublasHandle argument from the call by constructing an intermediate function in CUDA that does not need the argument.

In a first attempt, we will just eliminate the CublasHandle from the interface and insert required setup and usage code inside our layer. For better usage, we put the original return type as a call by reference argument instead of a return value and convert it into integer too, as it is of type CuBLASError_t which is an enum. We noticed that the compiler had some difficulty also dealing with enum types inside OpenACC compute regions. We call this intermediate function invokeDeviceCublasSaxpy.

In the CUDA implementation of the invokeDeviceCublasSaxpy function, we would do the setup for a new CublasHandle use it in an actual call to the device interface of CuBLAS saxpy then destroy the CublasHandle created. This process would save the user from trying to use a pointer into opaque structure inside an OpenACC compute region. The intermediate function`s header is shown in Listing 6 below.

Listing 6: A possible prototype for an intermediate callable device layer SAXPY that acts as an intermediate layer calling CuBLAS SAXPY

```
__device__ void invokeDeviceCublasSaxpy(int *returnValue,
int n, const float *const d_alpha, const float *const d_X, float *const d_Y )
```

This function, with the adequate setup in OpenACC, is callable from within OpenACC compute regions. The CUDA implantation of the invokeDeviceCublasSaxpy function, one possible implementation is shown in Listing 7 below.

Listing 7: One possible implementation of the body of an Intermediate SAXPY routine.

```
01  cublasHandle_t cnpHandle;
02  cublasStatus_t status = cublasCreate(&cnpHandle);
03  if (status != CUBLAS_STATUS_SUCCESS){
04      *returnValue = status;
05      return;                          }
06  /* Perform operation using cublas */
07  status = cublasSaxpy(cnpHandle, n, d_alpha, d_X, 1, d_Y, 1);
08  cublasDestroy(cnpHandle);
```

The process for handling the production of a working executable is demonstrated in the following steps:

1. Define wrapper function
2. compile it with NVidia compiler
3. link using NVidia linker
4. Compile OpenACC code with OpenACC compiler
5. link with wrapper using OpenACC compiler

The invokeDeviceCublasSaxpy we created and compiled was tested and proved to be successful in our tests on PGI OpenACC compilers 14.7, 9 and 14.10.

However, we tested the code for concurrent launches from within OpenACC compilers and noticed that the invokeDeviceCublasSaxpy calls are not being always run in parallel inside the GPU. The profiling results are shown in Figure 4 below; where profiler visualization of the code execution shows that routine calls are running in sequence even though they were called using AwCuBLAS simultaneously from concurrent OpenACC gangs.

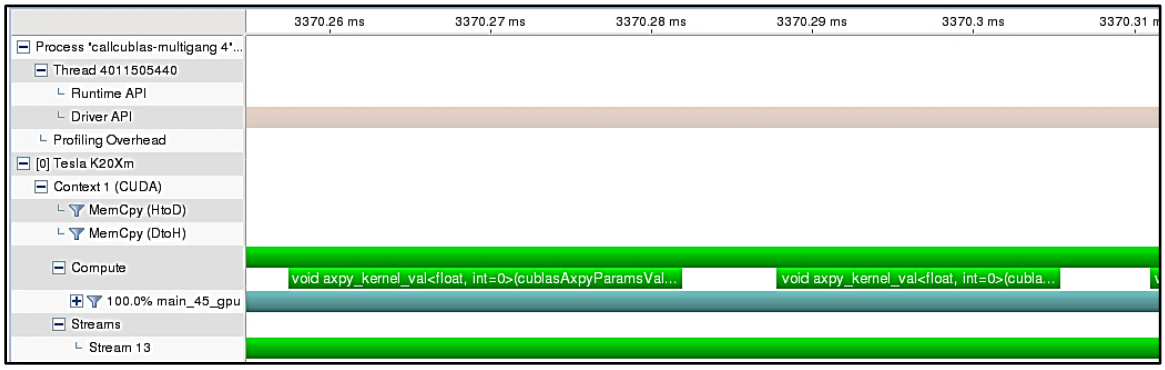

Figure 4: limited concurrency for concurrent calls in the first version of AwCuBLAS.

That experiment was conducted on a K20 GPU capable of running multiple kernels concurrently. However, we found out that calling the functions that construct and destroy a library handle from within each AwCuBLAS function call is limiting the concurrency of these functions. We mentioned above that our function implementation of AwCuBLAS uses these calls inside each function to eliminate the need for using the handles from within OpenACC compute regions; which are pointers to opaque structures.

We already expected an extra overhead to be present due to the embedding of a setup and closure for a CuBLAS Handle with each function call as shown in previous examples. However, the measurement of the result has shown a substantial setback in performance.

After analyzing the code runs and searching for the cause, we reached a conclusion that this behavior is due to the closure code for the CublasHandle used inside our intermediate implementation. The embedding of closure code for the CublasHandle inside each call to the invokeDeviceCublasSaxpy was causing a call to cublasDeviceSynchronize, which introduces a performance penalty. For that reason, NVidia recommends minimizing cublasCreate and cublasDestroy occurrences in code [70].

Because of the above, we decided to generate a new solution that would not embed a setup and closure code for the CuBLAS handle with each library function call. Instead, we create a Global array of CublasHandle type that resides on the GPU. then, we choose to replace – rather than eliminate – the CublasHandle argument with an integer one. The integer argument will be used to identify a consistent CublasHandle to be used through

the calls for the same integer. This would move back the responsibility of setting up and closing CublasHandle into the client code. We just needed also to create an intermediate layer for CuBLAS calls designed for setting up and closing CublasHandle. Those intermediate calls need to work on integer identifiers rather than CublasHandle ones. The original CuBLAS create and destroy function prototypes are shown below along with their intermediate counter ones:

Table 3: CuBLAS header prototypes for handle management versus intermediate ones from AwCuBLAS

`cublasStatus_t cublasCreate(cublasHandle_t *handle)`	`void cublasCreate(int *handle)`
`cublasStatus_t cublasDestroy(cublasHandle_t handle)`	`void cublasDestroy(int handle)`

Using the above intermediate layer create and destroy functions would get back the usage pattern of the layer into a very similar scenario to the original CuBLAS one as now the user again can create and destroy handles at user specified times. Concurrency here would be much better. The experiment was implemented on a 400-element array. The measurement tool (shown in green color) shows the timing it took successive 15 iterations to complete, which is 474.337 milliseconds on the machine hosting the experiments. The first line of segments shown in red color shows the overhead of the driver API. The second line of segments, which is shown in gold color, shows memory copy operations happening automatically between calls.

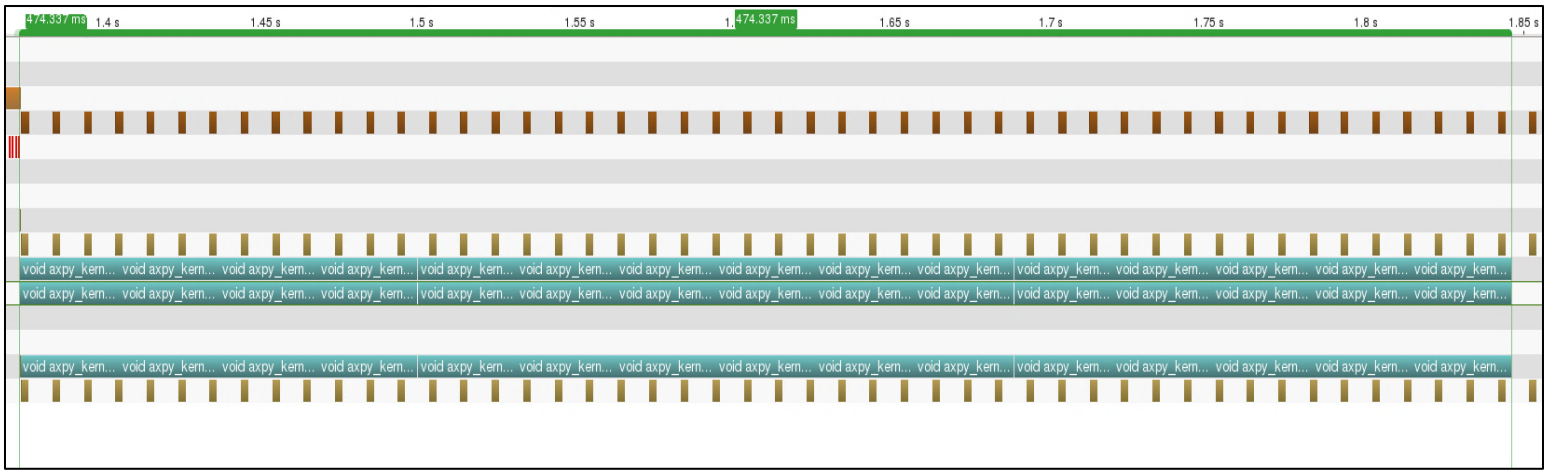

Figure 5: consecutive iterations of saxpy done using CuBLAS' host interface from within OpenACC' host region

Figure 6: consecutive iterations of saxpy done using AwCuBLAS from within an OpenACC compute region

Since CuBLAS and other libraries, in general, have a consistent usage pattern when it comes to handles, we can automate the process of generating an intermediate layer for the whole of CuBLAS' GPU interface. Hence, we made a layer library interface generator that accepts some rules in addition to the header of a CuBLAS library to generate an intermediate library that can be tweaked for any suitable purpose. We call this automatic generator ***libimorph***.

For our purpose of CuBLAS, ***libimorph*** would wrap the CuBLAS interface into another GPU-based interface that changes CuBLAS handle parameters into integer ones. Implementations of Generated function would internally call CuBLAS routines with their needed Handle parameters. This automation saves coding time, centralizes modifications and allows easy regeneration of the layer when newly introduced functions arise with new Library versions. One more benefit of this layer is the ability to recompile it for new CUDA functions while still linking it to the same compiler version, avoiding the need for compiler upgrades while still getting any possible performance enhancements with the new versions of the library.

After generating the new layer, we tested it with two applications that we describe here. The conjugate gradient (CG) method [72] is a method for solving linear systems of the form Ax = B. We use it here as a test case as it can be expressed entirely as steps of BLAS operations. The CG makes use of matrix-vector products, vector updates, and inner products. A serial version of the algorithm is listed in the literature[73, Sec. 3.1] in its preconditioned form. The non-preconditioned serial form of CG is listed in Listing 8 below. We also tested how the application of our developed suggestion solution is applied to the problem of matrix exponentiation. In our setup, to calculate A^e, the algorithm needs to multiply matrix A by itself for *(e)* times. Our goal here is not deploying an optimal algorithm of matrix exponentiation, but applying the same algorithm comparatively for both purely OpenACC and AwCuBLAS with OpenACC. For that purpose, we used the basic approach for matrix exponentiation.

Listing 8: The serial algorithm of the Conjugate gradient adapted from [74] after removing the preconditioning

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  $\rho_{i-1} = r^{(i-1)^T} r^{(i-1)}$
  if $i = 1$
    $p^{(1)} = r^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = r^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end

To assess the benefits regarding performance, we tested our setup on mixed OpenACC and CuBLAS. We expect a performance improvement due to the following effects

1. Eliminating overhead of repetitive kernel launches in a similar way to what the test case in [18] shows.

2. Performance improvement due to many specific CuBLAS functions being faster than auto-parallelized OpenACC kernels for the implementations available currently.

3.4 Performance and results

We performed the tests of the benchmarking operations on systems containing NVidia's K20 Kepler architecture GPUs. In the first phase of tests, described in section 3.5, we used systems that KAUST provided access to, these systems contained k20c cards. In the

second phase of experiments described in section 3.6, we used a system provided by the ICS department in KFUPM; containing k20x GPU card. The first phase of experiments was performed using double precision floating numbers. After the benchmarking, the second phase used floating point precision numbers to be able to saturate the machine with largest possible matrices. In both phases, all measurements were conducted over the timing of the kernel, disregarding pre and post data transfer times. Data transfer times depend on system hardware configuration as well as on idioms used during the transfer.

3.5 Benchmarking BLAS auto-parallelization

For the first three experiments, we used NVidia profiler to test kernel timings. Since all timings were measured using NVidia profiler, accounting for profiling overhead is not important for this comparison. Two OpenACC implementations were tested against the cuBLAS library, namely the open source accULL and PGI. Resolution of dimension variation was higher than what the x-axis labels illustrate on figures.

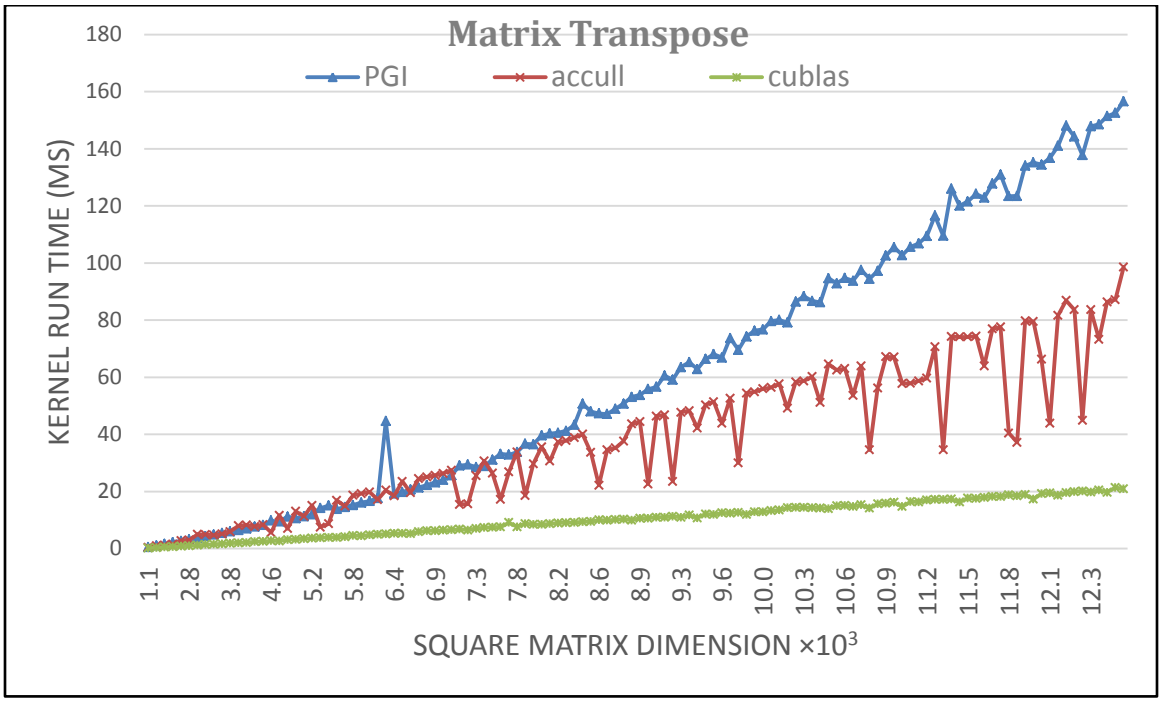

Figure 7: Performance of CuBLAS versus PGI and accULL against matrix size in Matrix transpose application.

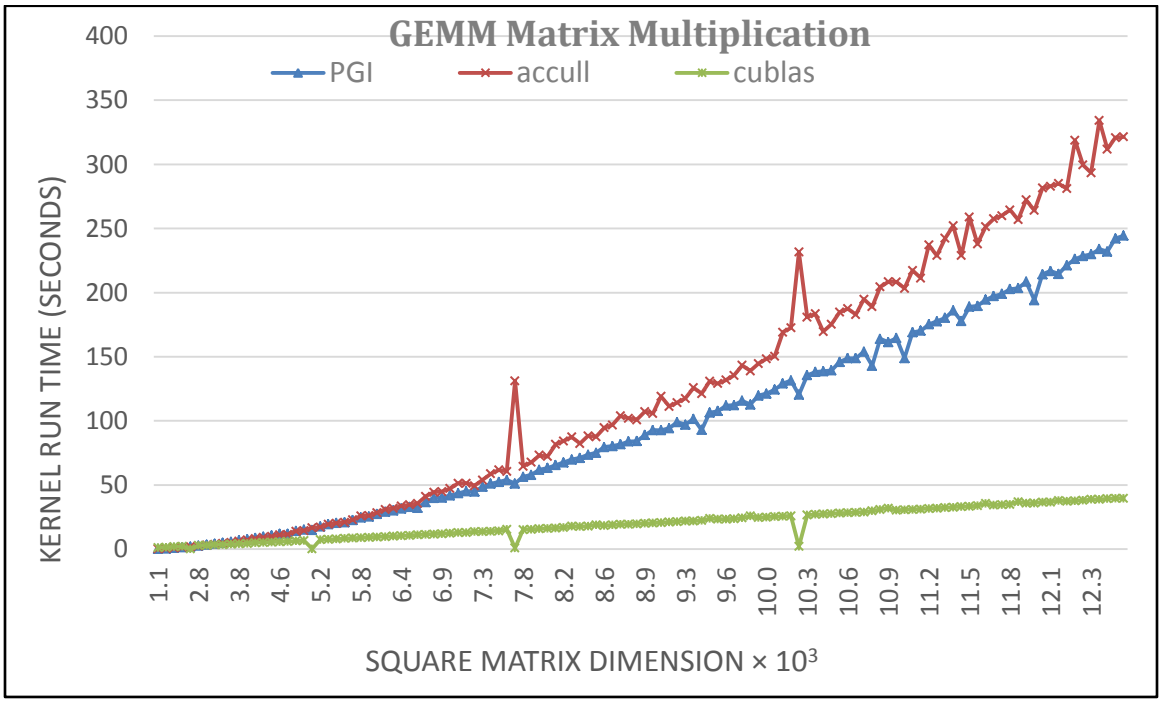

Figure 8: Performance of CuBLAS versus PGI and accULL against matrix size in GEMM Matrix multiplication application

Figure 7 above shows execution times of two OpenACC implementations (PGI and accULL) of matrix transpose versus CuBLAS library`s execution time against different matrix size. A clear performance advantage for is shown by CuBLAS library, followed by accULL OpenACC implementation. PGI's implementation shows the slowest implementation of the transpose application. The figure also shows the instability of accULL's performance with variations of matrix dimension (scaling). Figure 8 above shows the execution of PGI and accULL versus CuBLAS library for the GEMM (General Matrix Multiplication) against several matrix sizes. Again a very clear performance advantage is shown by CuBLAS library. Moreover, PGI OpenACC and CuBLAS shows jumps towards better performance for certain dimensions on the contrary of accULL; which shows worst performance for those same points. Figure 9 below shows that for matrix addition, CuBLAS and PGI are almost on par with each other with a very slight consistent advantage for PGI. To wrap up, while accULL is clearly outperformed in matrix addition; it performed better than PGI in matrix transpose. Moreover, there was a clear fluctuation in performance due to size change in accULL, less noticed in PGI and almost non-existent in CuBLAS. PGI OpenACC and CuBLAS shows very rapid performance increases for certain dimension ranges on the contrary of accULL which shows worst performance for those same ranges of dimensions. It was unclear why such performance rapid increases happened for certain dimension ranges except for the observation that CuBLAS implementation executes less number of kernel launches (specifically only one) while it calls two extra kernels for other dimensions. In remaining experiments, since -for the previously mentioned reasons- we will only use PGI OpenACC and CuBLAS; we decided to use the dimensions that show performance peaks. These dimensions set clear peaks for both implementations in performance and provide better isolation for measuring the performance of the algorithm's implementation as they apparently need less setup code as verified by profiling the CuBLAS, where only single kernel launches were noticed for these dimensions.

Looking at figures above, one can see the performance variability of compilers among different problems and –for some compilers – among different matrix sizes. This performance variability in auto-parallelized code gives developers a bigger motivation to

offload their work to such libraries as the parallelization process for them are usually tested and tweaked for performance. However, some applications involve invoking BLAS Operation among other operations that are not found in Libraries. Moreover, there are iterative algorithms that could benefit from the complex algorithms that contain steps could benefit from removing the overhead of invoking libraries using their host interface repeatedly; a process which OpenACC could make easier.

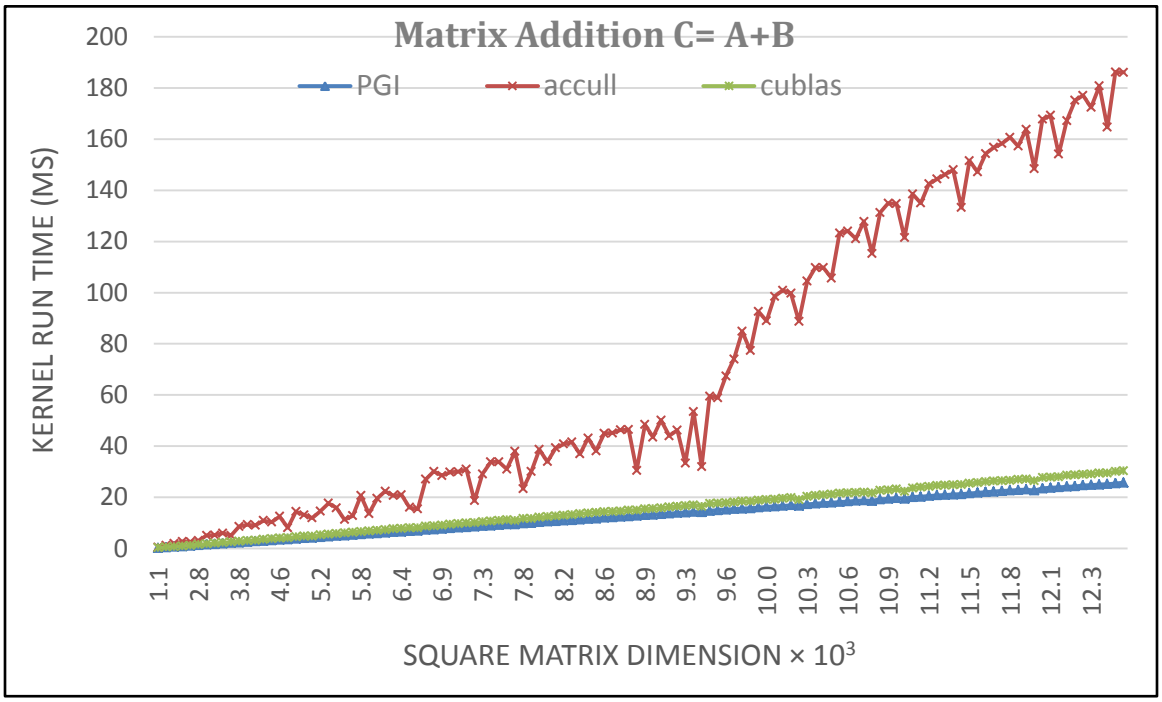

Figure 9: Performance of CuBLAS versus PGI and accULL against matrix size in Matrix addition application.

The concept of using OpenACC with libraries is referred to as OpenACC interoperability. However, we only saw references to the concept of interoperability in the context of mixing OpenACC code with libraries (mainly CuBLAS) using their Host interface. Restricting OpenACC' interoperability to be used in that context would introduce performance penalties or miss performance improvement opportunities, due to the following reasons:

- Implementing GPU algorithms on device code will be limited to constructing them in a way that calls to GPU libraries happen from within CPU code; holding

the obvious overhead of having to exit the GPU region to call a library that will set itself to execute on GPU.

- Synchronization points that are not necessarily needed will be imposed at the points that library functions need to be called from; since departing GPU parallel regions will imply a wall among thread blocks
- Parallel task decomposition among parallel CTAs will be limited mostly to regular decomposition due to the previous point, which limits developers' options to enhance performance.

In [67], authors compared the use of device interface of CuBLAS against the host interface for repetitive matrix multiplications of small size. The authors demonstrated that using device interface, 21 matrix multiplications can be achieved with the CuBLAS device interface using CUDA versus only four matrix-matrix multiplications using CuBLAS host interface in the same time duration (a 5x speedup). That was due to the PCIe bus and operating system overhead. In order to achieve such performance gain for CuBLAS from within OpenACC in iterative algorithms, one needs to be able to call CuBLAS device interface from OpenACC implementations.

3.6 A method for calling Libraries with pointers to opaque structures as arguments from within OpenACC compute regions

In this section, we will use CuBLAS library as an example of a library that expects an opaque struct pointer as an argument that resides on the device. All components of CuBLAS library have an interface that expects a pointer to an opaque data structure as an argument.

As a first step, we propose a solution that would allow calling CuBLAS and any library with a device interface that have such a condition. The method can be described using the following steps:

1. Define a CUDA device function in a separate file. This can be done via prefixing the declaration with the __device__ keyword. It should receive all the parameters needed by the Library subroutine interface; except for the Opaque ones (the CuBLAS handle in the case of CuBLAS).
2. The defined function shall encapsulate a call to the intended Library routine while managing any code that needs to create and assign the opaque operands needed by the library call. In the case of CuBLAS, this means creating a CuBLAS handle and closing that handle after the call.
3. The function mentioned in the previous step shall be compiled via NVidia's compiler in two phases to provide an object code that the OpenACC compiler can use to call that function.

- Notice that in some cases; only one phase is needed for OpenACC compilers that have a linker compatible with NVidia's.

  In OpenACC, a special directive shall be used to declare the function as being in a form already transformed to the device. This directive is the routine directive, which has the usage form shown below

```
#pragma acc routine seq
(function prototype)
```

4. After that, calls to the defined function can be made strictly from within OpenACC parallel regions (in OpenACC 2.5 standard, these calls can be made from kernels regions too)
5. CUBLAS calls returns a status code informing the caller about the success or failure of the call (in the case of the code can be used further to determine the possible cause of failure). However, this status cannot be used currently in OpenACC, due to a limitation in the ability of the tested PGI compiler to recognize the status code type inside OpenACC programs. However, since the data type is an enum, the wrapping function can return a result casted to an int.

The proposed method allows calls to be made from within OpenACC compute regions into libraries with the same condition as the one can be seen in CuBLAS' interface. Obviously, this process is tedious. First, it needs the programmer to create a wrapper

function for each needed CuBLAS routine in advance. Second, it needs the programmer to switch between different compilers for accomplishing this task. Those stages can be reduced to three stages for compilers that are compatible with CuBLAS; since the compilation and linking phases might be done with a single command. The process needed for calling a library`s device routines from within OpenACC compute regions can be as follows:

1. Define wrapper function
2. compile with NVidia compiler
3. link object code with device library using NVidia linker (not needed for PGI)
4. Compile OpenACC code
5. link with wrapper using OpenACC compiler

We demonstrate below a proof of concept example for that process. For simplicity, we use one CuBLAS call in the OpenACC region. Obviously, our intended audience would need to use this call among a complete algorithm with multiple steps being parallelized using OpenACC, and possibly multiple calls to different CuBLAS routines. Our example considers the need to perform AXPY operation (scale a vector and add it to another vector) in their algorithm. Notice that users need a wrapper function for each different CuBLAS routine they intend to call.

1. The first step is to define the wrapper function in a conventional CUDA file "cublasSaxpy_wrapper.cu". This function would contain the actual code for invoking SAXPY CUBLAS call, shown in Listing 9 below.
2. The second step involves compilation using NVidia's *nvcc* compiler tool, producing an object file.
3. The third step is to use NVidia linker to link the device object code with CuBLAS device relatable object code.
4. The fourth step involves the example coding of OpenACC code in a separate file "callSaxpy_example.c"; remember that this code shall be prefixed with the necessary OpenACC routine directive to stop the compiler from trying to parallelize the invoked

function. Listing 9 shows example code. For simplicity, the code and related interface for identifying possible error code are omitted as it does not affect the operation of the example.

5. The fifth step involves compiling the OpenACC code, producing object code
6. Then this object code needs to be linked using PGI compiler with the object code of the prior steps.

Listing 9: Wrapper functions and corresponding client code for invoking the saxpy CuBLAS device routine.

```
01 // file: cublasSaxpy_wrapper.cu
02 __device__ void invokeDeviceCublasSaxpy(int n,
03                                         const float *const d_alpha,
04                                         const float *const d_X,
05                                         float *const d_Y
06                                         )
07 {
08     cublasHandle_t cnpHandle;
09     cublasStatus_t status = cublasCreate(&cnpHandle);
10
11     if (status != CUBLAS_STATUS_SUCCESS)
12     {
13         return;
14     }
15     /* Perform operation using CuBLAS */
16     status = cublasSaxpy(cnpHandle, n, d_alpha, d_X, 1, d_Y, 1);
17     cublasDestroy(cnpHandle);
18 }
```

```
01 //file: callSaxpy_example.c
02 //function prototype
03 void invokeDeviceCublasSaxpy(int n,
04                              const float *const d_alpha,
05                              const float *const d_X,
06                              float *const d_Y);
07 //function prototypes are needed for every different routine the user
   intends to make a wrapper for.
08 #pragma acc parallel num_gangs(1) vector_length(1)
   pcopyin(x[:n],y[:n],ptr_alpha[:1])
09       {      … //prior code if needed
10              invokeDeviceCublasSaxpy( n, ptr_alpha, x, y);
11              … //posterior code if needed }
```

Our view of this process is that it is still inconvenient for developers. We also want a standardized method that would work regardless of the OpenACC compiler being used. In the next section, we do exactly that; we take the work a step further and automate the process of generating intermediate wrappers for such device library interfaces.

3.7 Automating the process: auto-generated Device Library wrappers

The process described earlier to use device library subroutines that have a pointer to an opaque structure as an argument is obviously not straightforward; as it needs a wrapper function for each needed device function call.

However, we noticed that pointers to opaque arguments have a generic and systematic method of use in Library interfaces, such as the case with CuBLAS handles;which opens an opportunity for a systematic approach that eases the use of such libraries in general.

We decided to make this approach easier by making an API interface to provide easier access for the user to use advanced device library routines from within OpenACC compute regions.

Our API would make it convenient to call device library routines by the user immediately from OpenACC parallel regions, just as easy as just invoking a function call.

Our presented work is a library that writes all the wrapper functions necessary to call CUBLAS Routines. The new process for using CuBLAS Device library routines involves the following steps:

1. The user calls any necessary CuBLAS function (with a predetermined small name modifier) inside parallel regions of OpenACC as necessary. The user does not include the CuBLAS handle argument and does not make any consideration for the needed management code for it.
2. The user compiles the code with the necessary linker flags with a single step.

To continue with our SAXPY example, we list the involved steps in creating the necessary calls. Listing 10 and Figure 10 show the new code and the diagram for the new process for invoking the CuBLAS library subroutines. The code shows an example of the code needed to invoke CuBLAS library routines from within parallel regions. The code is much smaller in size than the previous process that does not involve our API. Notice that

this example is simplistic, where only a single gang is needed to launch the code for demonstration. Looking at the diagram, we can see the simplicity of the new process for invoking CuBLAS device routines from within OpenACC compute regions. The code launches a single OpenACC gang and vector because the example problem contains only one instance of saxpy that needs to be solved. The called saxpy routine will dynamically launch a new kernel with a sufficient number of blocks and threads.

Listing 10: Example saxpy client code from within OpenACC compute region using our API.

```
01  //file: callSaxpy_example_api.c
02  #include “cublas_introp.h” // this would be our wrapper libraries header
    file
03  #pragma acc parallel num_gangs(1) vector_length(1)
    pcopyin(x[:n],y[:n],ptr_alpha[:1])
04        {       … //prior code if needed
05                invokeDeviceCublasSaxpy( n, ptr_alpha, x, y);
06                … //posterior code if needed
07        }
```

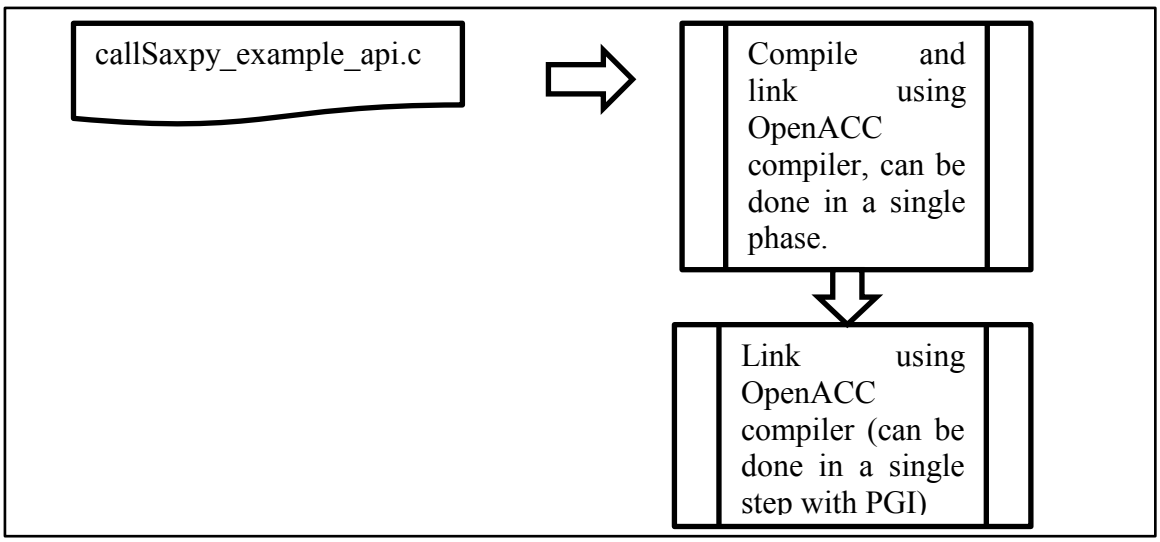

Figure 10: The new process for involving CuBLAS from OpenACC regions when using our AwCuBLAS API.

3.8 Application tests

To assess the benefits regarding performance, we test our setup with mixing OpenACC and CuBLAS; expecting the performance improvement to due to the following effects

1. Eliminating overhead of repetitive kernel launches
2. Performance improvement due CuBLAS functions being faster than auto-parallelized PGI OpenACC kernels.

In the next few sections, we will list results of sample tests that exploit each of the previous aspects.

The conjugate gradient (CG) method [75] is a method for solving linear systems of the form Ax = B. we use it here as a test case as it can be expressed entirely as steps of BLAS operations. The CG has matrix-vector products, vector updates, and inner products. A serial version of the algorithmcan be found [73].

In our tests, we fixed the matrix generation procedure to

We also tested the application of our solution to matrix exponentiation. In our setup, to calculate A^e, the algorithm needs to multiply matrix A by itself for *e* times. Our goal here is not finding an optimal algorithm for matrix exponentiation, but to apply the same algorithm for purely OpenACC and for OpenACC+AWCuBLAS so that we can compare the improvement.

The performance results show that for Matrix exponentiation, our method drastically improves the application of OpenACC to solve the problem. Invoking AwCuBLAS from within GPU improved the performance of the solution with up to 34 times the Pure OpenACC solution. Thisspeedup comes mainly from the fact that matrix multiplication is much faster to accomplish in CuBLAS library.

For the conjugate gradient solver, however, the performance improvement was modest gains the pure OpenACC solution due to the fact PGI's OpenACC parallelization is better for the matrix-vector multiplication and also for vector inner product and scaling. The reason for the slight improvement in our approach here comes from the fact that our approach allows calling all CuBLAS operations with a single kernel launch from GPU, eliminating the overhead of repetitive kernel launches.

Listing 11: OpenACC+AW-CuBLAS code listing for the conjugate gradient linear solver

```
01 #pragma acc data … // rest of data clause
02 {
03 #pragma acc parallel num_gangs(1)  copyout(handle)
04   {
05 #pragma acc loop seq
06       for (i = 1; (i <= max_iter) && !converge_flag; i++) {
07           invoke_cublasScopy(n, (const float *) r, 1, z, 1);
08           invoke_cublasSdot(n, (const float *)r, 1, z, 1, &rho);
09           if (i == 1) {
10               invoke_cublasScopy(n, (const float *) z, 1, p, 1);
11           } else {
12               beta = (rho / rho_1);
13               invoke_cublasSscal(n, &beta, p, 1);
14               invoke_cublasSaxpy(n, &alpha_one, z, 1, p, 1);
15           }
16           invoke_cublasSgemv(CUBLAS_OP_N, n, n, &alpha_one, A, n, p, 1,
   &beta_zero, q, 1);
17           invoke_cublasSdot(n, p, 1, q, 1, &alpha);
18           alpha = rho / alpha;
19           alpha2 = -alpha;
20           invoke_cublasSaxpy(n, &alpha, p, 1, x, 1);
21           invoke_cublasSaxpy(n, &alpha2, q, 1, r, 1);
22           sum = 0;
23           invoke_cublasSnrm2(n, r, 1, &resid);
24           rho_1 = rho;
25           if (stop_on_converge && (resid <= tol)) {
26               tol = resid;
27               converge_flag = 1;
28           }
29       }
30   }
31 }
```

Listing 12: OpenACC code for conjugate gradient solver

```
01  #pragma acc data … //data clause omitted as it is big
02  {
03         for (i = 1; (i <= max_iter) && !converge_flag; i++) {
04  #pragma acc  kernels
05                 {
06  #pragma        acc loop independent
07                         for (j = 0; j < n; j++)
08                         z[j] = r[j];
09                         rho = 0;
10  #pragma acc loop
11                         for (j = 0; j < n; j++) {
12                                 rho += r[j] * z[j];
13                         }
14                         if (i == 1) {
15  #pragma acc loop independent
16                                 for (j = 0; j < n; j++) {
17                                         p[j] = z[j];
18                                 }
19                         } else {
20                                 beta = rho / rho_1;
21  #pragma acc loop independent
22                                 for (j = 0; j < n; j++)
23                                 p[j] = z[j] + beta/*(rho / rho_1)*/* p[j];
24                         }
25  #pragma acc loop independent
26                         for (j = 0; j < n; j++) {
27                                 sum = 0;
28  #pragma acc loop
29                                 for (int j2 = 0; j2 < n; j2++) {
30                                         sum += A[j * n + j2] * p[j2];
31                                 }
32                                 q[j] = sum;
33                         }
34                         alpha = 0;
35  #pragma acc loop reduction(+:alpha)
36                         for (j = 0; j < n; j++) {
37                                 alpha += p[j] * q[j];
38                         }
39                         alpha = rho / alpha;
40  #pragma acc loop independent
41                         for (j = 0; j < n; j++) {
42                                 x[j] += alpha * p[j];
43                         r[j] += -alpha * q[j];
44                         }
45                         sum = 0;
46  #pragma acc loop
47                         for (j = 0; j < n; j++) {
48                                 sum += r[j] * r[j];
49                         }
50                         resid = sqrt((double) sum) / normb;
51                 }
52  #pragma acc update self(resid)
53                 if ((resid <= tol) && stop_on_converge) {
54                         tol = resid;
55                         converge_flag = 1;
56                 }
57                 rho_1 = rho;
58         }
59  }
```

We noticed when applying the transformation into AwCuBLAS a code size decrease regarding lines of code down to 52.5% for the conjugate gradient application kernel and down to 53.8% for the matrix exponentiation application kernel.

We can see that the act of using libraries relief programmers from researching the parameters that would affect the performance such as optimizing the size of software managed cache or optimal block size to increase the performance of their OpenACC applications. Of course, this applies to only the parts of the software that can reuse components from a GPU library.

3.8.1 AWCUBLAS performance results

We performed several tests explained in section 3.3 over AwCuBLAS from within OpenACC compute regions. Figure 11 shows the performance in execution time of OpenACC versus OpenACC+AWCuBLAS for the CG solver application against various matrix sizes. We notice that using device AwCuBLAS routines from within OpenACC yields better performance than using OpenACC auto-parallelization. This result is interesting because our profiling result for specific kernels shows that PGI's OpenACC implementation for dot product and matrix-vector multiplication is faster than the CuBLAS counterparts. Those operations are the key operations in determining the timing of CG solver`s implementation. Still, we see an improved performance when depending on AwCuBLAS from OpenACC. This improvement is due to the ability to ensure the serialization of all the iterations of CG-solver without the need for repetitive kernel launches from CPU.

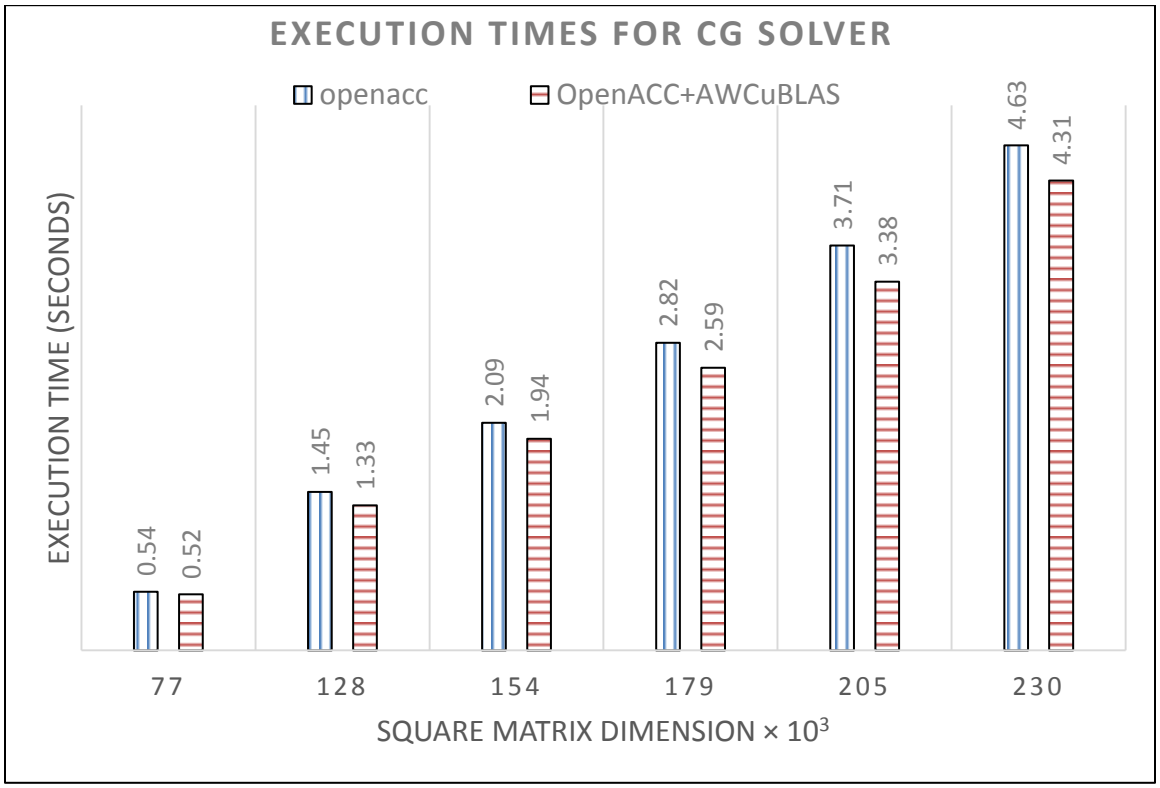

Figure 11: Execution times of OpenACC versus AWCuBLAS for CG solver application against matrix size.

Figure 12 presents execution time for the matrix exponentiation application. It shows execution times over variation in exponent for the Matrix exponentiation program. A clear advantage of OpenACC+AWCuBLAS over pure OpenACC is evident from the chart. The case that uses our AwCuBLAS shows better results when used from within OpenACC compute region versus depending on the automatic parallelization of PGI compiler for the serial algorithm. As expected, the more multiplications (exponent), the wider the performance gap. The major contributing factor in this gap is the superiority CuBLAS has when it comes to matrix multiplication. While another factor is the elimination of the need for repetitive kernel launches as a single block of threads can dynamically call CuBLAS device interface while ensuring the serialization of the iterations of the algorithm.

We measured the speedup of both approaches in our experiments over the pure Host-based CuBLAS interface approach. This comparison was possible because the applications we are testing can be expressed fully in BLAS operations. This o answers a

question that may arise for applications that can be fully expressed via BLAS operations, does OpenACC carries any benefit for those applications. Before conducting this test, one may think that developing such applications using pure host-based CuBLAS is sufficient for getting the best performance provided by such library. However, looking at Figure 13 below, we see an advantage for our approach most clear for low exponents of matrices (lower number of iterations). This advantage is biggest for a low number of iterations because the overall percentage of kernel exit-entry overhead is biggest when the number of multiplications is still smaller.

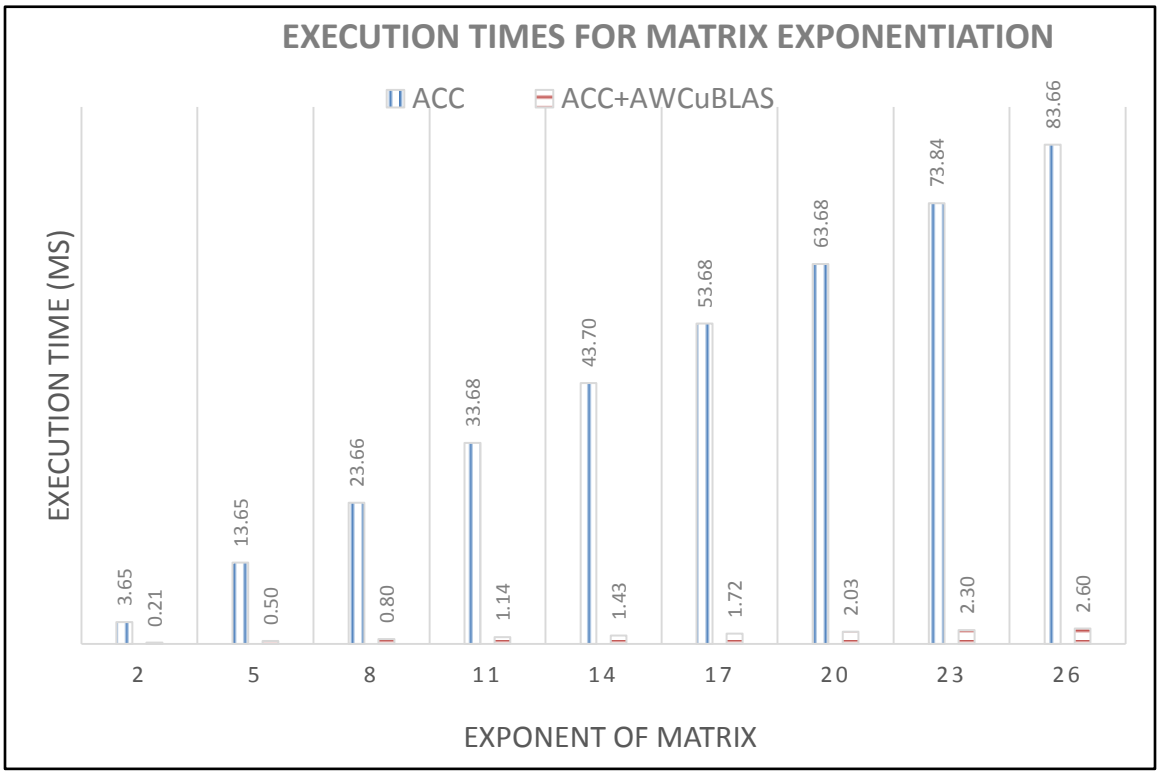

Figure 12: Execution times of OpenACC versus OpenACC+AwCuBLAS against the exponent for the Matrix exponentiation kernel.

We also tested for speedup of the approaches over variation in matrix size for the matrix exponentiation application. Looking at Figure 14, a clear advantage of OpenACC+AWCuBLAS over pure OpenACC is evident from the chart. Our approach has shown tremendously better performance regarding execution time attributed to

CuBLAS superior matrix multiplication along with the elimination of repetitive kernel launches in our approach.

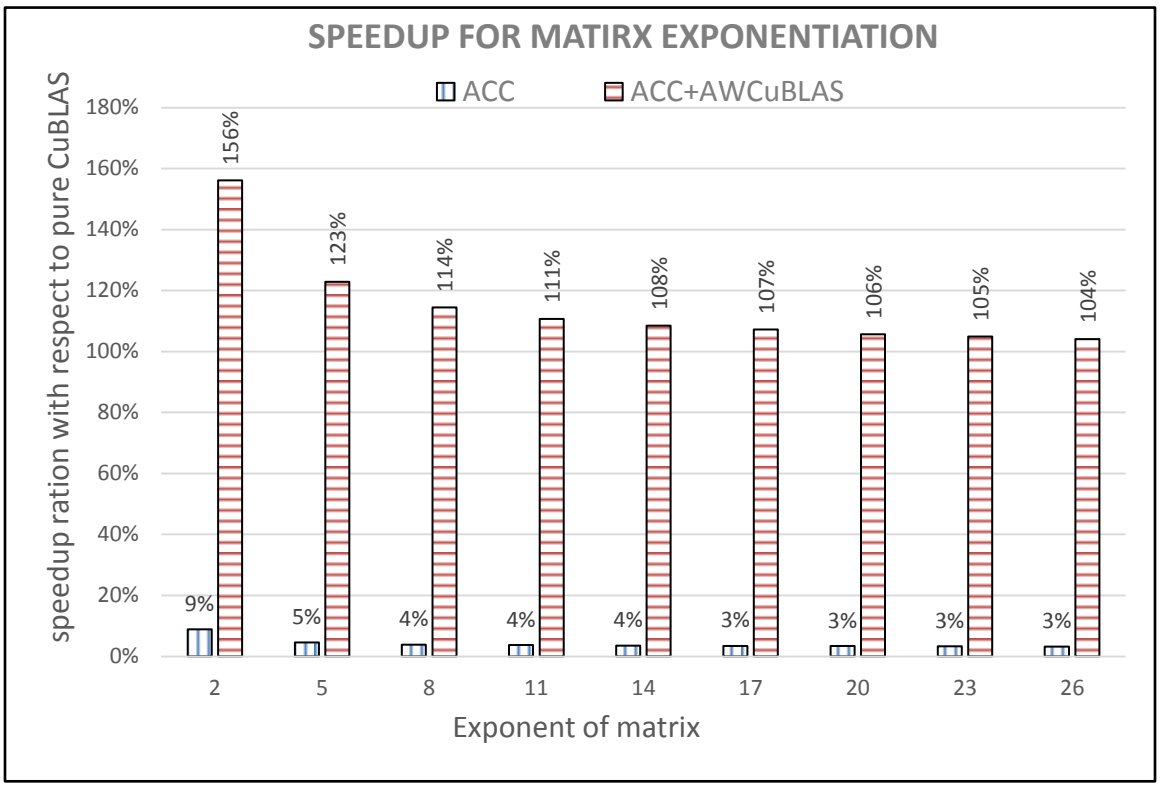

Figure 13: Speedup for Matrix exponentiation application against exponent of OpenACC versus our AwCuBLAS both measured over host-based CuBLAS approach.

In a similar way to the previous tests, a test for speedup of both approaches with respect to pure CuBLAS was performed, but this time while varying matrix sizes and fixing the exponent for the matrix exponentiation application. As seen in Figure 15 This test again shows that OpenACC+AwCuBLAS either perform similar to or better than pure CuBLAS in that application, while pure OpenACC has about one tenth the speedup over pure host based CuBLAS for this application.

These performance results show that our method drastically improves the application of OpenACC to solve the problem. Invoking AwCuBLAS from within GPU improved the performance of the solution with up to 34 times the Pure OpenACC solution. This

improvement comes mainly from the fact that matrix multiplication is much faster to accomplish in CuBLAS library; which explains the increase in speedup with matrix size.

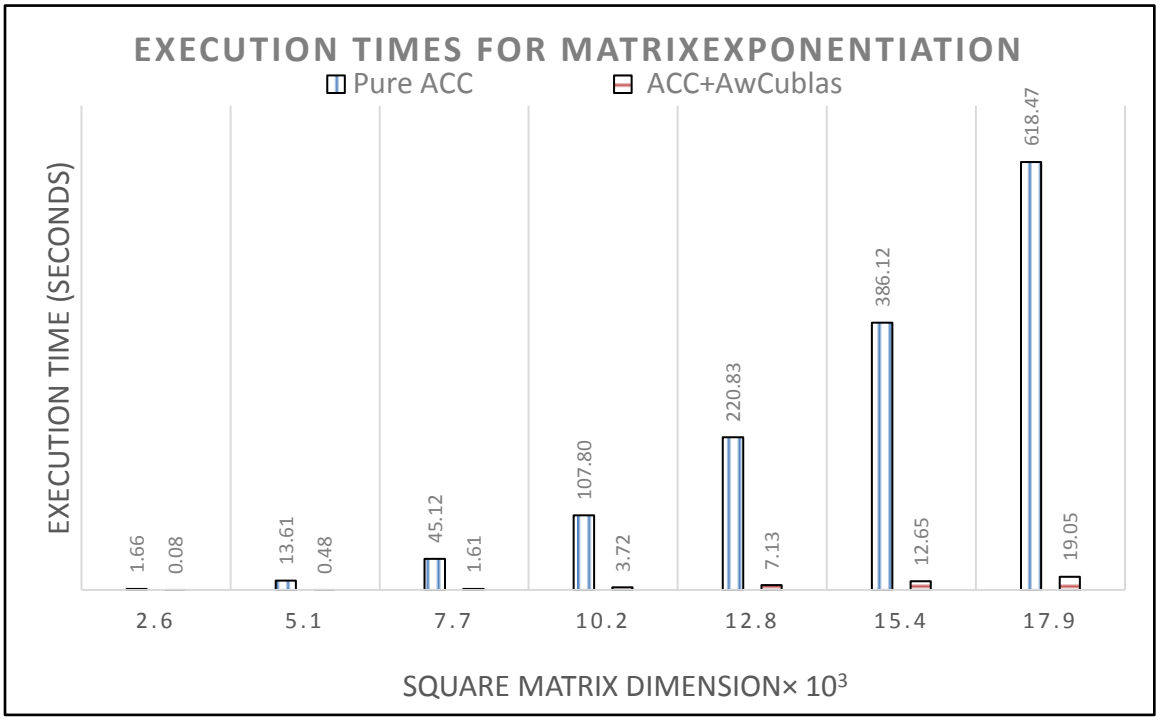

Figure 14: Execution time for the Matrix exponentiation application against matrix size of OpenACC vs. AwCuBLAS approach.

For the conjugate gradient solver, however, the performance improvement was modest against the pure OpenACC solution due to the fact PGI's OpenACC parallelization is better for the matrix-vector multiplication and for vector inner product and scaling. The reason for the performance enhancement in our approach here comes from the fact that our approach allows calling all CuBLAS operations with a single kernel launch from GPU, eliminating the overhead of repetitive kernel launches. In light of the fact that the system we are testing against in CG converges within about 4-6 iterations; the enhancement would be modest in this regard.

We noticed when applying the transformation into AwCuBLAS a code size decrease in terms of lines of code down to 52.5% for the conjugate gradient application kernel and

down to 53.8% for the matrix exponentiation application kernel from the code that parallelizes serial code.

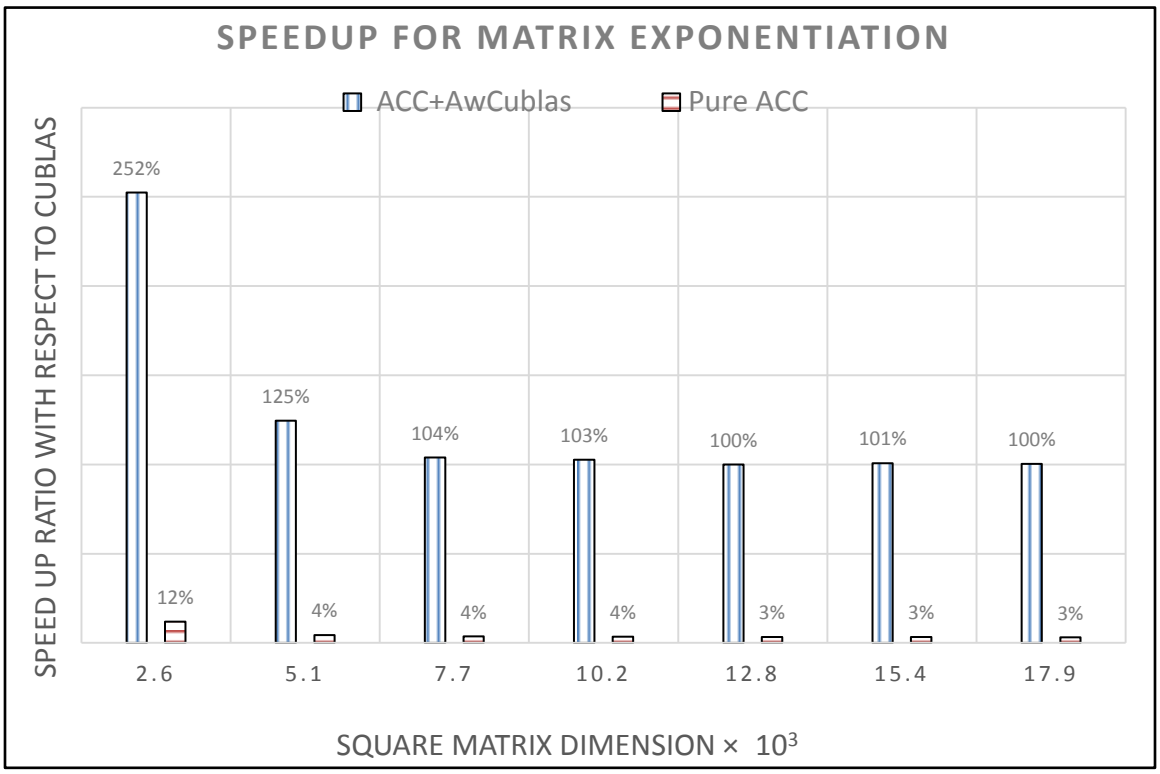

Figure 15: Speedup for Matrix exponentiation application against Matrix size for OpenACC versus our AWCuBLAS, both measured over host-based CuBLAS approach.

3.8.2 Libimorph: an automatic library interface changer to change call characteristics

The use of software libraries has recently become an essential part of software design. When the use of libraries is applyied appropriately, it would reduce development cost; this is because it reduces expertise needed by developers, reduces domain knowledge related to the problems solved by such libraries and reduces the overhead in software testing.

This concept takes a special importance in the domain of numerical computations and simulation; as computations in these domains share a lot of well-known algorithms and

methods. Enhancing the performance of numerical computations needs a deep understanding of algorithms and computer architecture involved. Using well-performing libraries in this domain would usually yield to an improvement.

Automatic parallelization techniques are also getting a place in targeting GPU accelerators for computations. However, as shown in Chapter 3.8.2, mixing library usage with automatically parallelized code is not always an easy process. In this chapter, we introduce libimorph. Libimorph would be a system for changing (morphing) the interface of input library headers in a way that adapts it to the most suitable use of different technologies. We needed the functionality of libimorph while we were developing a solution to the problem of calling GPU device routine libraries such as CuBLAS[18][NVidia 2008][13][OpenACC Working Group 2011] compute regions.

One commonly used aspect of API design is information hiding. Information hiding, in the context of API design, is a concept that refers to hiding the implementation details of a library or an object from its users (clients). Information hiding is considered an important quality of API design, as it has the advantage of vastly increasing compilation speeds for client code, in addition to the ability to upgrade or change the implementation without the need to recompile client code[76]. Physical hiding refers to the aspect of this concept where the source code that is kept private is not made available to the client. Logical hiding is concerned with using the feature from the programming language that limits clients' code access to the implementation details logical hiding is called encapsulation.

In C language, API information usually is implemented in C and C++ by an idiom called an Opaque pointer or a pimpl (pointer to implementation) [76], [77]. The pimpl is more of a workaround that an idiom since it is used to hide the implementation via a workaround because C and C++ languages do not provide a standard mechanism for accomplishing information hiding.

Inside CuBLAS library headers, a CuBLAS handle is defined as shown in Listing 13 below. We can see that this pimpl uses a forward declaration for a cublasContext, and defines the *cublasHandle_t* as a C language *typdef* of a pointer to a cublasContext.

Listing 13: Definition of cublasHandle the same the way it is found in cublas header file.

```
struct cublasContext;
typedef struct cublasContext *cublasHandle_t;
```

The interface for creating a cublasHandle (which is a pointer to a cublasContext) is shown in Listing 14 below.

Listing 14: The function prototype of cublasCreate the way it is defined in cublas Library header

```
__attribute__((host)) __attribute__((device)) cublasStatus_t cublasCreate_v2
(cublasHandle_t *handle);
```

One problem with opaque pointers is that the implementation would only know the size of the object when, for example, the object is created through the API. The lack of knowledge of the size at compile time is a problem with OpenACC compilers because they need to know in advance the size of the data before entering a data region. Moreover, even when using the cublaCreate_t function from within a compute region in OpenACC, for example, the compiler still faces a problem as there still must be a known size for the handle to be created inside the data region before being assigned to an object

3.9 Future work and conclusion

In this chapter, we proposed software that applies a systematic approach for enabling the use of libraries with GPU device interface from the compute regions of OpenACC. that approach is to change the interface of the device interface of a GPU library into a one that eliminates the need for opaque arguments. This can be expanded and adapted for other suitable adaptation needs of the interface. The advantage of our technique is its generality, where compilers need not support each library independently. Moreover, this

wrapping technique with handle management could be done once and used by multiple compilers.

Another advantage of our technique is that it provides ease of programming to OpenACC developers; through the enablement of a wider range of library device routines from within OpenACC compute regions. Also, while more OpenACC compilers get introduced, our approach would provide developers with more freedom to choose compilers regardless of their support for specific device library interfaces.

Finally, our approach provides OpenACC developers with opportunities to increase performance without switching programming paradigms. For example, when applying our customized approach to CuBLAS Library to generate AwCuBLAS with handle management, our method demonstrated a speedup that reached 252% over using host-based CuBLAS calls. A speedup against using OpenACC exclusively in some applications reached about 32X. Moreover, even for algorithms in which OpenACC produces better performance than Numerical libraries, our method demonstrated a slight performance advantage due to our method's ability to accomplish all the iterations using a single kernel launch. Moreover, using device routines from within compute regions would allow for a much more flexibility with data decomposition of parallel programs because it opens opportunities for using these libraries from blocks or threads, eliminating the need for ending compute regions to call GPU routines from CPU regions.

Regarding code size, our method decreased the needed code size down to 54% against using OpenACC exclusively. While looking for limitations, we could not find cases in which using well-made device routines from within OpenACC compute regions would be at a disadvantage regarding performance nor code size.

In the future, we expect more GPU libraries to come with device interfaces, and we would like to see the approach being applied to them. If an implementation of a device interface for a sparse computations library is found, we expect applying AWG there would be of significant impact.

CHAPTER 4
IN-HOUSE ORCHESTRATION OF ITERATIONS IN GPU PROGRAMS TO REDUCE OVERHEAD OF SYNCHRONIZATION

4.1 Introduction

4.1.1 Background

In the past decade, the field of high-performance computing has been rapidly changing. From one point, hardware accelerators have been pushing the boundaries of computational discoveries due to their characteristics that incorporate low-cost, energy efficiency, and performance[3]. Graphics processing devices, when designed and operated to target general purpose computing have demonstrated impressive advances in computations science. Some experts believe that those systems would be the building blocks for future high-performance computing platforms[4]. However, the programming of these devices is yet to mature enough to a state that allows developers easy access to the performance of these devices. That's why developing mature, efficient and easy-to-use programming models are vital to the success and longevity of these architectures.

Some languages and programming tools targeted GPUs to make GPGPU accessible to developers. Extensions to programming languages were introduced such as CUDA[8], OpenCL[9] and BSGP [21] made GPU programming accessible to developers.

Most commonly used hardware accelerators that are used for general purpose computing use multithreading to cover-up for operations of time-consuming latency, which are mainly memory operations. Thus, application writers need to expose a larger degree of parallelism for applications of this model to fulfill its intended performance goals.

Current Accelerator devices targeting high performance computing are constructed as IO devices, i.e. their physical memory and the host's physical memory are separate. That makes good memory management an important part of achieving high performance[22].

Usually, programming models for GPU accelerators present an abstraction that groups threads together sharing scratchpad memory and a processing element`s cache. These groups are called cooperative thread array (blocks in CUDA and gangs in OpenCL). Several groups of cooperative thread arrays (blocks) are usually assigned to each processing element in GPUs to tolerate latency operations via switching among these groups for each processing element.

In CUDA, a processing element is called an SM (while a cooperative thread array is called a block. The whole combined structure of blocks/threads is called a grid. In the rest of this work, we will use CUDA terminology as we are working using CUDA and related technologies.

Because of the architecture and design of GPU hardware accelerators, they are most suitable for data parallel applications [23]. This suitability is because GPU hardware accelerators lack a direct path between processing elements to share data between them.

Many parallel algorithms (data or otherwise task based ones) need global (barrier) synchronization to produce results correctly. One such case is iterative methods in numerical linear algebra where serialization needs to be enforced across iterations. Due to hardware constraints, programming models for GPU accelerators do not include a method for global synchronization in their API. Instead, a program needs to divide execution kernels around synchronization points. Global synchronization is accomplished in this method due to the ability to configure the system to wait for previous kernel launches in a certain context before launching new ones. This method is considered by some authors to have some significant overhead[78] on the technologies and architectures they tested on.

In CUDA, blocks assigned to the same SM are not able to communicate using the SM's shared memory. One possible solution to bypass this limitation is constructing global synchronization mechanism through global memory. *syncthreads* API call in CUDA functions as an intra-block synchronization mechanism, where threads within a single block are synchronized.

Some global-memory synchronization methods were proposed in the literature. Some of them are (lock-based) [8], [78], some of which are based on the system provided atomic operations over variables in global memory. Atomic operations, however, are associated with performance penalty due to memory access serialization. Some researchers proposed synchronization methods that avoid locks. Xiao's method[78], for example, uses an array of variables instead of a single variable the blocks would contend for.

In this work, we validate lock-free inter-block synchronization the way it was proposed by Xiao et al. [78]. We test programs done by Khan et al. [79], which implement Jacobi iterative solver using several synchronization methods. They utilized Xiao et al. [78] inter-block synchronization mechanism to test inter-iteration pipelining to Jacobi iterative solver. Although the authors referred to inter-iteration pipelining as relaxed synchronization, the term relaxed synchronization is usually used in literature to refer to the act of omitting synchronization where such an omission would not lead to catastrophic behavior of the system [80] in the context of the target application. However, because we will use the same programs used by the work`s authors, we will refer to them in the same manner that they use. They refer to their pipelined version of Jacobi as relaxed Jacobi (RJ), and we will do the same when referring to that algorithm, so the reader is advised not to confuse that algorithm with the ones discussed in the literature as a method of Jacobi solver with some synchronization omitted.

We have evaluated and analyzed the correctness and performance of inter-block synchronization lock-free technique as being used in literature. We contrast its performance and correctness with traditional, reliable methods using OpenACC automatic parallelization framework.

For evaluation of correctness, we use same programs used in another work [79] that was used to evaluate performance of inter-block synchronization when applied plainly and when applied to make inter-iteration pipelining (which is called relaxed synchronization by the authors of the work)

4.1.2 MANY-CORE ARCHITECTURES

GPUs are a form of throughput based high performance computing. It hides non-computational latency via fast context switching among threads. Each processor in a GPU accelerator device is a usually capable of running a massive number of threads. NVidia calls these processors as SMs or SMXs, depending on generation. GPUs usually have a parallel stream optimized memory shared among GPU processors, which NVidia calls global memory. A much faster, smaller and parallel memory also usually is present, which is shared between threads (cores) within each single GPU processor (called shared memory in NVidia terms).

CUDA is an extension to C programming language that makes it able to target NVidia GPUs. Execution on GPU starts when a section called the kernel is launched during CPU code execution. Kernels define the number of threads and blocks to be executed in that launch instance, the hardware scheduler assigns a block or multiple blocks to each processor, dispatching them as the execution of the kernel progresses. Threads in a single block are executed in groups of co-operative threads that NVidia calls warps. Some processors are capable of pipelining operations from several warps together. Operations in a single warp of threads happen at the same time. In the case where threads in a single warp are assigned different operations, those operations are serialized between threads of parting execution paths, where each group of certain path executes while the others are stalled waiting to be able to execute their own path of execution.

4.1.3 Review of barrier synchronization in GPUs

To support transparent scalability in GPU devices, manufacturers do not support barrier based synchronization from within the kernel. Instead, the approach recommended is to divide kernels around points where barriers are required, since the end of a kernel is a natural barrier for its threads.

In literature, however, other approaches were suggested and tested. Some approaches use atomic operations over global variables. One such approach also is to increment a counter by a single thread of each block and to force the blocks to wait for a specific value for the barrier to be accomplished. This approach, of course, has the drawback of a possibility of deadlock unless the number of blocks was less than or equal the number of Processors. Due to that, this approach is limited in concurrency and has a high possibility of causing a limitation into the devices occupancy.

Another approach is to synchronize with no atomic operations (locks). The avoidance of locks is a goal since locks are considered a serialization point that limits concurrency. One such approach [81] is to increment an array; the elements of which represents blocks. A single thread of each block sets the element representing that block. After that, those threads periodically check the values supposed to be set by the other threads. When threads detect that all participating blocks had reached the barrier, threads continue execution of the next steps in the code. This approach causes some contention on memory reads since many threads will read each element of the array. To solve this problem, another approach [78] adds a second array where after each block set its value in the first array, that block keeps checking only on its corresponding element in the second array. A designated block will then check for all values in the first array, then, after confirming that the barrier point has been accomplished, that block will use several threads to set all values in the second array for another block to be informed of the accomplishment of the barrier. This method, however, needs a fencing operation to enforce some form of memory consistency across all readers of array values. The authors, however, concluded that, since memory fencing is expensive in terms of time, it is

acceptable to use this method without the fencing operations on the architecture tested (before Fermi) without any impact on correctness. The authors also advised using the method on next generation architectures with or without fencing. Recently, inter-block synchronization approach has resurfaced and concluded to be the best approach to be used in iterative algorithms for linear algebra on Kepler and Xeon-phi architectures. However, the work does not mention of the consistency model enforcement. In this work, we re-evaluate kernel based barrier synchronization approach with performance when consistency is enforced in the code.

4.1.4 Jacobi iterative solver

In this work, we perform the test on Jacobi solver based on the work of [79]. We also implement our versions using PGI's implementation of OpenACC framework.

Jacobi solver is an iterative method for solving systems of linear equations. Usually, when systems of linear equations are large, it is preferable to use iterative methods to solve them. In iterative methods; a sequence is generated, and the algorithm keeps the iteration until a solution is reached that lies within some predefined error tolerance.

In Jacobi solver, an initial solution is assumed, and then the next solutions are calculated based on the equations:

$$x^{(0)} = (x_1^{(0)}, x_2^{(0)}, \cdots, x_n^{(0)})^{\mathrm{T}}$$

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \left(\sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k-1)}\right)\right], 1 \leq i \leq n$$

The first equation represents the initial state. Each element I in the solution vector in iteration k is calculated based on the second equation. Computationally, to avoid the burden of conditional statements while calculating the series sum, no checking for the

diagonal element will happen. However, the operation $a_{ii}x_i^{(k-1)}$ will be subtracted later from the total sum. Hence the computational method would the equation would become:

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \left(\sum_{j=1}^{n} a_{ij}x_j^{(k-1)}\right) + a_{ii}x_i^{(k-1)}\right], 1 \le i \le n$$

Listing 15 shows a sample implementation in C for the Jacobi solver. In each iteration step, the program performs a dot product that excludes the diagonal elements; hence the splitting for two loops around the diagonal. The serialization of iteration is mandatory for the correctness of the program. However, the inner loops that perform dot product can be parallelized without affecting correctness

Listing 15: The serial CPU code for Jacobi iterative solver.

```
01  for (iteration = 0; iteration < limit; iteration++) {
02      for (i = 0; i < n; i++) {
03          sum = 0;
04          for (j = 0; j < i; j++) /* compute summation of a[i][:]x[:] */
05              sum = sum + a[i * n + j] * x[j];
06          for (j=i+1; j < n; j++) /* compute summation of a[i][:]x[:] */
07              sum = sum + a[i * n + j] * x[j];
08          new_x[i] = (b[i] - sum) / a[i * n + i]; /* Update unknown */
09      }
10      for (i = 0; i < n; i++) /* update values */
11          x[i] = new_x[i];
12  }
```

4.1.5 Pipelined implementations on GPU

Kahn et al. [79] suggested a method that increases throughput in iterative solvers by using inter-iteration pipelining (they call it relaxed synchronization) they suggested the best way for barrier synchronization in this method is by using the lock-free synchronization method suggested by Xiao et al. [78]. Their implementation uses the persistent block to avoid the deadlock problem found in Xiao's problem, which limits the number of blocks in the kernel. Khan et al.'s first method for inter-iteration pipelined Jacobi (named RJ by them) is done by making blocks that finish their work traverse all other blocks and compute partially the next iteration based on ready parts of the vector x.

This is done via implementing vectors called presence vector for current and next iteration. The presence vector of iteration (k) registers computed results for corresponding blocks and is loaded into shared memory; information is extracted from it and loaded into a working vector. Based on their work vectors, threads of finished blocks repeatedly use partial inputs and make the partial calculations needed until the iteration (k+1) is finished. They will update presence vector needed for iteration (k+1) accordingly. Another method suggested by the authors (block-query Jacobi) is based on sequential repetitive polling of other blocks by each of the non-busy blocks. However, based on author tests, the first method (called RJ) performed best.

One noticeable observation is that the authors listed test results for the CPU-based synchronization only up to 128 blocks. Surely, the CPU-based synchronization method is not limited by the number of blocks the same way inter-block synchronization is. Limiting the number of blocks for CPU is acceptable only when trying to isolate the effect of inter-block synchronization regardless of the application involved, this will be done to measure the synchronization performance.

In our work, we will implement a CPU based synchronization method using OpenACC with and without limiting the number of blocks; we use the limited version in OpenACC to single out the performance of the synchronization method in the comparison.

4.1.6 Parallelization of Jacobi iterative solver

When parallelizing the Jacobi solver, data decomposition shall be determined to distribute the computation among different threads. To reduce needed communication as much as possible, we chose data decomposition based on the result vector. So, the responsibility of calculating the resulting vector will be distributed among different blocks of threads. Hence A one-dimensional decomposition of a computational grid would usually be sufficient to do this. Since the computation of some thread-blocks might finish before other blocks in a certain iteration. A barrier at the end of each iteration is needed to prevent thread certain thread block from getting a value $x^{(k-2)}$ in the

computation instead of $x^{(k-1)}$. Hence no block of threads shall be able to advance into a next iteration until all blocks of threads have finished the computation of the current iteration; a barrier at the end of the iteration allows this synchronization to occur. As discussed in section 4.1.3, the recommended practice in CUDA is to design the implementation in a way that divides the kernels into multiple zones around barrier points. In OpenACC, this is implemented naturally as part of the programming paradigm the same way. In the next few paragraphs, we progressively show some typical enhancements to such code that are usually discussed in GPU programming tutorials[3].

In order to avoid racing over eelements of vector x in iteration k by threads which are still using $x^{(k-1)}$ and threads which are producing new values $x_i^{(k)}$, there would be a need to separate storage of x into two arrays x and x_{new}, where x represents $x^{k-1}$and x_{new} represents $x^{(k)}$. Of course, after the kernel ends, the code needs to copy x_{new} into x after the implicit barrier and before the next iteration starts.

To avoid initializing two separate GPU kernels just because we exclude the diagonal element from the dot product in the Jacobi solver, we will initialize the sum into a minus value of the diagonal element's part of the dot product. This way, the code can be constructed in a way that does not split a single loop into two separate loops, which maps to two different kernels in the OpenACC model. An enhanced serial Jacobi is shown in Listing 16 below, where initializing the summation variable into the negative value of the diagonal element's part of the dot product would relief the programmer from splitting the iteration into two iterations around the diagonal. This enhancement would avoid the loop splits and would also reduce the number of launched kernels in the parallel port of the code. An OpenACC port of the code is shown in Listing 17 below. There, for each iteration step, the OpenACC compiler will create two GPU kernels; one for the matrix-vector product iteration nest in line 04, and another one for copying new_x back into x we can find in iteration on line 11.

---

[3] One example listing some of these enhancements is Jeff larkin's Getting started with OpenACC tutorial, GpuTech conference , 2013 , http://on-demand.gputechconf.com/gtc/2013/presentations/S3076-Getting-Started-with-OpenACC.pdf

Listing 16: Enhanced Jacobi serial CPU code.

```
01  for (iteration = 0; iteration < limit; iteration++) {
02        for (i = 0; i < n; i++) {
03               sum = -a[i*n+i] * x[i];
04               for (j = 0; j < n; j++) /* compute summation of a[i][:]x[:] */
05                      sum = sum + a[i * n + j] * x[j];
06               new_x[i] = (b[i] - sum) / a[i * n + i]; /* Update unknown */
07        }
08        for (i = 0; i < n; i++) /* update values */
09               x[i] = new_x[i];
10  }
```

Listing 17: OpenACC Directives for the Jacobi solver in Listing 16.

```
01  #pragma acc data copyin(a[0:n*n], new_x[0:n], b[0:n]), copy(x[0:n])
02      for (iteration = 0; iteration < limit; iteration++) {
03  #pragma acc kernels present_or_copyin(a[0:n*n], b[0:n], new_x[0:n]),
    present_or_copyout(x[0:n] )
04          for (i = 0; i < n; i++) {
05              sum = -a[i*n+i] * x[i];//0;
06              for (j = 0; j < n; j++) /* compute summation of a[i][:]x[:] */
07                  sum = sum + a[i * n + j] * x[j];
08              new_x[i] = (b[i] - sum )/ a[i * n + i]; /* Update unknown */
09          }
10  #pragma acc kernels present_or_copy(x[0:n]), present_or_copyin(new_x[0:n])
11          for (i = 0; i < n; i++) /* update values */
12              x[i] = new_x[i];
13      }
```

Another possible enhancement over the code in Listing 17 above is the elimination of the copy kernel in line 11. Since the program already separates the storage of x and needs to copy x_{new} into x, we can save the time of copying vector x_{new} into the vector x in the following manner. Since x_{new} contains the value $x^{(k)}$, initiating another kernel after the first kernel of line 04 to calculate $x^{(k+1)}$ from x_{new} is possible. For that we save the answer vector sequence $x^{(k+1)}$ into variable x, the input for the next iteration will be ready for the next iteration without a copy kernel. This way each iteration cycle would calculate two successive values of x. The only drawback in this method is that the iteration can only halt after a stride of two instead of the usual strides of one. The enhancement mentioned is listed in Listing 18

Listing 18: Optimized OpenACC code for the Jacobi solver code in Listing 17.

```
01  #pragma acc data copyin(a[0:n*n], new_x[0:n], b[0:n]), copy(x[0:n])
02  for (iteration = 0; iteration < limit; iteration +=2) {
03  #pragma acc kernels  present_or_copyin(a[0:n*n], b[0:n], new_x[0:n]),
    present_or_copyout(x[0:n] )
04        for (i = 0; i < n; i++) {
05              sum = -a[i * n + i] * x[i]; //0;
06              for (j = 0; j < n; j++) /* compute summation of a[i][:]x[:] */
07                    sum = sum + a[i * n + j] * x[j];
08              new_x[i] = (b[i] - sum) / a[i * n + i]; /* Update unknown */
09        }
10  #pragma acc kernels present_or_copyin(a[0:n*n], b[0:n], new_x[0:n]),
    present_or_copyout(x[0:n] )
11        for (i = 0; i < n; i++) {
12              sum = -a[i * n + i] * new_x[i]; //0;
13              for (j = 0; j < n; j++) /* compute summation of a[i][:]x[:] */
14                    sum = sum + a[i * n + j] * new_x[j];
15              x[i] = (b[i] - sum) / a[i * n + i]; /* Update unknown */
16        }
17  }
```

4.2 Validating the use of inter-block synchronization techniques for iterative solvers

In this section, we demonstrate our findings for testing inter-block synchronization in native CUDA code using the method described in [78]; versus the way of synchronization used by OpenACC codes and mainstream CUDA kernels, which depends on the GPU. The motivation behind these tests was to evaluate the performance effect of inter-iteration producer-consumer synchronization within the GPU when used on Jacobi solver (which is called **RJ-CUDA**). We also test the performance of the Jacobi solver without inter-iteration producer-consumer synchronization (**CUDA-SJ**), and a chaotic version that works without synchronization (**CUDA-AJ**) and (**ACC-AJ**). However, the ACC-AJ version was not implemented in a completely chaotic manner. Rather the only synchronization that was skipped is between each two consecutive loop iteration cycles using the *async* directive. That would effectively remove half the expected synchronization barriers. Implementing a chaotic version in OpenACC needs as many async streams as the iteration cycles, but this is not possible since there is a limitation on async streams concurrently launched.

It can be seen from Figure 16 below that OpenACC implementations, both synchronous (**ACC-SJ**) and asynchronous demonstrated the best performance versus CUDA implementations that implements inter-block synchronization (SJ-CUDA and RJ-CUDA) and versus CUDA implementation that ignores any type of synchronization (AJ-CUDA).

This results was due to optimizations that are done in OpenACC, in addition to the fact that inter-block synchronization schemes are forced to launch a specific number of blocks to avoid deadlock among blocks. This approach to preventing deadlocks in inter-block synchronization is described in the original author`s paper [78].

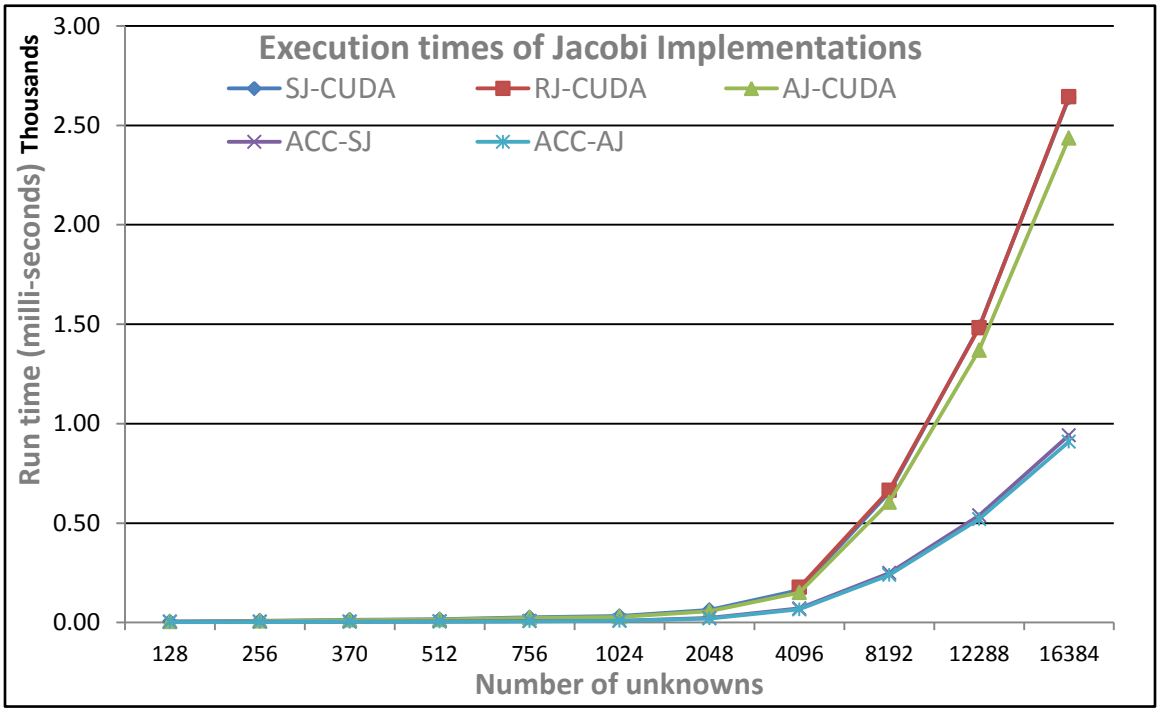

Figure 16: Execution time of 100 iterations of Jacobi solver for various implementations.

As OpenACC performed in Jacobi iterations better than CUDA inter-block synchronization techniques; we unified the parameters of kernel launches between the CUDA code and OpenACC. We did this to isolate the effect of inter-block synchronization on performance compared to the normal approach that uses out of kernel synchronization when all other parameters are unified and to eliminate the possibility of the thread granularity being the cause of the performance difference. This isolation would

give an impression of whether to motivate other work to alleviate obstacles relating to deadlocks in inter-block synchronization. However, we only succeeded in fixing the size of the block, as fixing the grid size (num_gangs) was not being obeyed by the compiler (PGI).

The behavior of the ACC-SJ here resembles the best practices of a code of such type (aside from the manually specified grid and block sizes) in the following terms

As the program needs to wait before updating X, the standard practice involves writing into another array (x_{new}). After all threads finishes; usually (x_{new}) is copied back to x. However; we took the approach of using the copy-back kernel for calculating the next iteration and storing its values in X.

The OpenACC model depends on launching a different kernel launch for each parallel (or kernels) region. This fact is the only way to be able to do synchronization across blocks. However, OpenACC provides data regions construct to be able to do several launches in a loop without repeating data copies between host and device. Depending on the profiling trace data, we found out that each synchronization operation is consuming between (0.025 and 0.047) milliseconds on the system under test.

Regarding asynchronous method, we enclosed the iteration loop in a parallel region that is specified to be sequential, with the loops inside it each having the loop directives; This way it surpassed the method that depends on two asynchronous kernel launches with a single kernel launch operating without barriers or synchronizations.

The number of iterations for each run could not exceed 255 iterations when CUDA-RJ is involved since it uses an unsigned char data type for the number of iterations in its internal workings. All CUDA implementations were run with a block size of 32 threads and a grid size of 128. All OpenACC versions were run with a block size of 32 threads and a grid size of (n) blocks, where n is the dimension of the matrix.

Table 4 below shows run times and corresponding speedups over CUDA-SJ after the unification of the number of blocks (gangs) to be on par with the each other for the

approaches tested. Figure 17 shows the speedup. The solver was run for 254 iterations in this experiment. We can now see that when it comes to performance while disregarding the restrictions. Out of kernel synchronization and inter-block synchronization are almost the same. Moreover, the ability of CUDA-AJ to perform better is because it combines the performance benefit of iterating from within the kernel and not performing any type of synchronization. Here we can notice that all the implementations are close together in performance except for the CUDA-AJ (which is a chaotic algorithm that ignores synchronization). The little performance for ACC implementation was due to kernel launch overhead added to the fact that we restricted the number of blocks to assess the synchronization performance.

Table 4: Comparison of Runtime and the speedup over CUDA-SJ against data size

	Run time per data size in ms (254 iterations)				speedup per data size over CUDA-SJ (254 iterations)			
Number of unkowns	4096	8192	12288	16384	4096	8192	12288	16384
CUDA-SJ	418.37	1675.29	3795.31	6744.59	1.00	1.00	1.00	1.00
CUDA-RJ	448.00	1686.45	3763.39	6731.92	0.93	0.99	1.01	1.00
CUDA-AJ	150.30	604.43	1361.93	2422.33	2.78	2.77	2.79	2.78
ACC-SJ	477.39	1820.01	4038.44	7133.13	0.88	0.92	0.94	0.95
ACC-AJ	455.73	1766.10	3939.23	6948.15	0.92	0.95	0.96	0.97

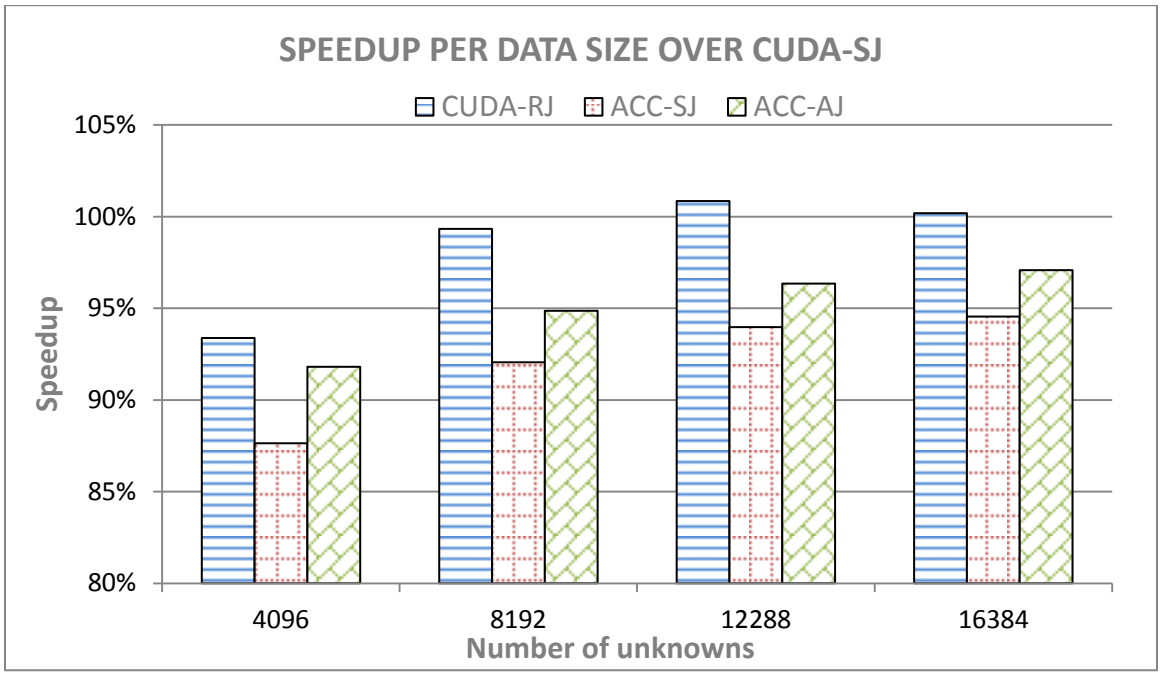

Figure 17: Speedup for various implementations of Jacobi iteration over CUDA-SJ.

We also lowered the number of iterations from 254 to 100 iterations to see the effect of varying the number of iterations on the difference in performance. It is clear from Figure 18 that the performance difference between these synchronization primitives is negligible. Moreover, the CUDA versions are inferior to the OpenACC version since OpenACC is not bounded by a certain granularity for the number of gangs.

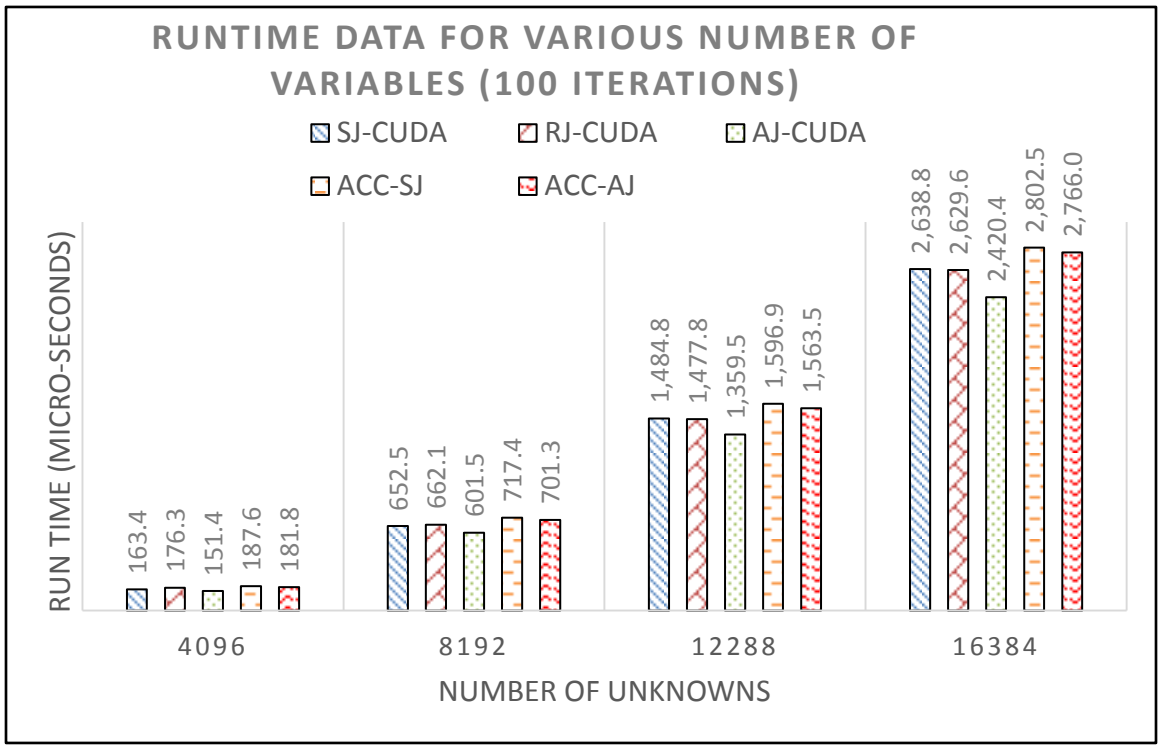

Figure 18: Execution time Comparison for 100 iterations of Jacobi solver.

4.3 Analysis of GPU Inter-block wall-barriers

4.3.1 Modeling wall barrier inter-block synchronization in GPUs

We will start our analysis with building a model for the wall barrier operation in GPUs in the model we develop here; we consider a GPU SM as a resource and consider a block as a process that requests a single SM resource. Blocks get assigned SMs where they are executed non-preemptively until they are finished. We define the number of **resident blocks** as the number of blocks that are assigned to an SM after dispatch so the execution scheduler can execute them concurrently on SMs. We hence define the **residency** of an SM as the maximum number of blocks that can reside on a certain SM.

Notice that in our model, the following are true:

1. It is guaranteed that a block requests an SM the moment it is launched
2. Since we are discussing wall barrier inter-block synchronization for blocks in the same kernel (identical blocks), it is guaranteed that once a block is assigned an SM, it will ask for a rendezvous at some point in time, expecting to meet all other blocks.
3. For multiple different synchronization points (barriers) in the code of a block, a block cannot reach a synchronization point unless all previous rendezvous synchronization points had been accomplished without a failure.

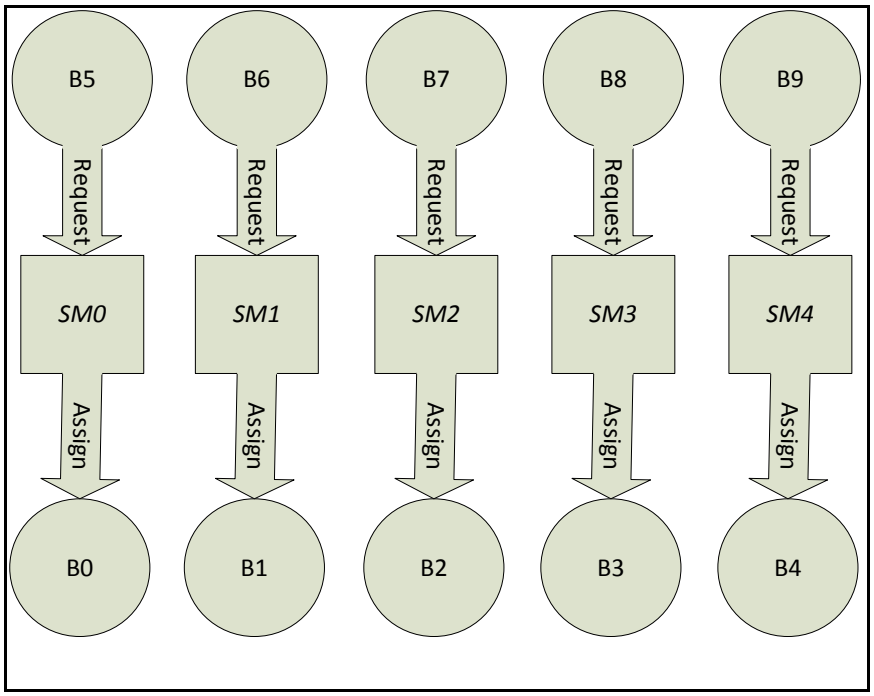

Figure 19: An illustrative example of resource allocation (SM as a resource).

Figure 19 above illustrates an example resource assignment for a kernel having a number of blocks that is two times the number of SMs. That example assumes basic SM architecture without pipelining capability among resident blocks. The illustration Figure 19 could be expressed as a graph that would be adequate for system analysis. Our goal with such graphs is to provide an adequate platform for studying the state of a system in at various points of kernel execution.

A resource allocation graph is shown in Figure 20. In a resource allocation graph, a directed edge with a block as a source and an SM (resource) as a destination represents a request to be fulfilled (a block waiting to start execution). On the other hand, an edge with an SM resource as a source and a block as a destination represents an assignment where a block is already executing on an SM (resource already assigned). In a RAG, all edges are directed between a resource and a process. To represent a direct relationship between processes in a system, a wait-for-graph (WFG) is deduced from a RAG. Deducing a WFG from a RAG is done by removing resources and collapsing appropriate edges of the resulting graph. A wait for graph is helpful to detect an unsafe system state easily; this will be illustrated shortly when we define cycles. Figure 21 shows the WFG corresponding to the RAG shown in Figure 20.

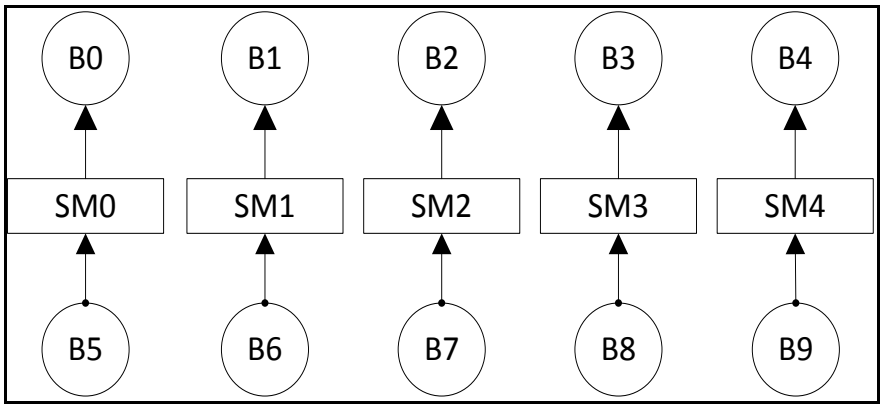

Figure 20: RAG (Resource Allocation Graph) corresponding to illustration in Figure 19

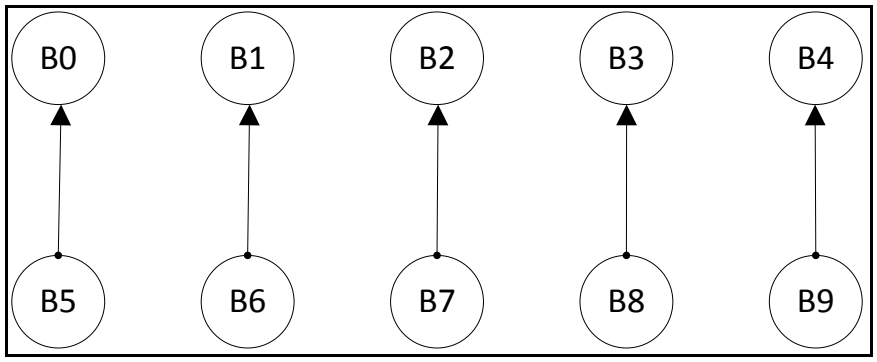

Figure 21: WFG (Wait For Graph) corresponding to RAG in Figure 20

To model a wall barrier in a RAG or WFG, we need to introduce the *consumable resource,* a consumable resource is a resource that once assigned, it will vanish. Hence there is no need for the directed edge tailed at the resource since such state will not exist. Instead, for a consumable resource, an edge tailed at a resource R_c into some process B_n means that resource R_c will be produced by process B_n. A consumable resource is indicated in RAG and WFG by a square contained within another square.

When a wall barrier is implemented in software, a shared counter or signal usually implements it. Each process that comes to the rendezvous increases the shared counter or sets a flag, and then waits for the counter to reach an expected value set to the total number of processes (Blocks). Because of that, the last process reaching the rendezvous is the one that sets the counter to the desired value or sets the last flag needed for the rendezvous to complete. This behaviour allows us to model the last process reaching the rendezvous as a producer for a signal, while the rest being consumers for that signal. That signal is the value desired for the counter or the flag to be set by the last process reaching the rendezvous. Notice that wall barriers in the GPU case are blocking, meaning that processes cannot go on with their execution until they receive the signal.

Now, we will define an expedient state of a system. An **expedient** state is a state in which all requesting edges in a RAG of a system are blocking requests, meaning processes requesting resources will not continue execution until they are resolved.

"A cycle C in a graph is a path who's first and last nodes are the same"[4]

"A knot K in a graph is a nonempty set of nodes such that for every node x in K, all nodes in K and only the nodes in K are reachable from x"[5].

Another definition needed preemptive resources. A preemptive resource is a one which can be temporarily re-assigned to another process, in which case the state of such resource is preserved for re-assignment for the process. In GPUs, an SM is a **non-**

[4] Tong Lai Yu, March 2010 , http://cse.csusb.edu/tongyu/courses/cs660/notes/deadlock.php
[5] Tong Lai Yu, March 2010 , http://cse.csusb.edu/tongyu/courses/cs660/notes/deadlock.php

preemptive resource in most cases, meaning that once allocated to a block, an SM will not be taken away from that block until the block has finished execution. Figure 22 demonstrates how a deadlock could happen in such a system. Since a wall barrier rendezvous can be thought of as a single producer multi-consumer model for a signal as explained earlier, and since the last block to reach a rendezvous is the producer for that signal; then block B1, waiting for execution is guaranteed to be the producer of that signal (the rendezvous resource).

For simplicity, the illustrated example in Figure 22 illustrates a GPU having a single SM and two blocks, with single block Residency for the SM. Both RAG (left) and WFG (right) are shown. In GPU kernels having more blocks than the total sum of SM Residency, blocks will wait for an SM to become available. However, execution of blocks is not preemptive in GPU SMs; hence our simplified example of one SM is sufficient for demonstration, where it shows a single SM system with single residency and a two-block kernel. A cycle is very clear in the WFG of the system, indicating a deadlock state. The explanation here is that a wall barrier that is found inside the code of the kernel is executed by block 0, which causes it to wait for block 1. However, block one will never get that chance to execute the barrier code in this system.

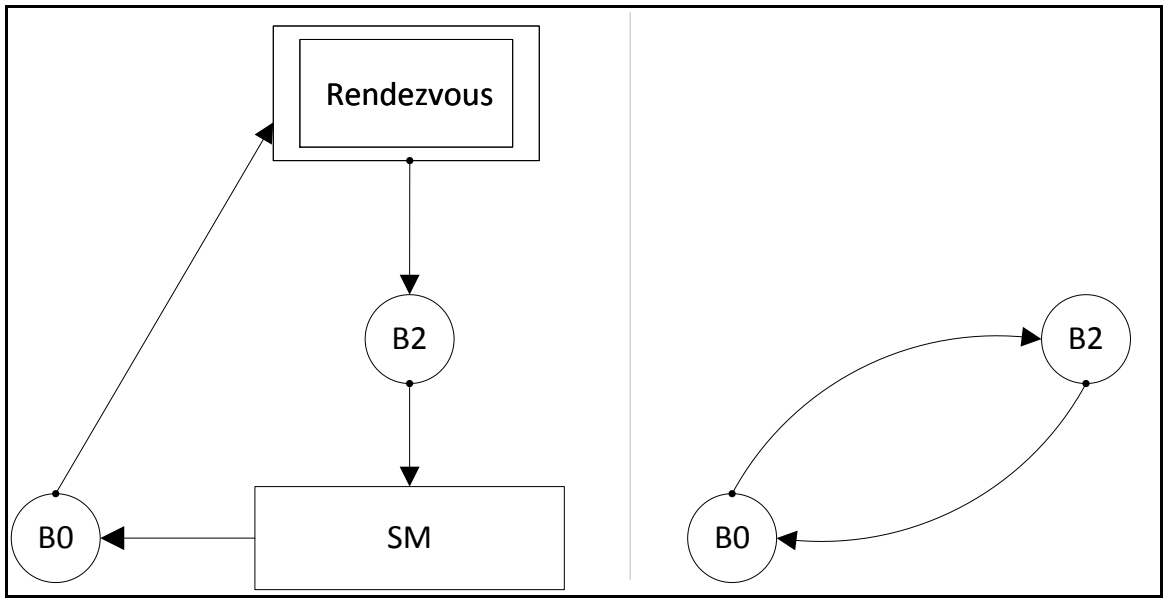

Figure 22: a RAG and WFG for a wall barrier. A cycle indicates a deadlock state.

The example in Figure 22 can be expanded for resources with multiple instances, such as SMs with multiple-block residency. **However, in such systems, states must not be simplified into a wait-for graph.** In resource allocation graphs, there is no difference between an SM having a residency larger than one and the existence of multiple SMs. The graph and analysis of such systems are the same for detecting unsafe or deadlock states. Figure 23 shows a RAG for a system with two SMs (or a single SM with a block residency capability of two). The figure shows that since there are three blocks in the system and only two SM resources, a knot would happen in case blocks tried to hold global wall barrier rendezvous synchronization). Notice that the analysis for such system cannot be done on a WFG since the system has resource replication.

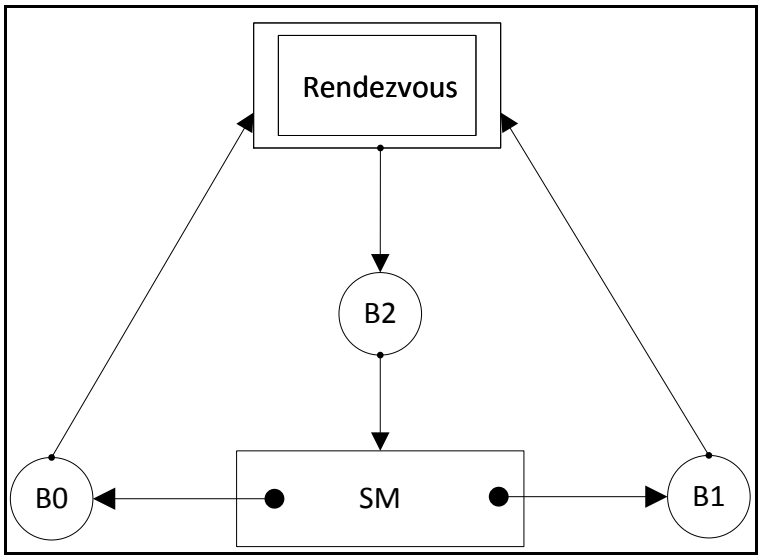

Figure 23: An example of a deadlock system state for a GPU system with SM replication

4.3.2 Deadlock avoidance and detection in GPU inter-block wall barrier synchronization

For a deadlock to occur, the following necessary conditions should apply to system properties:

1. Mutual exclusion, where a resource instance can only be allocated to a single or a limited number of processes.
2. Processes can ask for new resources without releasing resources which they are holding.
3. The existence of non-preemptive reusable resources

Current GPU systems have all these conditions described above since an SM is a reusable resource that can only be allocated to a limited number of blocks, blocks can ask for a resource (in our case the rendezvous resource) without releasing the SM. Finally, SMs are not preemptive, i.e. once a block started execution, it needs to finish before it releases an SM. This applies to cases that do not include dynamic programming. For such systems, we also add the following two conditions for a deadlock to occur. **These conditions with the previous three make a list of necessary and sufficient conditions for a deadlock**:

4. For systems without resource replication, a cycle in the RAG or WFG is a necessary and sufficient condition for a deadlock. Figure 22 shows an example of a state of the system containing a cycle.
5. For systems with resource replication, of which some resources have more than one instance; a knot is a necessary and sufficient condition for a deadlock. Figure 23 shows an example of a GPU system with a knot.

To avoid a deadlock, one needs to break one of these necessary and sufficient conditions. However, it is not possible to change the first three conditions above to avoid deadlocks; since they are inherent in the architecture of current GPUs. The only way to avoid a deadlock is to prevent knots and cycles from happening, i.e. to break conditions 4 or 5. In a wall barrier, a knot is guaranteed to happen if the number of blocks to be launched by a kernel is to exceed the number of SM resource instances. Figure 24 below shows an example of a system state where a kernel only had launched the same number of blocks as the number of SM instances. If the number of blocks is less than or equal to the

number of SM instances, it is impossible for a knot to happen in the system even when a rendezvous-type wall barrier is requested

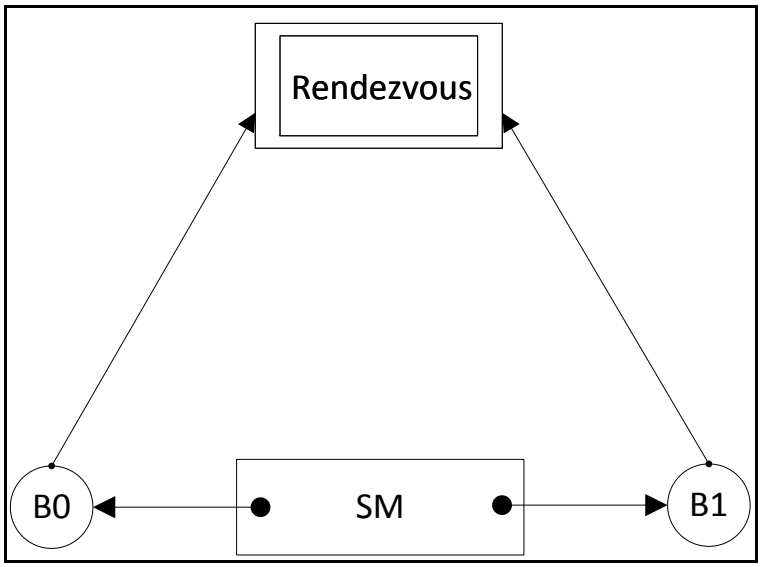

Figure 24: a system is executing a wall-barrier in a safe state.

In this section, we addressed part of the analysis we deem required for studying in-GPU synchronization techniques. However, we still need to shed more light on GPU systems that allow multiple blocks to reside on a single SM. For analysis of such systems, one needs to replicate the SMs resource per the number of blocks residing on them. We explain how exactly to mathematically do so in the next section.

4.3.3 Deadlock in inter-block synchronization on modern GPU architectures

Recent GPU hardware tremendously enhanced multi-processor capabilities in concurrency. These capabilities allow schedulers in multiprocessors to handle multiple warps from several blocks. For example, NVidia's Kepler architecture multiprocessors can concurrently handle the dispatch and execution of an instruction from four independent warps; as they have four warp schedulers and eight instruction dispatch units

each. The warp scheduler selects four warps, where two instructions per warp can be dispatched in each cycle. An enhancement over NVidia's Fermi architecture, some Kepler chips such as the GK110 allows double precision instructions to be dispatched with other instructions.

The improvement mentioned above extends the widely-known concept of multiprocessor occupancy into the desire to occupy each multiprocessor with maximum blocks possible.

The maximum number of blocks a processor can handle simultaneously (currently only regarding scheduling and dispatch) is called maximum residency. Kepler's compute capability 3.5 multiprocessors have a maximum residency of 16. While the ones with compute capability 5.0 have a maximum residency 32 blocks.

However, reaching maximum block residency of a multiprocessor is not always possible; because there are other variables involved that need to be satisfied. Table 5 shows some of the variables that usually are taken into consideration when programming GPUs with CUDA, including warp and thread Residency. Notice that the table contains the term "compute capability", which simply refers to the hardware generations of the NVidia GPU.

From the information mentioned above, one can see that in inter-block synchronization, one cannot easily determine whether a certain number of blocks in a kernel would or would not get into deadlock during inter-block synchronization. However, before we get into such analysis, we would like to provide a maximum number of blocks for a deadlock during inter-block synchronization. We define a **multiprocessor residency** (**SMR**) as the maximum number of blocks that can simultaneously reside in execution state on that multiprocessor, regardless of other limiting factors such as shared memory usage. We also define the **resident blocks** (**RBSM**) of an SM as the number of blocks that the SM can practically handle for a specific kernel. Since resources available per GPU change per architecture, we define the **Maximal-Residency** ($\mathbf{R_M}$) for a certain GPU G that contains NSM of Symmetric Multiprocessors of non-preemptive scheduling architecture, as the

maximum number of blocks that can be executed without a deadlock during inter-block synchronization. R_M can be computed using the formula:

$$\mathbf{R_M = NSM_G \times SMR_G}$$

Maximal Residency only tells us the number of blocks that if exceeded for a certain GPU hardware generation would cause a deadlock during execution of a wall-barrier inter-block synchronization regardless of the kernel application used. For analyzing specific applications over the same GPU, we define the **residency limit $\mathbf{R_L}$** for an application kernel instance K on a GPU G as:

$$\mathbf{R_L = RBSM_K \times NSM_G}$$

Notice that residency limit would never excel maximal residency for a GPU. However, if we want to find a definite number for a certain application, we need to analyze processor occupancy for that specific application. As an example, we will demonstrate the occupancy-deadlock tradeoff on inter-block synchronized Jacobi done by Khan et.al [79].

4.3.4 Deadlock-occupancy tradeoff analysis in inter-block GPU synchronization

In this section, we demonstrate occupancy analysis to find the maximal residency number for the Jacobi implementation done with inter-block synchronization. That implementation was done in a way that accomplishes the iterative application in a single kernel launch.

The first number we need to calculate is the per-block shared memory size used by the kernel, along with the number of threads used per register. The application at our hands uses tiling with a (block size × block size) tile of floating point type stored in shared memory. Hence, for an instance with block size of 32, the shared memory used in example kernel = 32 × 33 × sizeof (float) = 32 × 33 × 4 = 4224 bytes.

However, the allocation unit size for the shared memory in compute capability 3.5 is 256, hence the shared memory size allocated per (**SMEM_ALLOC)** would be

SMEM_ALLOC =

$$\left\lceil \frac{shared\ memory\ used\ per\ block}{shared\ memory\ unit\ allocation\ size} \right\rceil \times shared\ memory\ unit\ allocation\ size$$

$\left\lceil \frac{4224}{256} \right\rceil \times 256 = 4353\ bytes.$

After that, we can divide the configured shared memory size for the SMs by the used shared memory to calculate the limit on processor block occupancy imposed by shared memory usage. Notice that some GPUs allow the size of shared memory per multiprocessor **SMEM_SM** to be configured with several predetermined values:

Thus, the residency limit due to shared-memory capacity (**RL_SMEM)** =

$$\left\lfloor \frac{SMEM\_SM}{SMEM\_ALLOC} \right\rfloor \times SMEM\_ALLOC$$

$= \left\lfloor \frac{49152}{4353} \right\rfloor = \lfloor 11.29 \rfloor = 11\ blocks.$

Another arresting factor for SM residency the limit imposed by the number of warps to be launched on a GPU multiprocessor (Our launch configuration uses a block dimension of 32 threads):

Number of warps per block (NWARPS) $= \left\lceil \frac{block\ size}{Wapr\ size\ for\ GPU} \right\rceil = \left\lceil \frac{32}{32} \right\rceil = 1$.

Thus, we can calculate SM residency due to warp limit of architecture (**RL_WL**) = $\frac{warp\ limit\ per\ sm}{NWARPS} = \frac{64}{1} = 64$ blocks.

One final calculation needed depends on registers allocated per thread, which also limits the number of warps the SM can. Although our example kernel uses few registers, we

will list this calculation for showing the analysis required. Notice that at this stage, the easiest way to get the register use is from the compiler output during compilation.

Our example application uses 39 registers per thread (**RPT**). For the GPU we are using, register file allocation is handled in chunks called register allocation unit size (**RAU**) of 256.Multiprocessors of compute capability 3.5 have a register file Warp allocation granularity of four (**WAU**), meaning that they allocate register resources for four Warps at a time with no possibility of partial allocation.

Thus, to calculate the residency limit due to register use (RLR) we need to calculate the number of warps we are able to launch based on register use (**WL_REG**):

WL_REG =

$$\left\lfloor \frac{\text{Total registers per multiprocessor}}{\frac{\left\lceil \frac{\text{RPT} \times \text{Warp size}}{\text{RAU}} \right\rceil \times \text{RAU}}{\text{WAU}}} \right\rfloor \times \text{WAU}$$

$$= \left\lfloor \frac{\frac{65536}{\left\lceil \frac{39 \times 32}{265} \right\rceil \times 265}}{4} \right\rfloor \times 4 = 48 \text{ warps.}$$

Hence residency limit per SM imposed due to register usage (**RL_RU**) is

$$\left\lfloor \frac{WL\_REG}{Warps\ per\ block} \right\rfloor = \left\lfloor \frac{48}{\frac{32}{32}} \right\rfloor = 48.$$

Finally, the Bocks concurrently resident in an SM would be

RBSM = min {RL_RU, RL_WL, RL_SMEM, SMR}.

Hence our example kernel (K) when launched on the compute 3.5 K20c architecture would have a residency limit of:

$\mathbf{RBSM_K}$ = Min {48, 64, 11, 16} = 11 blocks.

We now calculate the number of blocks per multiprocessor that would not cause a deadlock during a rendezvous wall barrier inter-block synchronization, which would be total resident blocks in GPU. For our example application, shared memory imposes the residency limit. Since tile shared size in the tiling optimization (which is essential for performance) in the application depends on Block size. When running on a NVIDIA k20 model card, the actual residency limit for the launch configuration specified in our kernel (k) above would be: $R_L = \mathbf{RBSM_K} \times \mathbf{NSM_G} = 14 \times 11 = 154$ blocks.

In this section, we analyzed the aspect of residency in GPU multiprocessors. This aspect helps us to be able to judge correct parameters when managing resources in GPU programs. We applied the case here to global synchronization inside the GPU. For that, we need to apply the analysis in the previous section while replicating the SM resource to the same number as the residency of the GPU. The act of limiting launched blocks in a system will in many times limit the machine occupancy hence the performance of the application on the GPU. Therefore, in section 4.5, we propose a new method free of that limitation; relevant to certain types of synchronization.

In section 4.7, we will see an example of how we can use software instrumentation tools to measure the residency of a GPU application.

Table 5: Partial NVidia GPU hardware Technical Specifications for all Compute Capabilities (hardware generations) available [8]

Specifications for various NVidia GPU architecture versions	GPU architecture version (Compute Capability)							
Technical Specifications	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum number of resident grids per device [6]	16		4	32				16
Maximum dimensionality of grid of thread blocks	3							
Maximum x-dimension of a grid of thread blocks	65535	$2^{31}-1$						
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	1024							
Maximum z-dimension of a block	64							

[6] Concurrent Kernel Execution - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#concurrent-kernel-execution

Specifications for various NVidia GPU architecture versions	GPU architecture version (Compute Capability)							
Technical Specifications	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum number of threads per block	1024							
Warp size	32							
Maximum number of resident blocks per multiprocessor	8	16				32		
Maximum number of resident warps per multiprocessor	48	64						
Maximum number of resident threads per multiprocessor	1536	2048						
Number of 32-bit registers per multiprocessor	32 K	64 K			128 K	64 K		
Maximum number of 32-bit registers per thread block	32 K	64 K						32 K
Maximum number of 32-bit registers per thread	63	255						

Specifications for various NVidia GPU architecture versions	GPU architecture version (Compute Capability)							
Technical Specifications	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum amount of shared memory per multiprocessor	48 KB				112 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block	48 KB							
Number of shared memory banks	32							
Amount of local memory per thread	512 KB							
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB					10 KB		
Cache working set per multiprocessor for texture memory	12 KB	Between 12 KB and 48 KB						
Maximum number of instructions per kernel	512 million							

4.4 Analysis of the GPU block scheduler

To analyze what's happening in the rendezvous wall barrier, it is of importance to understand how the block scheduler works Inside the GPU. However, there are no official or published resources explaining the working of the block scheduler in regards to assigning blocks to multiprocessors within a single kernel [82, Sec. 6.3]. This lack of documentation is understandable, since researching the assignment of independent blocks into independent Multiprocessors is likely to be only valuable to the designers and implementers of the specific hardware at hand. However, the block assignment into multiprocessors (SMs) and the ordering of such assignment is knowledge of value in our research, since the block execution order –and hence the block assignment to multiprocessors is of value in iterative applications.

In this section, we will analyze how the block assignment scheduler assigns blocks into SMs. Some work has been done for scheduling concurrent kernels in GPU, or dynamic block creation for kernels. However, white papers and documentation resources have no mention of the block assignment to SMs [82, Sec. 6.3]. So, we need to reverse engineer kernel execution to reveal what we need to know about the block scheduler behavior in this regard.

To reverse engineer block assignment in the GPU, we need to know at runtime the block into SM assignment. Unfortunately, CUDA API from NVidia does not provide the ability to retrieve such information.

One would assume that depending on clock values retrieved using CUDA API inside threads would be of help, but clock values are defined to be per-multiprocessor [8]. Hence the clock values would not be sufficient to yield information about block assignment per processor. However, if combined with other information that will follow shortly, it will give a significant insight of processor numbering, clustering and block execution ordering within a single or several multiprocessors.

Thankfully, C language provides the ability to mix assembly language statements. Since there is a register in each block that holds the value of the ID of the multiprocessor, we can retrieve the value from already running threads. This value, along with our knowledge of the block index from the API gives a glimpse of the block assignment into SMs during the kernel launch configuration applied. For that purpose, we will adapt some code to do this mapping that was designed to analyze the NVidia Fermi block scheduling behavior[7].

We run a program that would run a dummy load by each block and provide a mapping between block number values and SMID register values that were read using a PTX assembly statement and logged during launch by each block. The example application was run with a launch configuration of 40000 blocks. The Resulting mapping between blocks and SM IDs is shown in Figure 25. Notice that block assignment into multiprocessors stays regular until the maximal residency of the GPU is reached (224 blocks at block ID 223). After the maximal residency is reached, blocks are dispatched to multiprocessors that finish execution blocks, i.e. based on availability.

Notice how SM IDs had a very clear pattern relating to the Block IDs up to but excluding block number 224. starting at block number 224, there was an apparent loss of a clear pattern for Block ID to SM mapping. We note that this number is identical to the *Maximal residency* number for the K20 NVidia GPU board where the experiment was conducted.

To understand why the maximal residency had resurfaced in our analysis, it helps to understand that the dummy kernel load coded had no significant effect on shared memory or warp register use since it only had one warp per block. Hence, the multi-processor residency for that kernel was at its maximum value. The start and end timings of blocks have a value that will shortly come to sight, as it will show the reason for the disturbance after the GPU residency hit its limits.

---

[7] How the Fermi Thread Block Scheduler Works , http://users.ices.utexas.edu/~sreepai/fermi-tbs/

The observed scheduling pattern for block numbers that are less than actual kernel residency limit R_L on a specific GPU would as follows:

$$SMID = \text{N}SMs - reminder\ of\ \left(\frac{BlockID}{number\ of\ SMs\ in\ GPU}\right), \text{BlockID} < \text{RL}.$$

NSMs represents the number of SMs (GPU multiprocessors) in the target GPU platform. So, the blocks come to be distributed in a round-robin descending fashion until the R_L has been reached. After which, it is expected that it is based on availability

The clock values reveal another set of interesting information about the block assignment. We run a test where each block records the clock start values of its execution this way we can deduce the dispatching behavior of the GPU. It can be noticed from Figure 26 that blocks are dispatched in a monotonically increasing order. Another interesting observation is the fact that there is a clear distinction in clock values at intervals of 224; which is the maximal residency limit of the GPU under test, the dummy kernel tested is a one with good load balancing profile. Therefore, the residency limit is observed with a clear-cut jump in clock start values. This observation is further validated by observing the graph in Figure 27; where block ID values are normalized by residency limit of the kernel and are plotted against their starting clock cycle values. Observations for these two figures should further clarify the connection between the residency limit and deadlock in global synchronization since blocks clearly cannot have a global rendezvous-style synchronization approach since their execution timelines do not overlap.

Finally, we illustrate visualization for the approximate block scheduling behavior in Figure 28. We assume a GPU with 4 SMs having a maximal residency $\mathbf{R_M}$ of 16 blocks (four per multi-processor). We also assume the illustration kernel does not cause any arresting limitations on the residency of SMs, i.e. $\mathbf{R_L} = \mathbf{R_M}$. First, blocks with IDs less than R_L are dispatched in a round robin manner. After that, blocks are queued for dispatch waiting for availability of an SM. The reader should be able to notice that the only way block 16 could start execution in this non-preemptive system is if and only if a

resident block completes execution. This means that one of the blocks resident in SMs 0 to 3 will never coincide with any of the blocks 16 to 21.

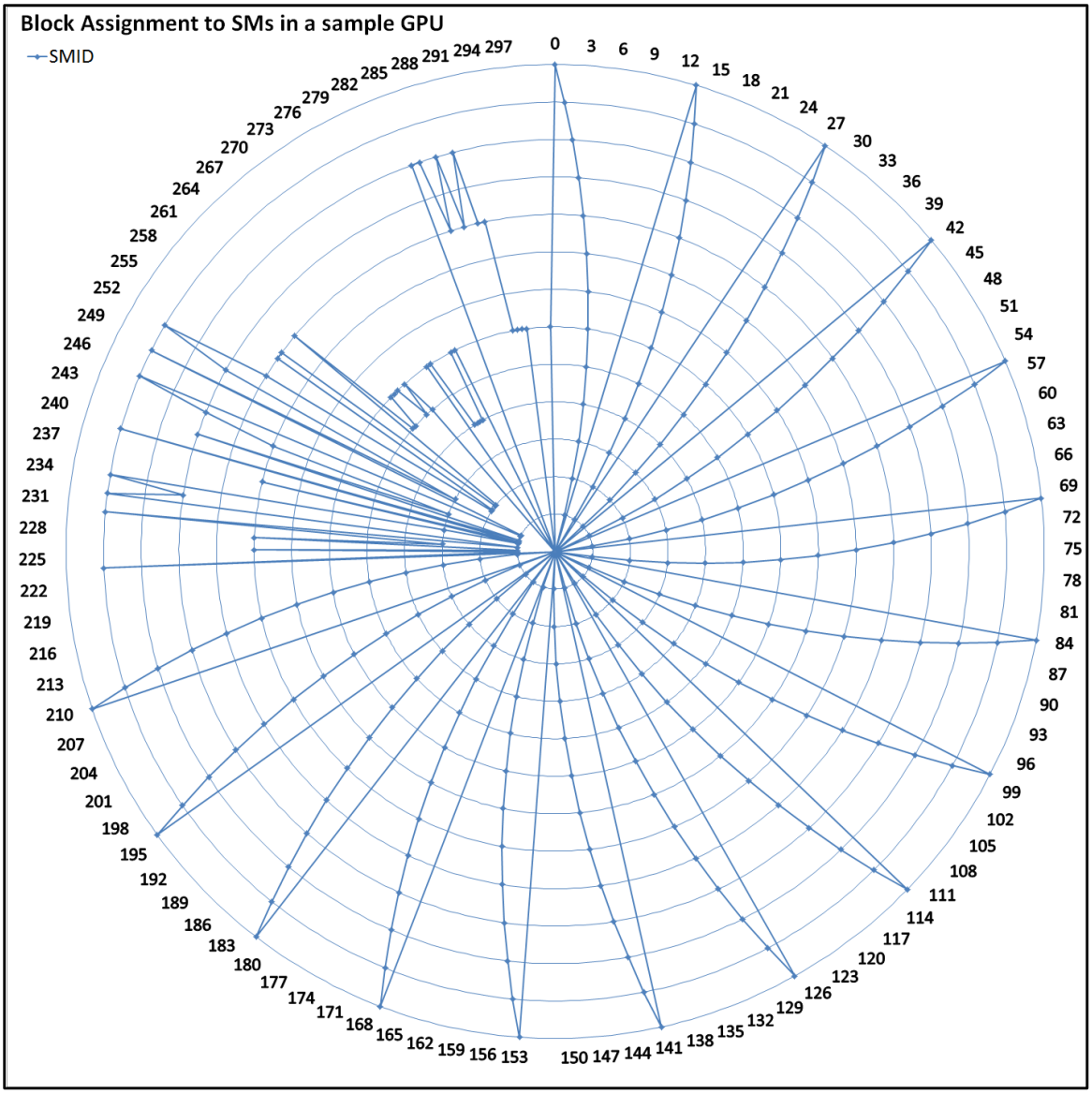

Figure 25: SM to Block Assignment in the K20 GPU (14 SMs)

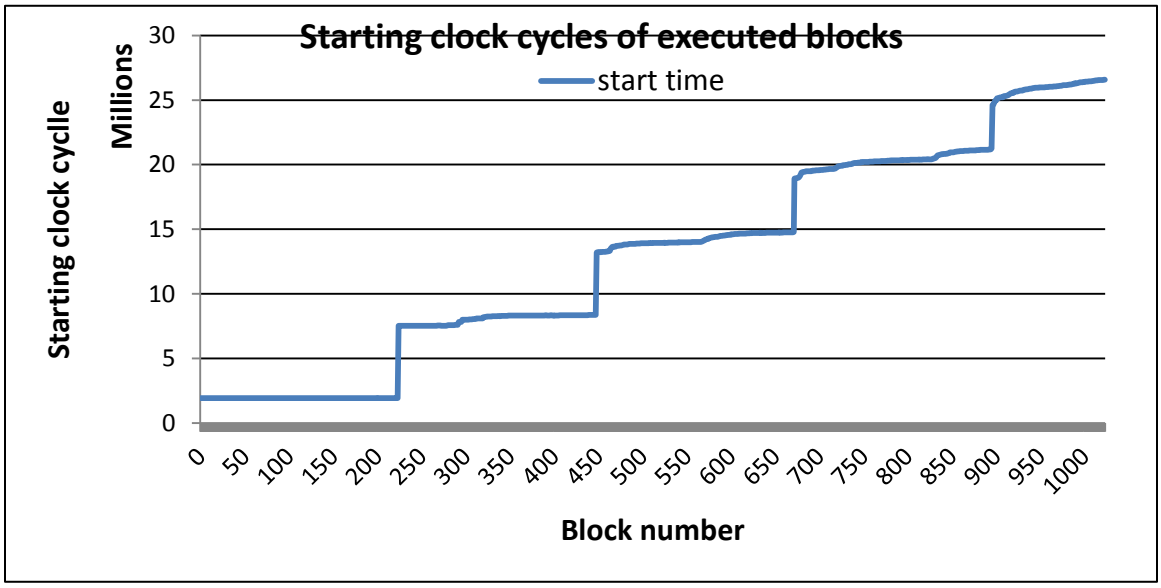

Figure 26: Values of execution start clock cycles against block IDs.

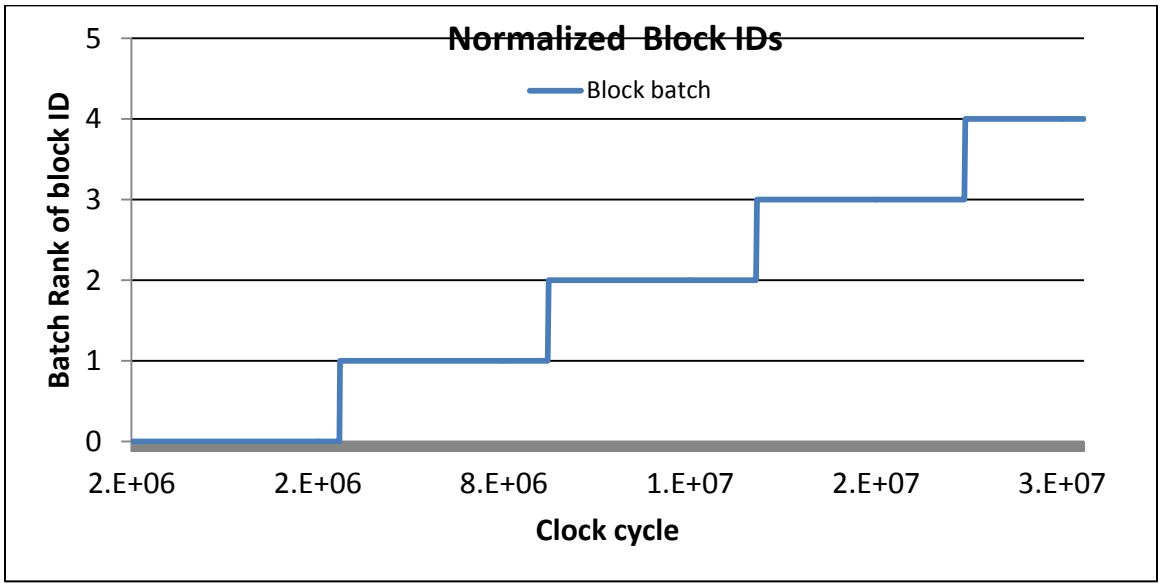

Figure 27: Block ID values normalized to multiple of K20 GPU maximal residency against their starting clock cycle values.

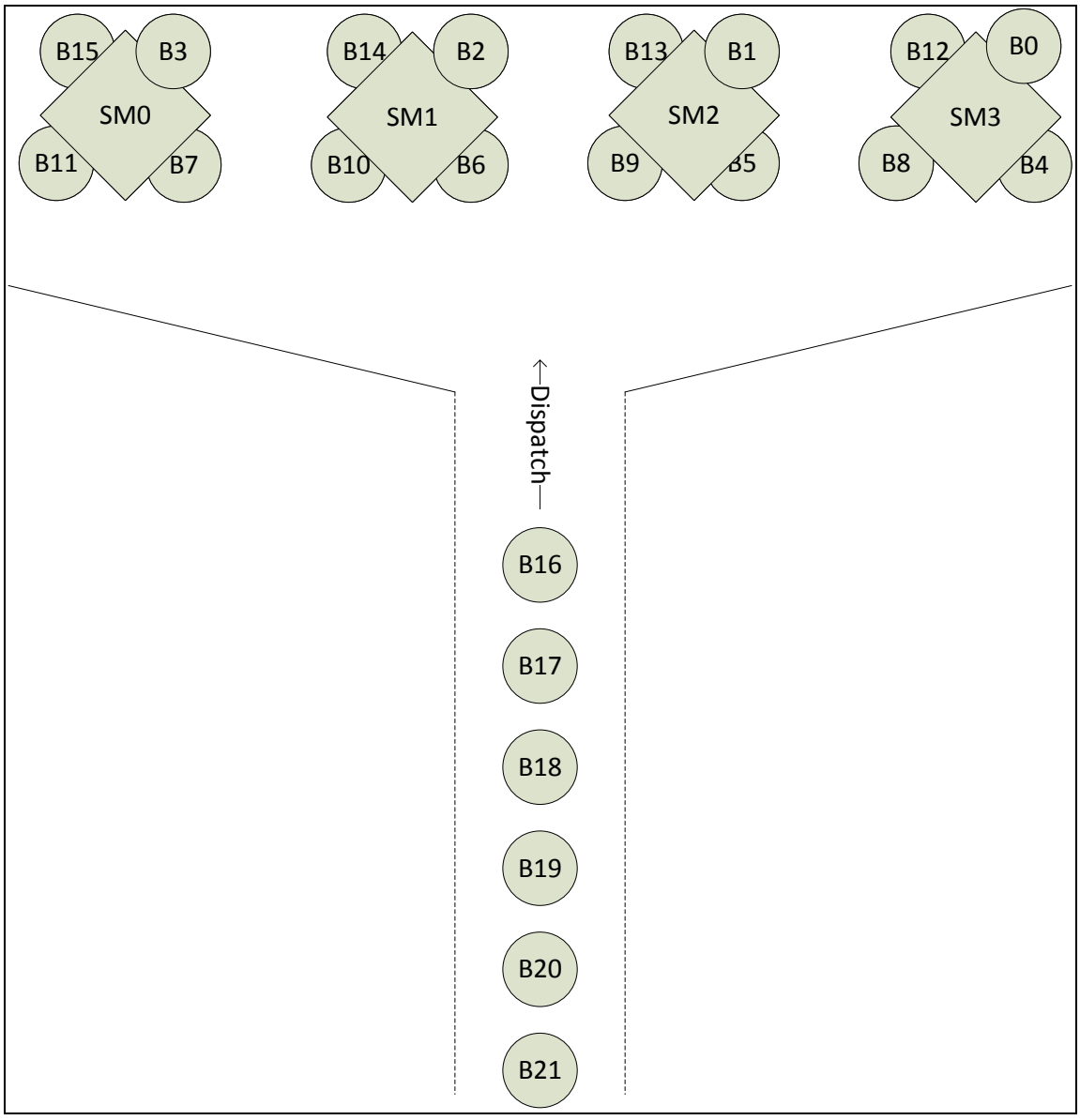

Figure 28: An illustration of block dispatch mechanism on GPU hardware

4.5 Practical deadlock detection and bypass for validation of resutls

The goal of this section is to validate the conclusions we reached about the rules of the maximal residency and the block scheduler's monotonically increasing block dispatching strategy. We designed an experiment where blocks trying to do a rendezvous synchronization using global synchronization technique discussed earlier; where we

measure the waiting time during synchronization. If the waiting time exceeds a predefined time-out, then the thread-block reports a timeout incident and releases itself from the synchronization point as if the synchronization incident has succeeded for that block. If the program finishes with reported timeout incidents, then we know that deadlocks had occurred. Of course, we expect time outs to occur if and only if the number of blocks participating in the synchronization operation exceeds the residency limit of the kernel in the conducted experiment.

For that experiment, we need to introduce a new version of the global synchronization code that was done by [78]. The new version applies the timeout mechanism for blocks that are participating. The code for the adapted global synchronization function is listed in Listing 19 below. Notice that this is the final production ready code for this task, where the block orchestrating the synchronization (always chosen as block number zero here) needs to spans the flags in case the number of blocks exceeds the number of threads in a single block. In line 13 each block clocks itself in. lines 41 – 50 are used to force each participating block to wait for a block-dedicated release flag to be set by block zero. Block zero will be the last block to reach these lines since it will be in lines 19 - 37 orchestrating the synchronization operation by checking arrivals for the rendezvous (in a monotonically increasing order, then raising individualized release flags that announce the end of the wait operation for the blocks waiting in lines 41 - 50. In line 43, each block checks the time spent since arrival to the rendezvous. If a block finds that time exceeds the time given as an input to the routine (in clock cycles); the block reports the incident by adding a counter in line 45 then sets its own release command in line 46 instead of waiting for it to be set by block zero. Notice that in a deadlock situation block zero will still be executing lines 24 - 26 waiting for all participating blocks to announce their arrival. This implies that block zero will keep waiting for arrivals for the rendezvous even when those arrivals haven't been dispatched into execution. When the last (R_L-1) blocks execute the synchronization operation (which only happens after all previous blocks finished execution), the synchronization operations would not timeout since the flags set by previously launched blocks are persistent and indicate the arrival of all previous synchronization operations to the rendezvous-type synchronization. This is because the

flags are integers set to a goal value that indicates the synchronization operation's number. Because of that, there will always be R_L blocks that never report any timeouts (the last R_L-1 blocks in addition to block number zero). Because of that, when the variation 0f number of blocks in different runs of the experiment changes from R_L to R_L-1 the number of timeouts registered by the kernel jumps from zero to R_L-1

For example, if an iterative application does 2000 iterations with a grid size of 1024 blocks and a residency limit of 224, then this is clearly will be a deadlocking situation with a non-zero number of timeouts timeouts when the routine in Listing 19 below is used for global synchronization.

Of course, when deadlocks are resolved the way they are here then the results will be wrong or unpredictable. However, the goal of this experiment is to show that the analysis about when to expect a deadlock in the previous sections is correct. We show the results of this experiment in the next section.

Listing 19: Global synchronization with with deadlock detect and bypass.

CUDA code for global synchronization routine with timeout

```
01  __device__ int number_timouts = 0;
02
03  __device__ void __gpu_sync_withTimeOut(unsigned int goalVal, int iter,
04                 volatile int *Arrayin, volatile int *Arrayout, long long
    clockOut) {
05        unsigned int tid_in_blk = threadIdx.x;
06        unsigned int nBlockNum = gridDim.x;
07        unsigned int bid = blockIdx.x;
08        unsigned int tid = bid * blockDim.x + tid_in_blk;
09        int numBatches = nBlockNum + 1 / blockDim.x;
10
11        long long tic, toc;
12        if (0 == tid_in_blk) {
13               tic = clock64();
14        }
15
16        if (tid_in_blk == 0) {
17               Arrayin[bid] = goalVal;
18        }
19        if (0 == blockIdx.x) {/*no more than one block can cover this task*/
20               int index;
21               for (int batch = 0; batch < numBatches; batch++) {
22                      index = tid + batch * blockDim.x;
23                      if (index < nBlockNum) {
24                             while (Arrayin[index] < goalVal) {
25
26                                    //Do nothing here
27                             }
28                      }
29               }
30               __syncthreads();
31               for (int batch = 0; batch < numBatches; batch++) {
32                       // release all blocks from synchronization point
33                      index = tid + batch * blockDim.x;
34                      if (index < nBlockNum) {
35                             Arrayout[index] = goalVal;
36                      }
37               }
38               __syncthreads();
39
40        }
41        if (tid_in_blk == 0) {
42               while (Arrayout[bid] < goalVal) {
43                      toc = clock64();
44                      if (toc - tic >= clockOut) {
45                             atomicAdd(&number_timeouts, 1);
46                             Arrayout[bid] = goalVal;
47                      }
48               }
49        }
50        __syncthreads();
51  }
```

4.5.1 Experimental setup and results

We tested the global synchronization with the deadlock detect and bypass method over the Jacobi iterative solver. The Jacobi kernel's residency limit is the same one calculated earlier in section 4.3.4 (page 95), which is 154 blocks on the on K20 GPU and 165 blocks on the K40 GPUs. We varied the number of blocks and iterations in the kernel and noted the number of timeouts. We varied the size between 1024 and 16384, number of blocks between 16 and 512; where the variation step was 1 block at a time around the residency limit value and larger elsewhere. Iterations between 2 and 10 for a total of 3700 experiments. None of the results mismatched expectancy of a deadlock. Meaning that all experiments with blocks exceeding the residency limit 1225 actually had registered a number of deadlock bypass incidents (non-zero number of deadlocks) and all of the 2275 runs that had a number of blocks that is not exceeding the residency limit had exactly zero deadlock bypass incidents (no deadlocks). Of these 3700 runs , 1850 runs was run on the K20 with a kernel residency limit of 154 while the other 1850 runs was run on the K40 with a kernel residency limit of 165.

In the next section, we will use the information and analysis of this section to introduce a new novel method to avoid the deadlock problem in iterative applications while still running all the steps of the iteration on a single kernel launch.

4.6 Using Dispatch Order as an alternative loop structure in GPUs to reduce synchronization overhead

In the previous sections, we highlighted the limitations imposed while accomplishing rendezvous/wall barriers using inter-block synchronization. The limit imposed is the number of blocks allowed to be launched to avoid deadlocks during synchronization. We have shown that in such cases, our only option to avoid deadlocks is to limit the number of blocks into our defined residency limit R_L or maximal residency R_M. Accomplishing wall barriers using inter-block synchronization might be useful where applications with

persistent blocks/threads are needed. In iterative applications that perform a significant amount of scalable work, however, this method would not be adequate, and synchronization with repetitive kernel launches would be more adequate.

In this section, we propose a new method for performing producer-consumer inter-block synchronization without the imposed limitation for the number of blocks to avoid deadlocks.

4.6.1 Exploiting order of block dispatch to orchestrate loops inside GPUs

It was discussed in earlier sections that blocks are issued to the GPU board's grid in an increasing order per block ID. We can conclude with certainty that if a block with index B is dispatched into a GPU multi-processor, all blocks of index B` < B would have already been dispatched into the grid. This means that **the system should be in a safe state if a block with block ID B performed a busy-wait to receive/consume signals/output from blocks with index B` < B**. This conclusion opens the path to our contribution discussed below. At this stage, it is also worthy to note from Table 5 above that the number of blocks in the x dimension that can be configured in the kernel launch configuration is extremely large for modern compute capabilities from NVidia (2^{31}-1).

Notice that wall barriers impose a restriction where all threads/processes shall come to a rendezvous. The participants need to exchange data to signal arrival to the rendezvous. Iterative algorithms that are implemented using wall barriers are implemented that way because of the existence of true flow dependence between successive iterations. That means the output information from iterations is needed as input for next iterations.

In our example of Jacobi iterative solver, each iteration I produces a solution vector X_i that is needed by iteration I+1 for it to calculate another solution vector X_{i+1} that should be closer to the real solution of the system than solution vector X_i. When several processes or threads are involved in that calculation; a wall barrier need to be introduced to cover inevitable cases when some working processes/threads finish before others and

start calculations related to the next iteration, of which the input vector is only partially produced at that time. When the work needed by iterations is decomposed over blocks in the kernel launch configuration, all blocks need to make a rendezvous barrier before going to the next iteration. This rendezvous is traditionally held at the CPU by designing a GPU kernel to accomplish a single iteration, while repetitively launching the kernel as much as needed with new parameters. These barriers are automatically performed when calling kernels repeatedly in the default stream. Another method highlighted earlier is to hold that rendezvous inside the GPU, where there would be a residency limit R_L imposed over the number of blocks.

The purpose of synchronization in parallel systems is usually either to enforce ordering or to enforce mutual exclusion. In cases of parallelizing a loop with true data flow dependence, a consumer must ensure that data it needs to process are already stored in memory. That insurance can be accomplished by a **post-wait** synchronization pair, where a producer of data sets a flag variable in memory to indicate the finish of a job, where a consumer would wait for the flag to be set before proceeding

We are proposing a method where iteration space is **projected** into extra blocks in the x dimension. Ordering can be enforced using post-wait synchronization mechanism, where signaling and waiting threads/blocks need not co-exist together in a rendezvous wall-barrier manner. This idea would work because producers of the source of the flow can finish and vanish since they are not assigned the task of performing work on next iterations (which if they are, they would be the consumers of their own output). Rather other blocks launched concurrently or at a later stage would be assigned the task, where the scheduler would dispatch them into execution when hardware limits allow. The consumer blocks need only to successfully gather completion signals from relevant producers to proceed with their tasks. Figure 29 shows an illustration of such a projection mechanism.

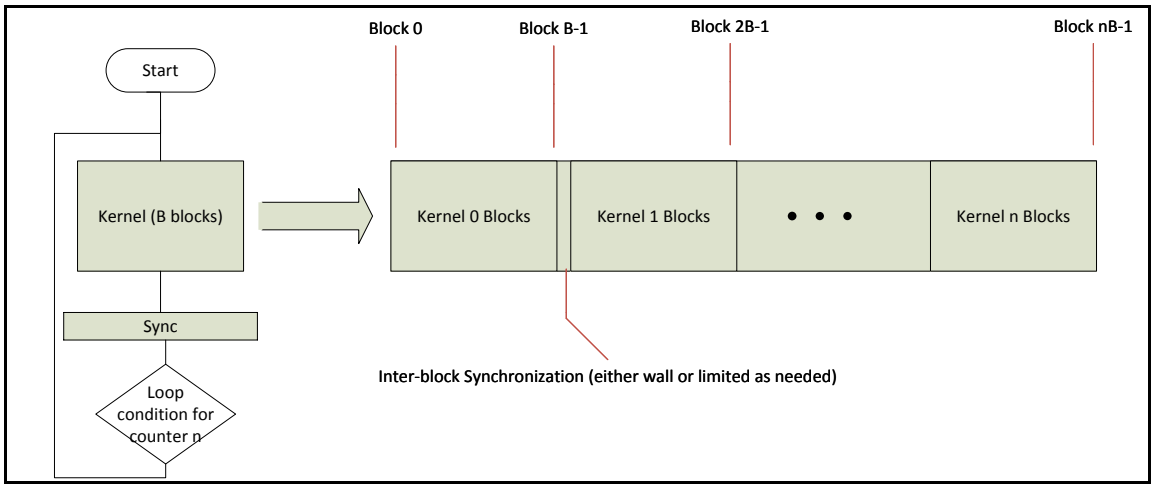

Figure 29: An example kernel call structure projection for the new methodology

The words source and sink are usually used when studying data flow behavior in algorithms to describe the roles played by producers and consumers respectively. Notice that, due to ordered dispatch behavior, a block residing on GPU usually would only coincide in the grid with block indexes with distance $\mathbf{R_L}$**-1**. However, if the source block index is less than the sink block index, the system would be in a safe state, and there would be no limitation imposed on the number of blocks assigned to iteration cycles in the kernel launch configuration. An inter-block synchronization might be needed to ensure serialization of iterations. To understand why we would need inter-block synchronization for that purpose, Figure 30 shows an illustration of block dispatcher mechanism in NVidia GPUs. Notice that the dispatching mechanism is a monotonically increasing one, while blocks do not necessarily finish in the same order they started execution with, i.e., it is **not** a FIFO execution mechanism. If we consider block IDs to be the order of launch in GPU, we can say that a NVidia GPU has a FIFO dispatch queue but an irregular-serving execution pattern. While the dispatcher slides execution window progressively; blocks of two GPU kernels could co-exist in the execution window together. This concurrency might not be the intended behavior of the developer and is not the intended behavior in serial iterations with true data-dependence. Thus, applying projection necessitates an inter-block synchronization operation to maintain execution ordering. This synchronization operation, however, does not need to be a rendezvous

style one. Only a flag need to be set by the producer and consumed concurrently or at a later stage by consumer blocks.

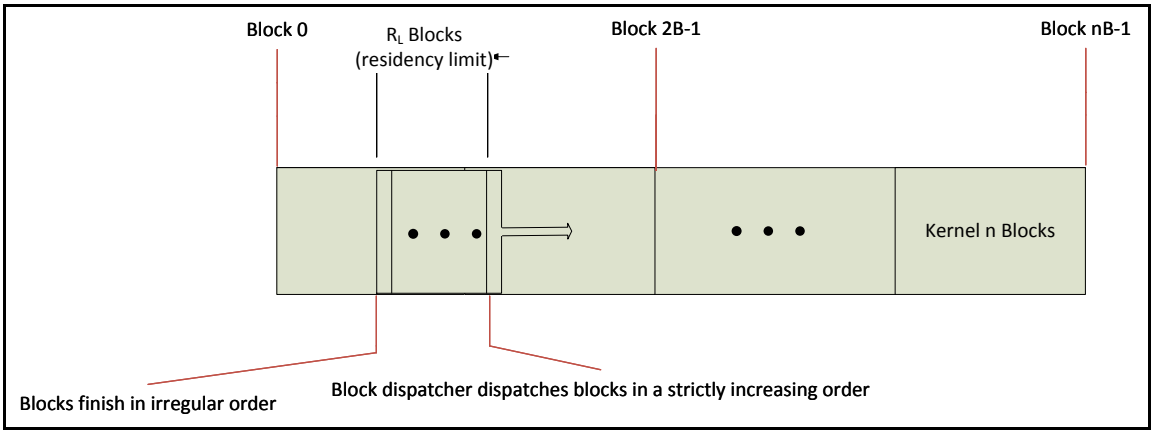

Figure 30: Block dispatcher mechanism (First-In).

Other than our approach, there are two prior approaches available to enforce the seriality between iterations of an application. The first is kernel exit-reentry; where the CPU orchestrates the iterative loop. The second is wall-barrier using inter-block synchronization [78].

4.6.2 Execution model

The execution model of our approach depends on implicit serialization of block execution. This serialization is forced against different chunks of blocks by hardware limitations. The ordering of block dispatch is noticed to be done by the block dispatcher. However, as discussed earlier, there is a window of concurrency among chunks of blocks demonstrated in Figure 30 above. The size of the block-concurrency window is dependent on the combination of block resource usage and GPU hardware model or generation. Enforcement of true flow dependence between blocks inside the concurrency window should be addressed by a synchronization model suitable for the application. The chosen model in a shared memory system such as the GPU is usually an asynchronous producer-consumer synchronization model. Synchronization could also be enforced by

using persistent blocks along with the global synchronization discussed earlier. However, the more decoupled the synchronization method, the better the performance should be, and the flexibility of the resulting application would be.

In summary, there are three models available for enforcing the seriality of iterative methods in GPU programs. First, there is the traditional officially endorsed method of repetitive kernel calling where the loop is orchestrated inside the CPU. The second method is to transform the iterative loop into the kernel while synchronizing the iterative work using a wall barrier global synchronization. Our method, the third, is to map iteration cycles into blocks in increasing order. Strict serialization of inter-iteration execution could be enforced using post-wait synchronization.

Figure 32 - Figure 34 illustrate execution models for the three approaches of orchestrating a serial iteration over GPU. In the case of the iteration space projection, we suggest a producer-consumer synchronization model; where flag variables are used to enforce the respect of flow dependence between blocks of different iterations. This approach would be both scalable and allow more flexibility in program restructuring. Also, our method allows programs to make use of second level caches between iterations.

4.6.3 Performance factors in different approaches to loop orchestration

Notice that global inter-block synchronization the way it is used in [78] uses persistent blocks, where a block stays "alive" from kernel launch until kernel end. If this was applied to a software model where a block works on consecutive data, this model might have certain advantages regarding data locality in comparison with an approach that divides the same data set between different blocks. However, these advantages will not work for large data sizes where each pass of a thread is working on a large set of data.

As an example, we discuss smith-waterman string matching algorithm that is usually used in DNA sequencing. The cost analysis phase has a true iteration dependency between each element and the three elements to the north, northwest and the west of each

element. The true flow dependence forces parallelized versions of the algorithm to do a sweep over the anti-diagonals in the direction of the main diagonal. For demonstration purposes, we assume the simplest data decomposition for such algorithm, which is to divide each anti-diagonal among all available threads in the iteration. This type of data decomposition leaves no room for data locality for threads involved. Parallelization of the application needs concurrent calculation of elements in the anti-diagonals, with a serial sweep in the direction of the diagonal [83]. We illustrate in Figure 31 below that the straightforward option is to distribute each anti-diagonal among different threads, resulting in loss of data locality of reference between cooperative groups of threads (warps) and losing the opportunity of data coalescing. Of course, that happens when using arrays with rows larger than the size of cache lines available to blocks.

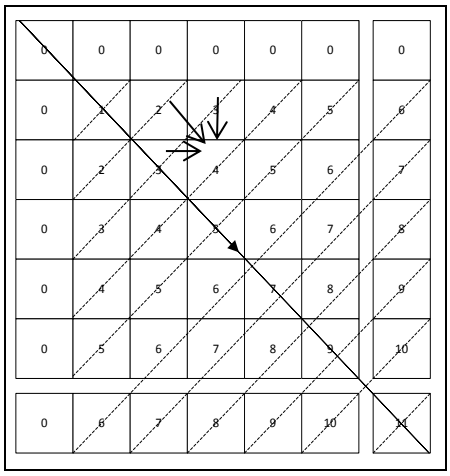

Figure 31: Smith Waterman data dependency and wave propagation

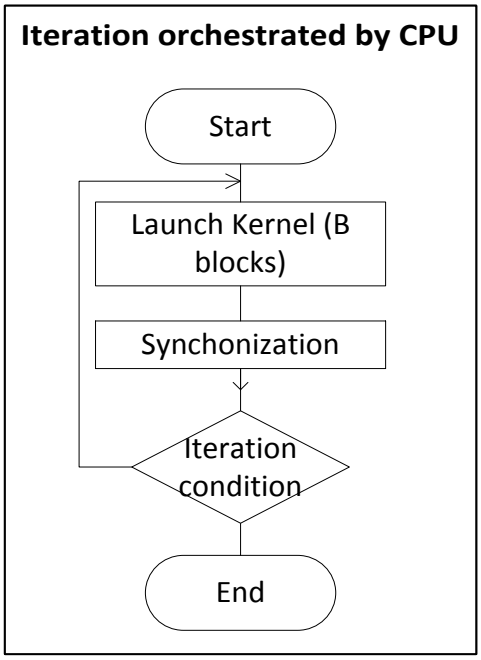

Figure 32: Diagram of iteration ordering using Kernel exit-entry (CPU-orchestration)

The cache argument mentioned above applies to all the approaches used. However, there might be a misconception at first sight regarding data temporal locality among different iterations. At first sight, the reader could conclude that different approaches have different effects for temporal data locality when it comes to memory accesses. However, the reader should be reminded that for the first level of cache, the most important aspect of data locality is the locality of reference among cooperative thread arrays (threads or WARPS), not temporal locality. The reason behind this is the time-sharing of each multiprocessor among different warps of the same block and several blocks when the resources and residency permits. However, modern GPU architectures also have a second level of cache known as L2 cache, as shown in Figure 35 below. The L2 level caches memory accesses originating from all multiprocessors in the GPU. For example, NVidia Kepler GK110 Chip has 1536 KB of L2 cache. Such cache size could contain the whole array of the smith-waterman cost array shown in Figure 31 above if it was a square array of size up to 627x627.

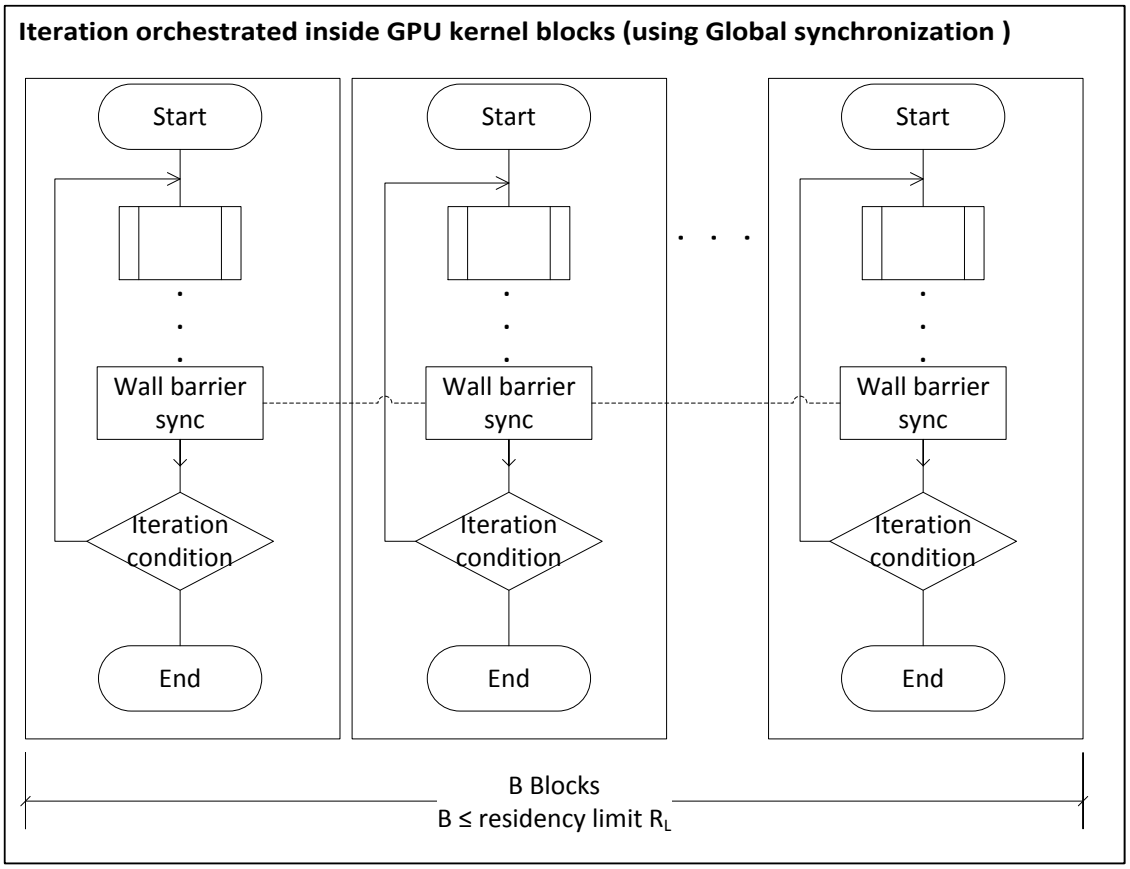

Figure 33: Diagram of iteration ordering using global synchronization

Discussing GPU memory hierarchy is relevant in our context to compare the traditional CPU-orchestrated loop seriality with the other approaches of GPU global inter-block synchronization and our iteration space projection technique. In the CPU-orchestrated loop seriality, the synchronization depends on exit – reentry from a kernel which causes all caches to be flushed to ensure memory coherency. While L2 cache locality would still be relevant in inter-iteration execution in other approaches. This makes temporal and referential locality more relevant across loop iterations in cases where the data layout and decomposition of the algorithm allows it. This concept will be clearer when test case experiments are discussed.

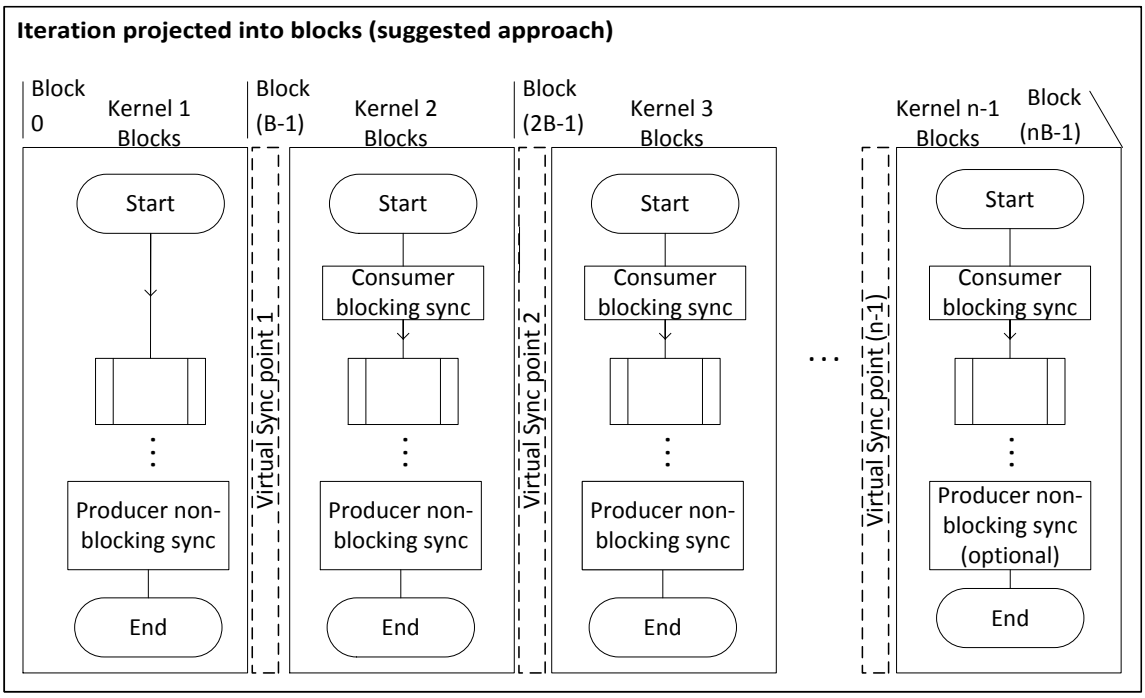

Figure 34: Diagram of newly proposed Iteration pattern

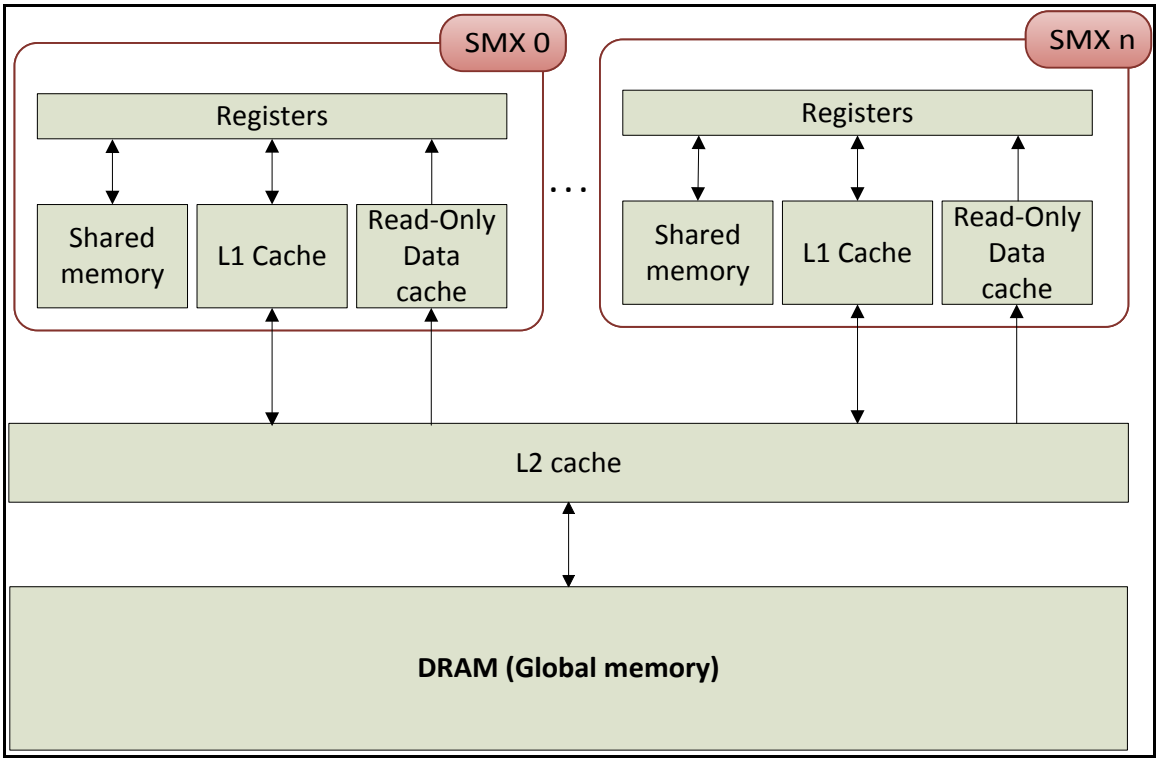

Figure 35: Memory hierarchy in modern GPUs

4.6.4 Producer- Consumer synchronization

To accomplish producer-consumer synchronization, we need blocks assigned to an iteration cycle to wait for signals of completion from all blocks assigned to the immediately previous iteration. There could be many ways to handle such behavior; we illustrate our chosen method in Listing 20.

Listing 20: Algorithm for one-way producer-consumer synchronization

```
//Producer code:
01  // … (do work)
02    if (threadIdx.x ==0)
03  // set flag announcing the finish of the current block work
// consumer code:
04  // if this block is not responsible for first iteration
05  if (current_iteration > 1) {
06    int flag
07    //wait_here:
08    Do {
09    // keep reading flags from all blocks of producer work
10    } while (! aggrigate_flag_set)
11  } // end if
```

4.6.5 Projection patterns and synchronization verbosity

One can notice the projection proposed expects explicitly knowing the number of iterations in advance; our projection methodology is only applicable to applications of the form displayed in Listing 21.

Listing 21: Form of iterations that would be projectable into block-space

```
01  for (/*loop conditions*/) {
02      Kernel_function<<<grid_size, block_size>>> ();
03      cudaThreadSynchronize (); // optional explicit synchronization
04  }
```

The execution timing for this form of GPU applications can be expressed by the following formula displayed originally in [78]:

$$\sum_{i=1}^{M}\left(t_O^{(i)} + t_C^{(i)} + t_S^{(i)}\right)$$

Where M is the number of kernel launches, $t_o^{(i)}$ is the kernel launch time, $t_c^{(i)}$ is the computation time, and $t_s^{(i)}$ is the synchronization time; all are for the i^{th} kernel launch.

When projecting such applications, the time enhancement expected is limited by two factors, namely the total sum of t_o and t_s. The projection is expected to eliminate t_o. However, reduction of t_s is subject to communication pattern and its projection.

The reader needs to notice, however, that grid size in GPUs currently has to be determined in advance. Hence loops that are completely projectable in this pattern are only loops with an explicit number of iterations. Some loops have an unknown number of execution cycles and the loop condition needs to be checked after each kernel launch. These loops shall be projected in steps (e.g. projecting two iterations at a time); this way the loop and synchronization overhead are reduced to 50% in cases applicable or in cases that the possibility of an extra iteration has no negative effects. The code inside the kernel can still be made to ensure that no extra iterations are performed if needed. Figure 36 below shows the diagram of such strategy. Notice that 2B blocks need to be launched at a time after the transformation.

After the analysis shown in section 4.4, readers should know by now that there is a firm limit of Block distance (R_L) that after which, block number ordering is guaranteed. To show an example, we highlight a demo kernel with applicable grid block residency (residency-limit) R_L of 6. The example is illustrated in Figure 37. In that example, if we assume that in our demo kernel implementation pattern, blocks have a true flow dependence of block distance F_d= 7. In that case, since F_d is greater than R_L; the consumer already guarantees that its producer (i) have already produced necessary data for the consumer (i). We call this **implicit synchronization**. Designing iterations of GPU kernels in a way that projects true dependence into a block distance greater than the kernel's R_L is the desired implementation decision since it eliminates the synchronization overhead.

We will highlight an example of an application that can be naturally projected in some cases to benefit from implicit synchronization behavior when we show the matrix exponentiation example.

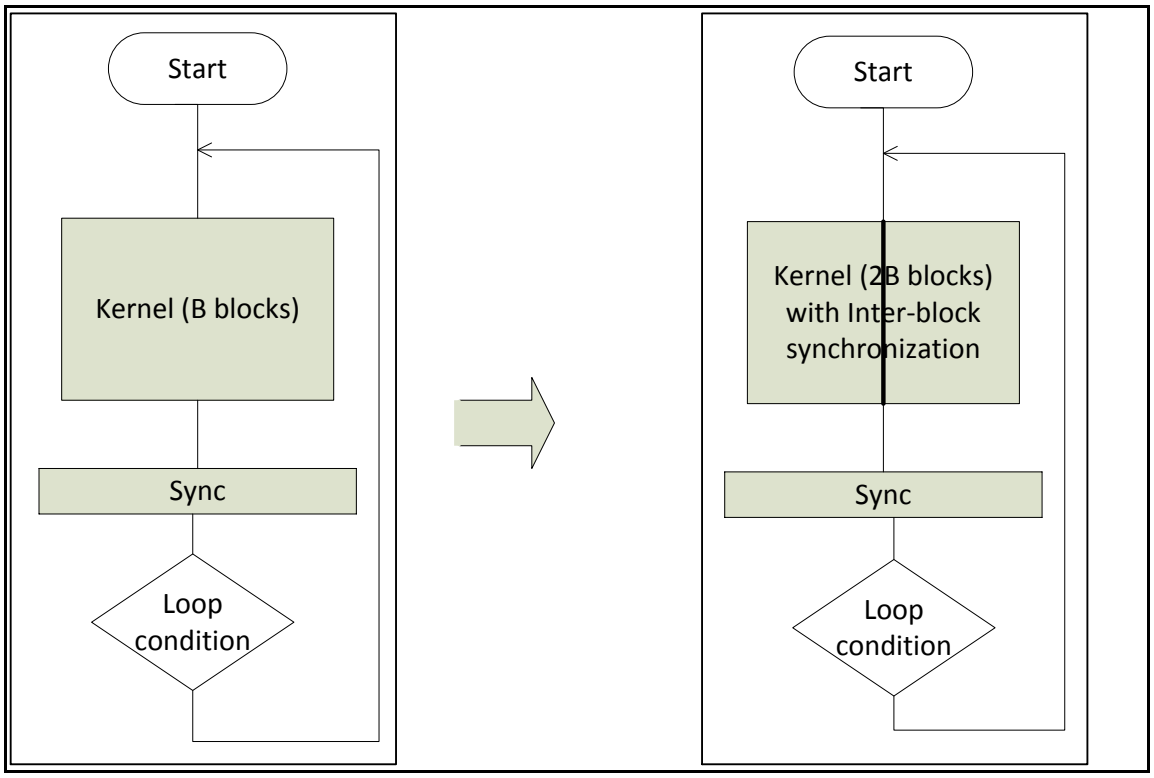

Figure 36: Diagram of projection pattern for loop conditions with undefined number of execution times

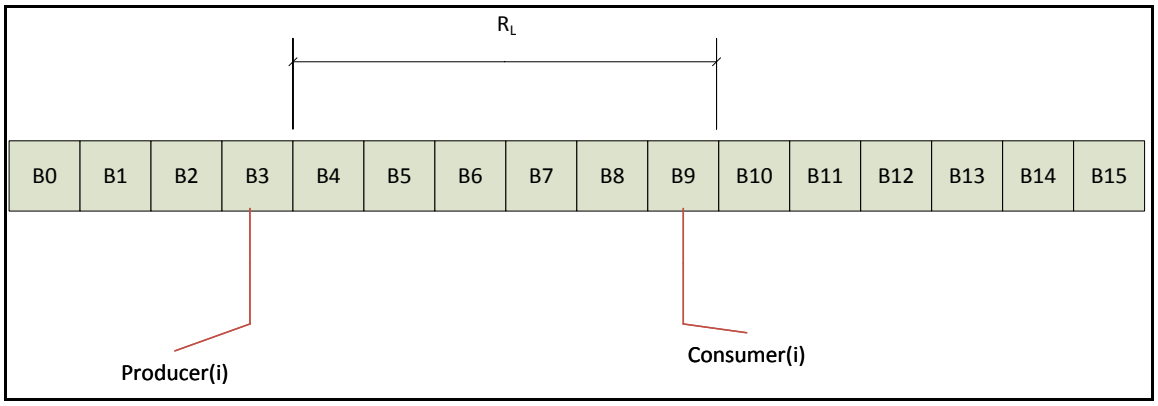

Figure 37: Producer - consumer relationship with no need for explicit synchronization

4.6.6 Experimental work and results

4.6.6.1 Introduction

Experiments in this section were conducted on a NVidia K40 GPU card. However, visualizations produced by the profiler such as processor utilization graphs and SM occupancy by blocks and warps were produced by **NVidia profiler** running on a machine with a NVidia K20 GPU card. The difference between the two GPUs is that K40 has an extra SM (15 SMs) and a slightly faster clock rate. This similarity means that Occupancy data produced on a K20 card should be the same for the K40. However, the relationships between the number of processors and certain features of the results in graphs should be different between the two cards in their positions when applicable. When a feature of a result is related to the number of SMs in GPU, we mention this in the relevant places.

In the example of Jacobi iterative solver, suppose that for a matrix of 8192, we want each thread to perform a single whole row operation for blocks of size 32. In such case, the calculation needs 8192/32 = 256 blocks, which is > R_M for compute 3.5 capability GPUs. If we expect to run the application for 100 iterations, we will launch a kernel of 256 × 100 = 25600 blocks in order to successfully perform the X vector wave propagation among blocks

For the Jacobi implementation, we converted a base CUDA one into projected. A pseudo version of the Jacobi code is shown in Listing 22. Listing 1 shows a CPU based serial Jacobi pseudo code. Notice, that instead of constructing 2 for loops around the diagonal, we can subtract the diagonal element's product from the total sum before or after the dot product operation and making a full dot product for each row using a single loop. It should be noted that.

Lines 02-10 are responsible for solving for X_i in every iteration cycle, while lines 11-13 are updating the solution vector. Lines 01 and 14 orchestrate the main serial iteration.

Listing 22: Sequential pseudo code for Jacobi iterative solver[84, Sec. 2.1.3]

```
Size: size of the matrix
X[Size]: solution vector
XOld[Size] Solution vector in previous iteration
A[Size][Size]” coeffecient matrix
B[Size]: right-hand side vector

01  Repeat
02        for i=0 to size-1 do
03              X[i] ← 0
04              for j=0 to i-1 do
05                    X[i] ← X[i]+A[i][j] × XOld[j]
06              end for
07              for j=i+1 to size-1 do
08                    X[i] ← X[i]+A[i][j] × XOld[j]
09              end for
10        end for
11        for i=0 to size-1 do
12              XOld[i] ← (B[i] – X[i]) / A[i][i]
13        end for
14  until a stopping criteria is reached
```

As the iteration advances in the Jacobi, solution vector X should get closer to the actual solution if the system meets the precondition. Currently, in our work, we are not considering any stopping criteria for the solver, and are testing our work on a predefined number of iteration cycles. Listing 2 shows the c code corresponding to the pseudocode above.

Listing 23: A corresponding code in C for the serial pseudo code shown in Listing 22

```
01  float sum=0;
02  for (k = 0; k < MAX_ITER; k++) {
03        for (int i=0; i<sz; i++) {
04              sum = -A[i+i*sz] * Xs[i];
05              for (int j=0; j<sz; j++)     {
06                    sum += A[i*sz+j] * Xs[j];
07              }
08              XNs[i] = (B[i] - sum)/A[i*sz+i];
09        }
10        for (int i=0; i < sz; i++) {
11              Xs[i] = XNs[i];
12        }
13  }
```

To parallelize Jacobi solver on a parallel shared memory machine, multiple dot products inside each step in the main iteration can be done in parallel. The main iteration,

however, needs to be serialized. i.e., Each step in the main iteration need to be finished before the next is started. A parallel pseudo code is shown in Listing 24.

Listing 24: A parallel pseudo code for the Jacobi iterative solvers

```
Size: size of the matrix
X[Size]: solution vector
XOld[Size] Solution vector in the previous iteration
A[Size][Size]" coefficient matrix
B[Size]: right-hand side vector

01  Repeat
02        for i=0 to size-1 do in parallel
03              X[i] ← 0
04              for j=0 to i-1 do in parallel
05                    X[i] ← X[i]+A[i][j] × XOld[j]
06              end for
07              for j=i+1 to size-1 do in parallel
08                    X[i] ← X[i]+A[i][j] × XOld[j]
09              end for
10        end for
11        for i=0 to size-1 do in parallels
12              XOld[i] ← (B[i] – X[i]) / A[i][i]
13        end for
14  until a stopping criteria is reached
```

We notice that even before the main iteration ends, we have a synchronization point inside it at line 10, just at the end of the parallel for loop of line 2. Otherwise, the value of Vector XOld would be altered in line 12 before all the workers consumed it. Listing 25 shows an optimized version of the pseudocode in Listing 24. These optimizations are assuming an output-based decomposition for the gangs in the parallel version. However, these same optimizations are applied for the serial code too. Before parallelizing the code, we applied some optimizations on the code in Listing 24 above which we list below. The reader can spot these optimizations in the parallel version of the pseudocode in Listing 25 below:

1. Fused the two loops at lines 4 and 7. These loops are split to exclude the diagonal element from the dot product, so we subtract that and use one whole loop without an exclusion conditional statement.
2. Fused the loops at lines 2 and 11. To be able to do that, we changed the holding result to be x[i] instead of XOld[i] in the loop of line 11. This change will create

the need to have a copy operation back into XOld from the resulting X for the iteration to be correct. Instead of that, however, we do the next optimization.

3. Change the copy operation from X to XOld into another step in the iteration that reverses the input/output variable roles. Hence X will be the old X and XOld will be the new x in a new loop identical to the fused loop of line 2.

For parallelizing the pseudo code, we adapt the parallel version to the GPGPU parallel programming model. Hence, we parallelize some loops across gangs and others across workers, i.e. some across blocks of threads and others across the threads themselves.

The resulting parallel code is listed in Listing 25. Notice that optimizations 2 and 3 above forces the algorithm only to be able to check for stopping criteria after two steps of the iteration instead of every one step. This change should have a negligible impact on the result. Moreover, we are not checking for stopping criteria in our experiments.

Listing 25: An optimized pseudo code version of a Jacobi solver for the GPU

```
Size: size of the matrix
X[Size]: solution vector
XOld[Size] Solution vector in the previous iteration
A[Size][Size]” coefficient matrix
B[Size]: right-hand side vector

01  Repeat
02   for i=0 to size-1 do in parallel across gangs
03       X[i] ← - (A[i][i] × XOld[i])
04       for j=0 to size-1 do in parallel across workers
05           x[i] ← X[i]+A[i][j] × XOld[j]
06       end for
07       X[i] ← (B[i] – X[i]) / A[i][i]
08   end for
09   for i=0 to size-1 do in parallel across gangs
10       XOld[i] ← - (A[i][i] × X[i])
11       for j=0 to size-1 do in parallel across workers
12           XOld[i] ← XOld[i]+A[i][j] × X[j]
13       end for
14       XOld[i] ← (B[i] – XOld[i]) / A[i][i]
15   end for
16  until a stopping criteria is reached
```

For the projective synchronization approach over the Jacobi solver, it was already discussed in the previous section why there is a need for synchronization point between

blocks assigned to consecutive elements. A projective pseudo code version is shown in Listing 26. Notice the *post* synchronization step in line 21 and the wait operation is placed naturally at the beginning at lines 02 and 03. We deliberately put a generalized wording without the specifics because the methodology that could be used for the performance is subjective to the application and choice of developer. Since multiple blocks are assigned the work of each iteration cycle; a post operation could be a complex operation that aggregates all flags of blocks assigned to each iteration into a single flag posted by a one of the participating block. This could be done, for example, to reduce the breadth of polling operations over memory done by the waiting blocks. We will discuss in later sections the specifics of such cases. However, before that, we need to shed more light on the implementation and data decomposition of the Jacobi GPU parallel experiment.

Listing 26: Pseudo code of the dispatch-order orchestrated Jacobi

```
    Size: size of the matrix
    N_iteration: number of iterations
    N_gang: number of gangs
    N_blocks: number of blocks dedicated for each iteration step
    X[Size]: solution vector
    XOld[Size] Solution vector in the previous iteration
    A[Size][Size]” coefficient matrix
    B[Size]: right-hand side vector

    01  calculate current_iteration based on gang_id
    02  if current_iteration is not the first
    03     do a wait synchronization operation for previous iteration
    04  if current_iteration is an odd_number
    05     for i=0 to N_blocks do in parallel across gangs
    06         X[i] ← - (A[i][i] × XOld[i])
    07         for j=0 to size-1 do in parallel across workers
    08           x[i] ← X[i]+A[i][j] × XOld[j]
    09         end for
    10         X[i] ← (B[i] – X[i]) / A[i][i]
    11     end for
    12  else
    13     for i=0 to size-1 do in parallel across gangs
    14         XOld[i] ← - (A[i][i] × X[i])
    15         for j=0 to size-1 do in parallel across workers
    16           XOld[i] ← XOld[i]+A[i][j] × X[j]
    17         end for
    18         XOld[i] ← (B[i] – XOld[i]) / A[i][i]
    19     end for
    20  end if
    21  do a post synchronization operation
```

4.6.6.2 Data decomposition and implementation:

The reader may have noticed that Jacobi solver revolves around matrix-vector multiplication. We need to make a parallel data decomposition that is suitable for a data-parallel machine like the GPU. Natural data decomposition, in that case, is an output-based parallel decomposition, where each thread is responsible for one element of the output vector X. This type of decomposition is naturally the fastest since it avoids inter-thread synchronization if there is enough parallelism available to utilize the GPU that is a sufficiently large size problem. Figure 38 shows an illustration of the output data decomposition used. It would be noticed that the chosen decomposition does not necessitate inter-thread synchronization operations in order to calculate each element of Solution Vector X. of course, the program still needs a synchronization operation after every iteration cycle in order to get a consistent previous solution vector for every iteration by all threads.

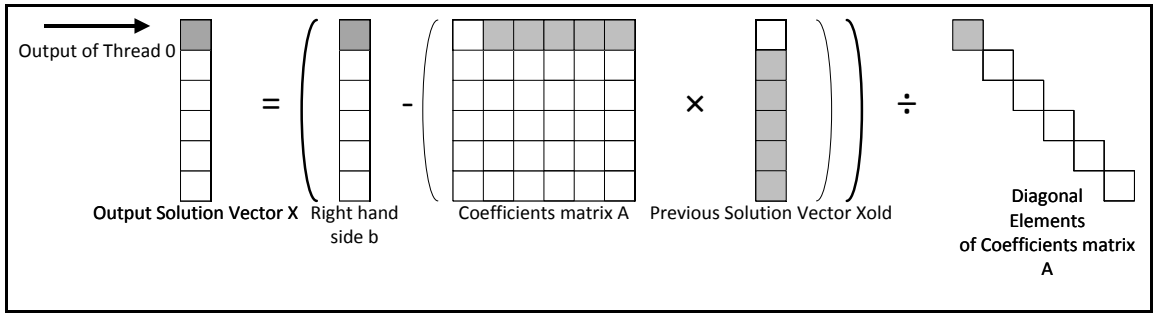

Figure 38: Output-based parallel data decomposition for a Jacobi linear solver; the data access pattern for the first thread is shaded as an example.

Listing 27 shows a simplified CUDA code that works per the output based row-wise parallel data decomposition discussed above. Notice that this code does not show memory architecture specific optimizations yet, such as memory tiling and coalescing.

Listing 27: CUDA code for Jacobi linear solver's Kernel

```
XN[Size]: solution vector
X [Size] Solution vector in previous iteration
A[Size][Size] coefficient matrix
b[Size]: right-hand side vector
wA : width of coefficient matrix A
gridDim.x : number of blocks
ix : absolute thread index
bx  : block ID (i.e. gang ID)

01  a = A[(ix + i) * wA + ix];
02  Csub = -a * X[ix + i];
03  m_row = (bx * BLOCK_SIZE * elements_per_thread + tx) * wA;
04  for (element = 0; element < wA; ++element) {
05    Csub += A[element + m_row] * X[element];
06    NX [ix + i] = (b[ix + i] - Csub) / a;
07  }
```

For the chosen data decomposition, each block of threads will work on consecutive rows concurrently. For example, each warp will simultaneously ask for a warp-sized column of elements. Since CUDA-C uses row-major storage format, that means memory access are not being coalesced among consecutive threads. To make memory accesses coalesced between threads. The usual workaround for coalescing memory accesses in Jacobi [85] is to read multiple consecutive elements in from each row cooperatively by cooperative threads. Figure 39 shows an illustration of a tiling operation done by four cooperative threads. Of course, tiling operation needs to be extended to cover entire rows using repetition operations, because neither shared memory nor the number of threads in a block would usually suffice for a single shot operation of entire rows. Listing 28 shows the Jacobi code with tiling.

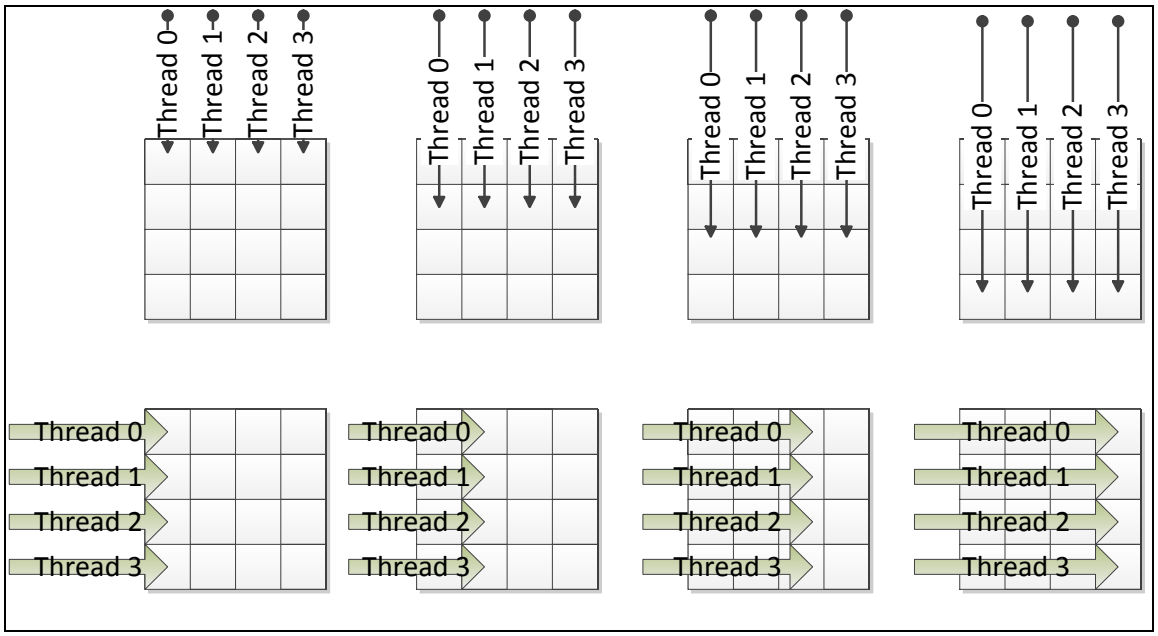

Figure 39: Coalescing memory access within warps while still preserving row-wise output based parallel data decomposition

Listing 28: Tiled CUDA code for the Jacobi linear solver[8]

```
    XN[Size]: solution vector
    X [Size] Solution vector in the previous iteration
    A[Size][Size]" coefficient matrix
    b[Size]: right-hand side vector
    WA: width of coefficient matrix A
    Bx: block ID
    gridDim.X: number of blocks

    01  a = A[(ix+i) *wA + ix + i];
    02  Csub = -a * X [ix + i]; // exclude diagonal element using subtraction
    03  for (tile=0; tile < End_tile; ++m_tile) {
    04    for (tile_row = 0; tile_row < BLOCK_SIZE; ++tile_row)
    05      DataTile [tile_row] [tx] = A[(bx*BLOCK_SIZE*+tile_row+i) *wA +
            m_tile*BLOCK_SIZE+tx];
    06    Xs[tx] = X[m_tile*BLOCK_SIZE+tx];
    07    __syncthreads ();
    08    for (tile_column = 0; tile_column < BLOCK_SIZE; ++ tile_column)
    09      Csub += DataTile [tx][ tile_column] * Xs[tile_column];
    10    __syncthreads ();
    11  }//03
    12  XN[ix] = (b[ix] - Csub)/a;
```

[8] This CUDA code for the Jacobi was adapted from code written by Ahmed Abu Nasser- MSc, Computer Engineering – KFUPM, for the parallel computing course.

After listing the core algorithm and code for Jacobi, the next section will discuss a variety of post wait synchronization techniques to reach the best performance.

4.6.6.3 Post-wait inter-block synchronization in GPUs

The performance of the synchronization step is the most critical point to make the projected approach cost effective. Several methods of inter-block synchronization were discussed in the literature, most of which depends on [78]. In [78], authors proposed methods for inter-block synchronization concluding that their lock-free method is the best performing one provided it is used without *threadfence* operation. A *threadfence* operation is necessary for guaranteeing the correctness of execution of programs of producer-consumer nature. However, the authors performed many tests in a specific hardware environment and concluded they did not need a *threadfence* on the specific hardware they tested on. However, they asked this assumption to be reevaluated no other platforms[86]. Lock-free synchronization was also concluded to be the best performer by [79]. However, their test did not span more than 128 blocks even for the CPU-orchestrated version, a number clause to the residency limit of the lock-free synchronization approach. Listing 29 shows the lock-free synchronization CUDA code as proposed and implemented in [78].Of course, this is a rendezvous-type synchronization that needs participants to spin-wait until the number of participants is complete. This synchronization is the type that would cause a deadlock in certain conditions that are discussed earlier in this dissertation.

Listing 29: Lock free inter block synchronization CUDA code(rendezvous-type synchronization) [78, Fig. 6]

```
    //GPU lock-free synchronization function
01  __device__ void __gpu_sync(int goalVal,
02                  volatile int *Arrayin, volatile int *Arrayout)
03  {
04          // thread ID in a block
05          int tid_in_blk = threadIdx.x * blockDim.y
06          + threadIdx.y;
07          int nBlockNum = gridDim.x * gridDim.y;
08          int bid = blockIdx.x * gridDim.y + blockIdx.y;
09
10          // only thread 0 is used for synchronization
11          if (tid_in_blk == 0) {
12                  Arrayin[bid] = goalVal;
13          }
14
15          if (bid == 1) {
16                  if (tid_in_blk < nBlockNum) {
17                          while (Arrayin[tid_in_blk]!= goalVal) {
18                                  //Do nothing here
19                          }
20                  }
21                  __syncthreads();
22
23                  if (tid_in_blk < nBlockNum) {
24                          Arrayout[tid_in_blk] = goalVal;
25                  }
26          }
27
28          if (tid_in_blk == 0) {
29                  while (Arrayout[bid] != goalVal) {
30                          //Do nothing here
31                  }
32          }
33          __syncthreads();
34  }
```

For the projected approach, Post-wait synchronization through the blocks can be accomplished with any of several methods. Our first go-to approach was a lock-free one. A lock-free approach is expected to utilize the parallelism in the target memory architecture in a way inspired by the lock-free synchronization in Listing 29.

Since the number of blocks assigned to every iteration cycle is constant among iterations, we allocate an array of integer flags. Each flag corresponds to a single block for an iteration based on the order of that block in that iteration. Listing 30 shows a pseudo code and a corresponding CUDA code for a lock-free waiting mechanism for synchronization.

Listing 30: Pseudo and corresponding CUDA code for a lock-free post-wait producer-consumer pairing synchronization

The Wait Operation Done by Consumers – Pseudo code

```
01  For I = 0 to number_of_blocks do in parallel across threads
02    Repeat
03    Until (finished_iteration[I] >= Previous_iteration)
04  End
```

The Post Operation Done by Producers – Pseudo code

```
01  If thread_ID_in_block = 0
02    finished_iteration[block_ID] = current_iteration
```

The Wait Operation Done by Consumers – CUDA code

```
01  __device__ inline void wait_for (int iteration, int finished_iteration [], int
    num_blocks) {
02    for (int i=0; (I <  (num_blocks)); i+=blockDim.x)
03      while (__syncthreads_and ( (finished_iteration[MIN(i+threadIdx.x,
        num_blocks)] < iteration  ) ) )
04        ;
05  }
```

Post Operation Done by Producers – Pseudo code

```
01   if (threadIdx.x ==0 ) // announce the finish
02       finished_iteration[blockIdx.x] = current_iter;
```

Another method for the post-wait method is a lock-based one. This method is feasible due to hardware/software support by GPU platforms for *atomic operations*. *Atomic operations* are operations that appear to the rest of the system components as if they happen instantaneously. They are indivisible i.e. either they succeed or have no apparent effect on the system. An implementation of a lock based post-wait synchronizations is shown in. One can notice in lines 01-04 that we single-out one thread to do the wait operation; because this reduces the pressure on global memory. Other threads would wait for a block-barrier operation since GPUs usually implement block/gang barriers. In the implementation of the lock in line 05, we chose to use an Atomic Add CUDA operation over using Atomic Increment operation even though *atomicInc* operation does the zeroing of the target operation automatically instead of manually doing it as in line 07. We are using *atomicAdd* over *atomicInc* because our tests show a better performance for *atomicAdd* in our application.

Listing 31: Pseudo and CUDA code listings of a lock based post-wait synchronization in GPUs

The Post Operation Done by Producers – Pseudo code

```
01  If thread_id_in_block = 0
02    Acquire exclusive access over variable g_blocks_posted
03      g_blocks_posted= g_blocks_posted+1
04      new_value = g_blocks_posted
05    release exclusive access over variable g_blocks_posted
06    if new_value = number_of_blocks
07      g_blocks_posted = 0
08      g_iteration = g_iteration+1
09    end
10  end
```

The Wait Operation Done by Consumers – Pseudo code

```
01  If thread_id_in_block = 0
02  Repeat
03  Until g_iteration >= iteration_to_start
04  Wait for thread 0 (block-barrier)
```

The Post Operation Done by Producers – Corresponding CUDA code

```
01  __device__ int volatile g_iteration = 0;
02  __device__ int  g_blocks_posted=0;
03  inline __device__ void __gpu_post_sync (const unsigned int num_blocks) {
04  if (0 == (threadIdx.x || threadIdx.y || threadIdx.z)) {
05  unsigned int blockOrder = atomicAdd(&g_blocks_posted, 1);
06  if (num_blocks - 1 == blockOrder) {// last block posting a flag
07  g_blocks_posted = 0;
08  g_iteration++; // maintain ordering of these two statements
09  }
10  }
11  __syncthreads();  /* between threads in block (not always necessary in
    producer) */
12  }
```

The Wait Operation Done by Consumers – Corresponding CUDA code

```
01  inline __device__ void __gpu_sync_wait(const int iteration_to_start) {
02  if ((0 == (threadIdx.x || threadIdx.y || threadIdx.z))) {
03  while (iteration_to_start > g_iteration) {
04  ;
05  }
06  }
07  __syncthreads();
08  }
```

4.6.6.4 Results

We measured our projected approach using both lock-free and lock-based synchronization approaches. We also run the experiments over global synchronization method to see how to compares to it. The first set of experiment is conducted over smaller data sizes that are less than 4096. The speedup of the Asynchronous Approach is listed as a guideline for the best that can be achieved without the post-wait synchronization method. For the small data sizes

For sizes larger than 64 (which needs to two blocks) neither of the approaches reaches nor exceeds the performance of the kernel exit-entry synchronization method. Of course, we are discounting the asynchronous experiment because it is not one of the approaches and just serves to set a foresight for the upper performance limit. In the asynchronous approach for data sizes that are small, we had to set a new experiment where we force a one-block per SM to be executed by both the CPU and Asynchronous approach when measuring the performance of Asynchronous approach. We do that to enforce correct behavior of the asynchronous approach by enforcing serialization among iterations during the phases at which the maximum degree of concurrency set for a single iteration is less than the hardware limit. Still, we cannot accurately measure speedup of asynchronous approach for data sizes that map to a number of blocks that is less than the number of SMs on the target machine; because otherwise, an immediate concurrency between different iterations would happen.

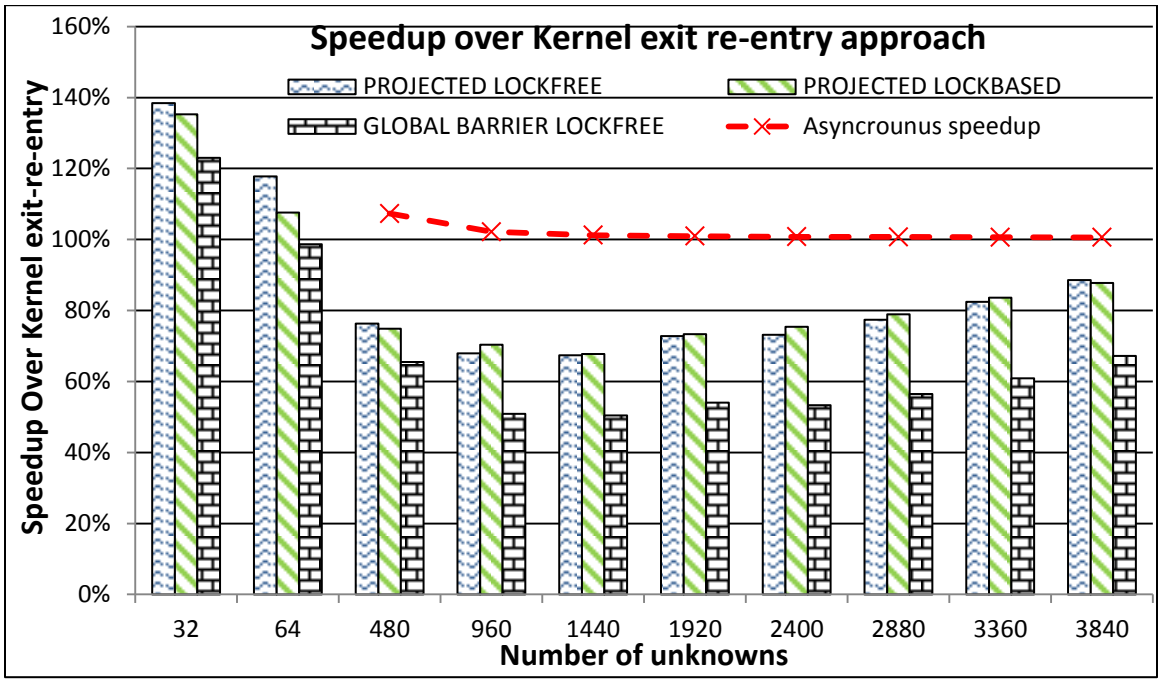

Figure 40: Speedup over Kernel exit-entry based Approach (for a set of relatively small data sizes)

The trend in speedup with respect to CPU based synchronization is expected to keep at that pace for larger data sizes as we will show below. This trend is caused by synchronization overhead. When looking at the tightly close speedup line of the asynchronous approach to the Kernel exit re-entry based one, the low speedup of the GPU orchestrated approaches becomes naturally expected because of the expected overhead. The tightly close speedup line of the Asynchronous approach is due to the high load balance among blocks in the Jacobi algorithm in addition to highly efficient latency hiding inherent in the GPU architecture's execution mechanism when the number of threads is plausibly large.

To evaluate the assumption that the synchronization overhead is the reason behind a lower speedup of our approach on the algorithm, we implement an implicitly synchronized projected approach where the synchronization is not needed. We do that by solving many right-hand side set of systems of linear equations, i.e., a set of systems of linear equations that have the same coefficient matrix but different right-hand side

vectors. Solving this problem with sufficiently large coefficient matrix sizes allowed us to drop the synchronization to evaluate our approaches efficiency.

The asynchronous approach for large data sizes need not have an enforcement of a single block per SM. However, a careless choice of an experiment data-size block-size pairing when measuring the speedup of Asynchronous behavior would not represent actual Asynchronous behavior; which could give a wrong impression about the upper limit of enhancement that could be achieved by Synchronous approaches. Since our kernels' maximum residency is 154, appropriate choices for our specific kernel are sizes that are multiples of both block size and 154 when iterations are assigned at least 154 blocks. This way, we can measure an upper limit for the synchronous approach by simply eliminating the post-wait synchronization step while still no blocks will be dispatched to work on iteration before at least one block has finished the previous iteration. This behaviour is the closest we can get to emulate an asynchronous behavior in the projected method to get an accurate upper limit for enhancements expected.

We can see from the figure below that the implicit synchronization approach reaches almost the exact speedup of the asynchronous approach. Our projected approach performs better than the global barrier approach for two reasons.

- Number of memory accesses when performing barriers
- Low thread granularity for the global barrier synchronization approach

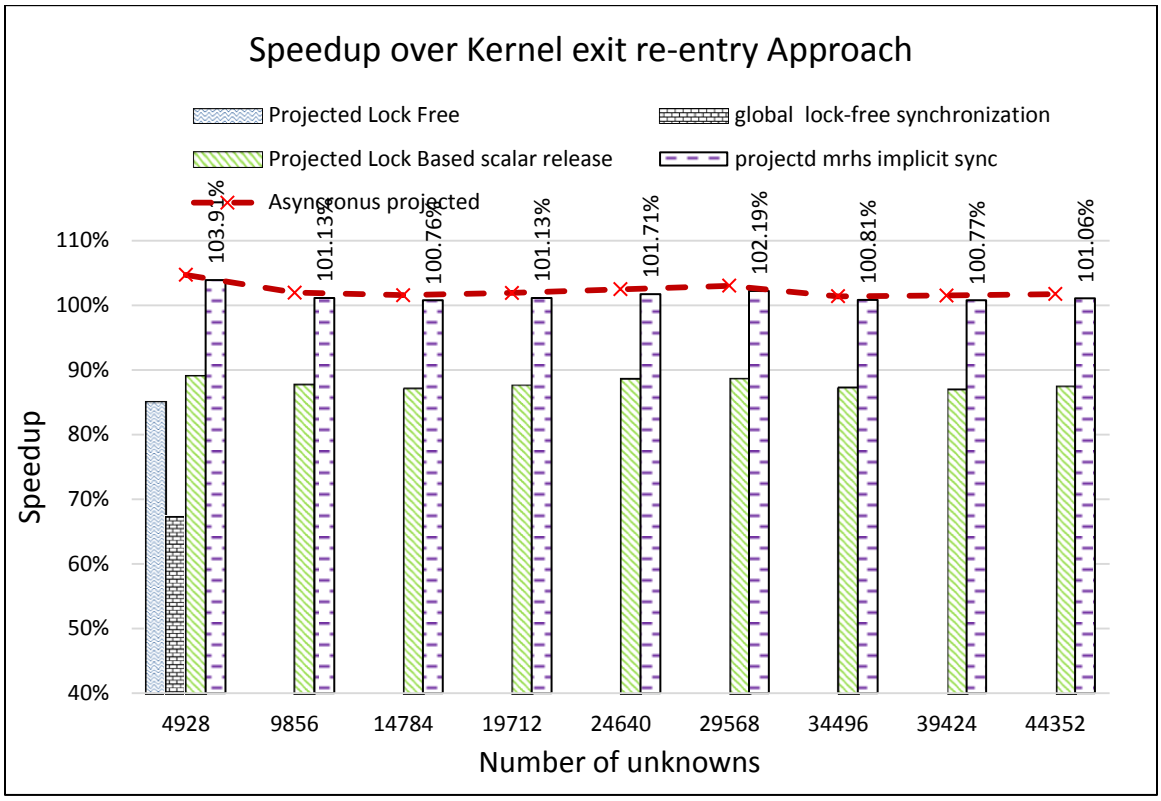

Figure 41: Speedup over Kernel exit re-entry approach of various GPU-orchestrated iteration approaches

4.6.6.5 Analysis of results

The bound of the number of memory accesses for the global synchronization is:

$$((B-1) \times W_c + 2B) \times S \times \text{flag\_size}$$

Where B is the number of blocks, W_c is the number of waiting cycles before the barrier is accomplished and S is the number of synchronization points, which is the same as the number of iterations in our example.

The number of memory accesses of the projected lock-free synchronization approach would be

$$(B-1) \times W_c + B) \times S \times \text{flag size}$$

This number would be lower than that of the globally-synchronized lock-free method. The number of memory accesses of the lock based approach should not be compared to that of a lock based approach since the accesses are of different nature.

To assess the effect of thread granularity on the performance we set up an experiment with a different number of blocks per data size. The number of rows per thread was varied between one row/thread up to the maximum possible for each data size. The figure below indicates how execution times are affected. It clearly shows that the best number of rows per thread is generally one row/thread (they are the global minima per series, which are highlighted by large black dots). This conclusion strictly holds true for data sizes larger than 16000.

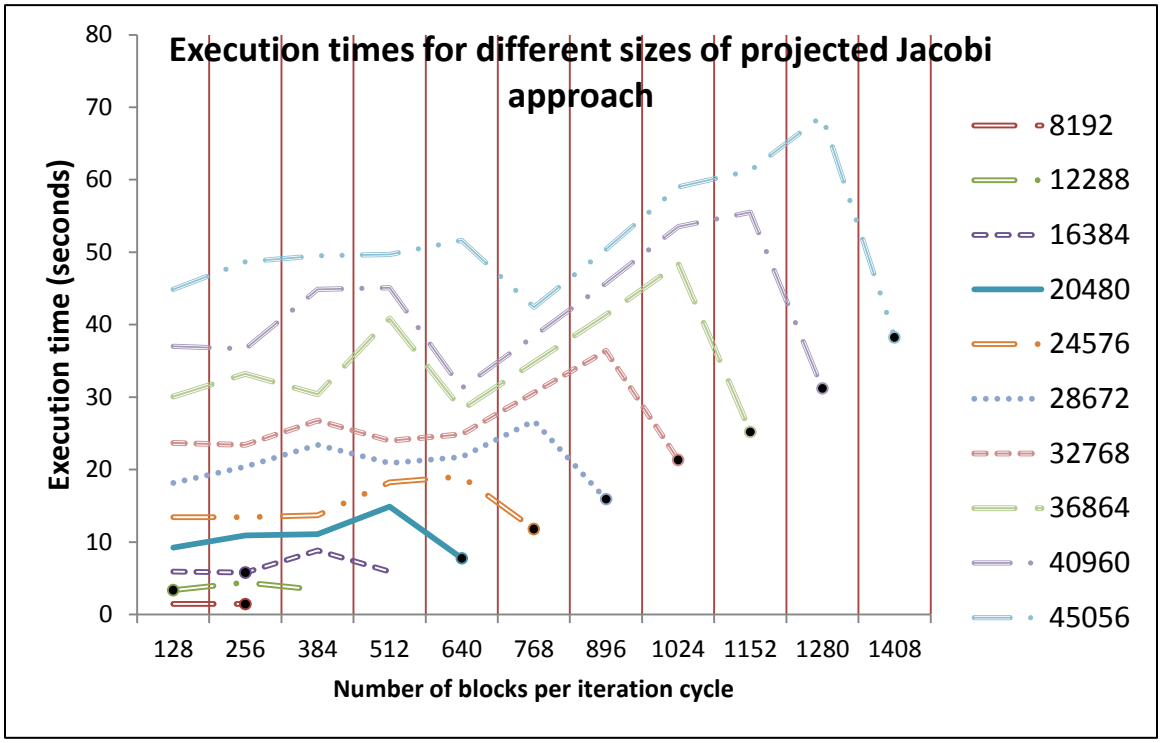

Figure 42: Tuning Jacobi for best performance against number of blocks per iteration cycle for different data sizes

The use of shared memory tiling enforces an extra synchronization inside each block; one after reading data to shared memory and the other after accomplishing operations on the shared data. However, using blocks of size 32 opens an opportunity to make the

programming warp-synchronous when using the 32 thread sized blocks we are using for tiling, which allows omitting the intra-block synchronization points needed inside blocks. However, we did not remove the synchronization operation. It is sufficient to keep block size equal to warp size for synchronization overhead to be negligible.

Another example we experimented with is matrix exponentiation. Matrix exponentiation takes an important role in scientific methods such as graph theory. We implemented an ordinary matrix multiplication approach that multiplies the original matrix by itself p-1 times for a power of p. Our goal in this experiment obviously is not to implement the best matrix multiplication algorithm, but rather to compare what effect our projection method have on such as, algorithms with multiple matrix multiplication algorithms.

4.7 Discussion: Cases of true data flow dependence in iterative applications

The maximal and maximum residency resource limit in GPUs, as discussed earlier enforces a form of dispatch based serialization for the block. We already discussed the utilization of the dispatch order to facilitate the orchestration of loops inside GPUs. The fact that there is a hardware limit on the concurrency windows size of blocks allows us to parallelize some applications without worrying about the synchronization overhead.

Our approach would orchestrate loops over blocks in GPU per two cases

- **Case 1**: the application needs explicit synchronization happens when dependence distance between blocks is less than or equal to execution window size
- **Case 2**: synchronization can be omitted (best case for performance) Happens when dependence distance between blocks is larger than execution window size. Synchronization here would have a minimal impact when implemented efficiently since data would have already been produced when consumer blocks are dispatched

- Developers, in that case, could choose to omit synchronization since it already is implied because of hardware dispatching mechanism and the limited execution window size. However, there should be some safeguards implemented for future portability in case the application may run on hardware that can provide a larger concurrency window size.

The data flow dependence between blocks representing different iteration cycles in an algorithm can be classified into three models. We want to find scenarios where applications can run under case 2 for each of these models. We tackle these scenarios on the next three sections

4.7.1 ONE TO ONE BLOCK DEPENDANCE PATTERN

The simplest of scenarios is when each block in the data decomposition needs the output of a corresponding block assigned to a prior iteration.

Examples of such pattern can be found in matrix exponentiation and red-black relaxation for ocean simulation.

For applications of this pattern to run under case 2 scenarios, the data decomposition needs to be considered in a way to increase dependence distance as much as possible. One such decomposition is the row based one for matrix multiplication. When matrix size is larger than a set limit, the application can run under case two for calculating exponents of a matrix. Figure 43 demonstrates an example during the first stage of how case two would be applied in such scenario. During that phase, the GPU is totally occupied by calculations related to a single iteration. In the last stages of an iteration cycle, however, the situation in a projected approach starts to pay. Figure 44 shows a demonstration of such a scenario where blocks from the next iteration cycle start execution. Because of the data decomposition involved, in addition to a problem instance size that is larger than the execution window, there will be no race conditions in calculating blocks concurrently.

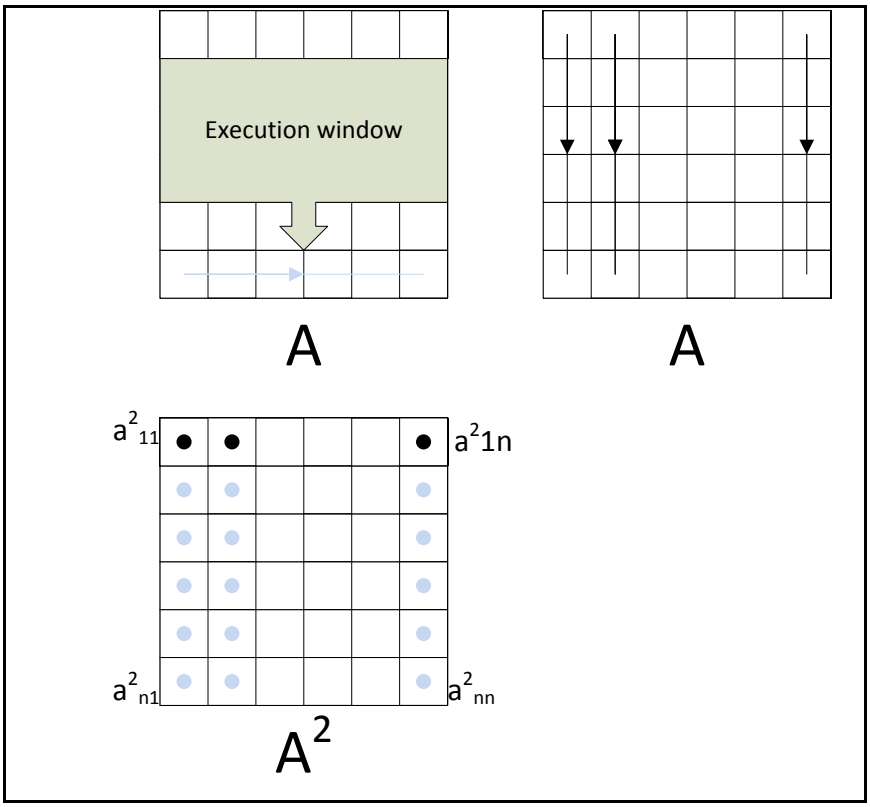

Figure 43: A demonstration of how Matrix exponentiation works during the first stage to calculate A^2.

We implemented matrix powering in an application that demonstrates this idea, in a block size setting of 32×32 threads (1024 threads per block, i.e. 32 warps per block for warps of size 32). Each SM of our GPU card can accommodate a maximum of 64 warps (i.e. two blocks) this can be seen in profiling results shown in Figure 46. Hence for a fifteen SM the execution window size is 2×15 = 30 blocks. Factors and parameters affecting this window size are shown in Figure 47, where we see that we have reached the maximum occupancy for that GPU in our application.

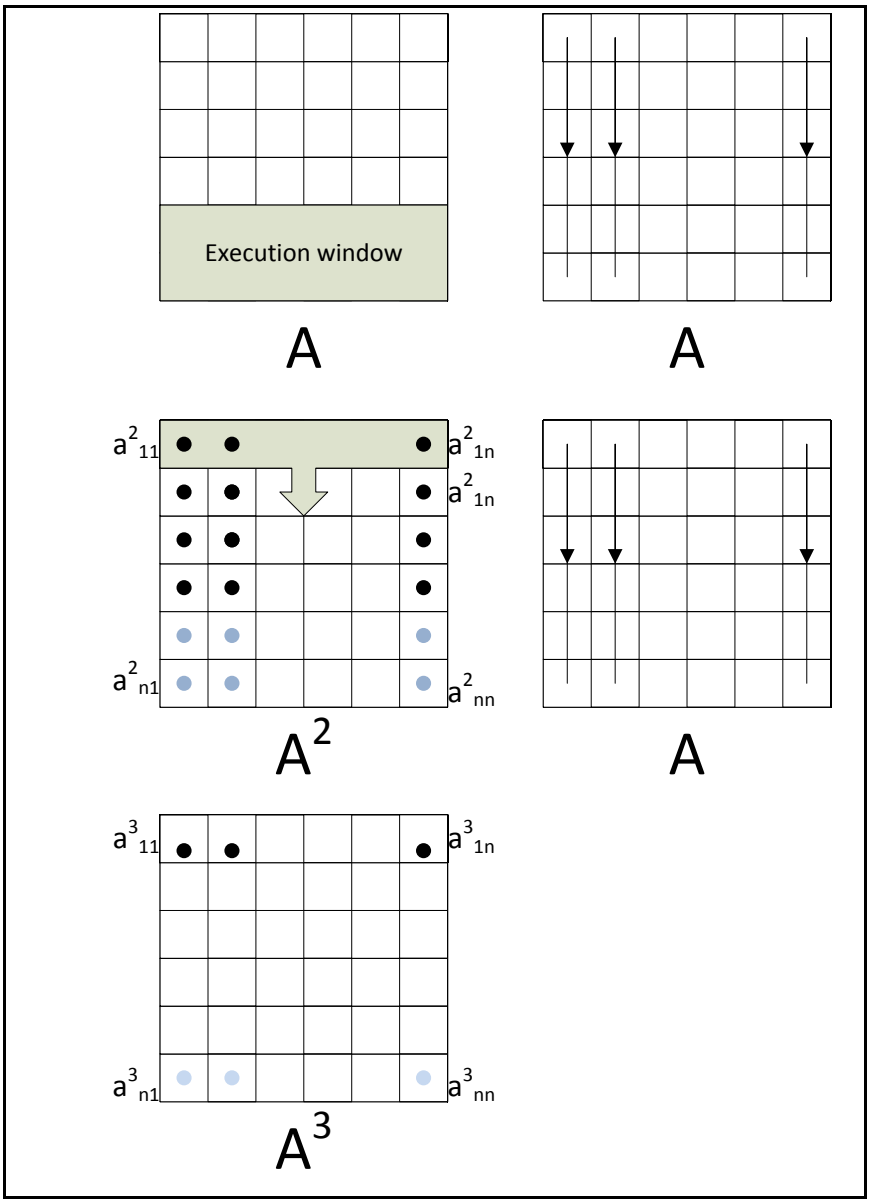

Figure 44: The second phase of matrix exponentiation. When the dispatcher starts dispatching blocks that will work on the next iteration

In input-row based data problem decomposition, the execution window size is 30×32=960 rows. Since execution is not guaranteed to be finished in a FIFO manner, we should double this number to become 1920. Depending on the application, however, (including the one we are discussing) this safety margin can be reduced. As a generic rule, the safety multiplier should not be less than nor equal to the least dependence distance allowed (MDD). MDD can be calculated per the following formula:

$$MDD = 1 + \frac{Number\ of\ Multiprocessors}{Execution\ window\ size}$$

In our example of matrix powers the minimum dependence distance MDD = 1+15/30 = 1.5. Hence minimum dependence distance would be 1.5 × 30 = 45 blocks or 1.5 × 960 = 1440 rows. However, we will start with matrix sizes of 4096×4096.

We found that in our experiments, the speedup merely reached 0.10 %, a very modest speed up that is related to the fact that the implementation of matrix powers has a very high utilization rate for all multiprocessors involved. For example, the utilization of multiprocessors on a K20 GPU card with 13 multiprocessors is shown in Figure 48. This high utilization rate leaves no room for any enhancement on the scale of the processor utilization.

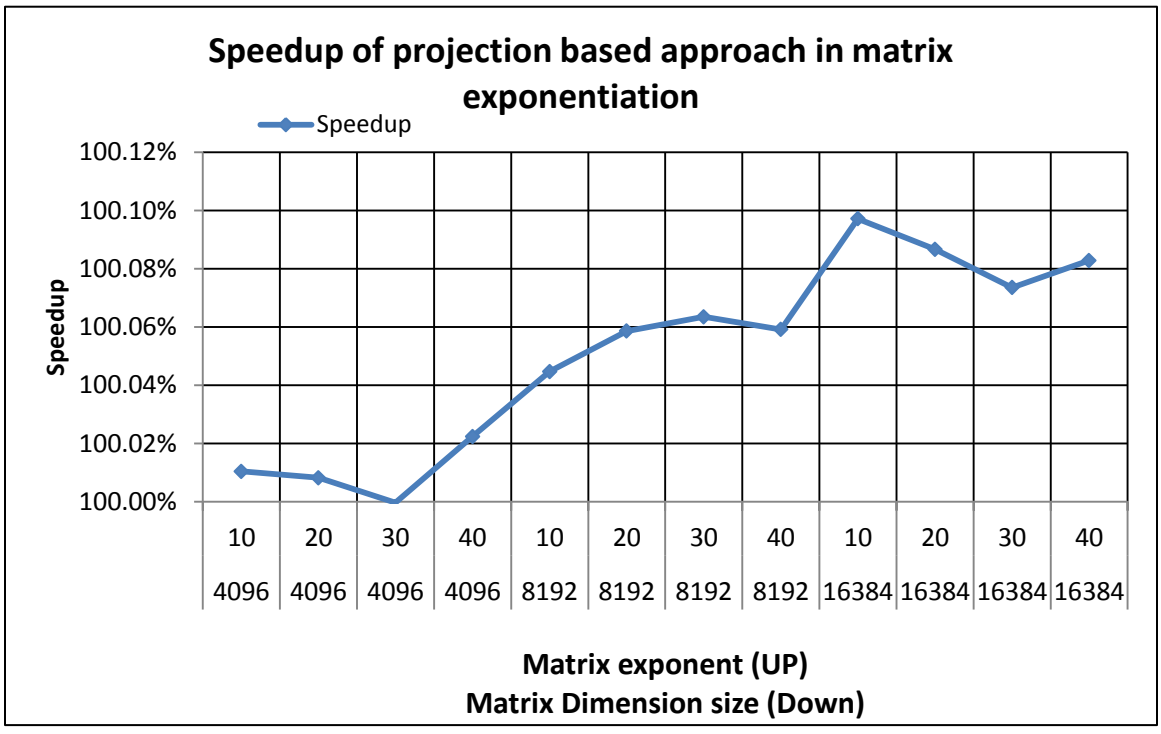

Figure 45: Speedup of projection based approach in matrix exponentiation application against Power and Size of matrix

Variable	Achieved	Theoretical	Device Limit	Grid Size: [20,20,1] (400 blocks)Block Size: [32,32,1] (1024 threads)
Occupancy Per SM				
Active Blocks		2	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Active Warps	63.31	64	64	0 5 10 15 20 25 30 35 40 45 50 55 60 64
Active Threads		2048	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	98.9%	100%	100%	0% 15% 30% 45% 60% 75% 90% 100%

Figure 46: Profiling the matrix exponentiation application shows that residency is two per SM (out of 16), limited by hardware warp and thread limit

Figure 49 shows the distribution of operations in a matrix powers application run. Even though the application is primarily memory bound, the processors are almost fully utilized. This utilization happens because a GPU multi-processor is considered fully utilized when it is assigned its maximum capacity of warps. Hence, multiprocessors are only underutilized in the last phase of every iteration cycle. However, due to good load balancing profile of the application, the underutilization is very minuscule.

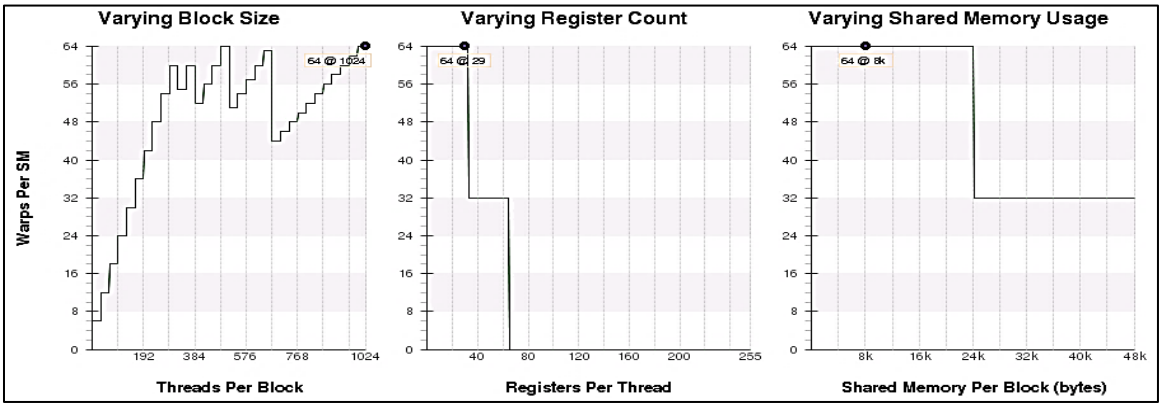

Figure 47: Parameters affecting SM residency in matrix exponentiation application

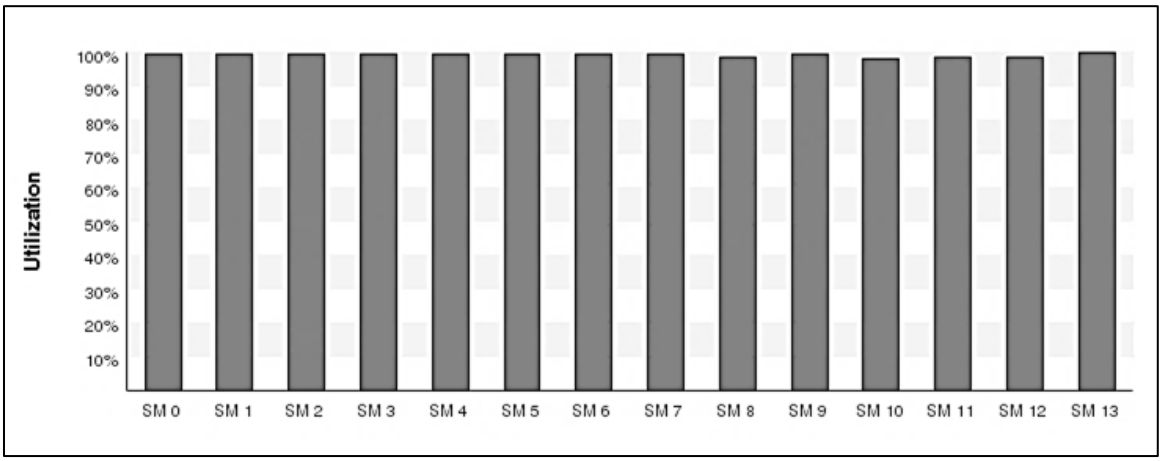

Figure 48: SM utilization by the matrix exponentiation application for kernel exit-reentry

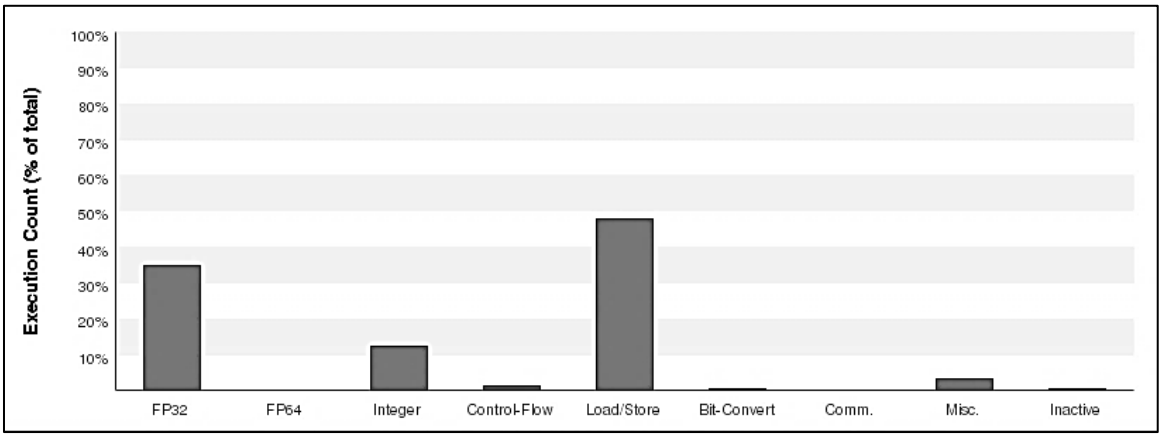

Figure 49: The distribution of multiprocessor operations for the matrix exponentiation application.

4.7.2 MANY TO ONE

A more complex scenario is when each block of the blocks assigned into iteration consumes the output of several blocks in the previous iteration.

One such example for this case is the pre-stack depth migration algorithm, used in reservoir simulation. Running this type of pattern under case two is similar to the previous case;the least dependence distance need to be larger than the target machines concurrency window size. Another example is the prefix sum operation.

4.7.3 ALL TO ALL

In an ALL TO ALL dependence pattern, each block of the blocks assigned to iteration consumes data produced by all blocks of the previous iteration.

Examples of this communication pattern are Jacobi linear solver, Bicg-stab and QMR iterative solvers ADI and N-body.

Applications with such pattern cannot depend on implicit synchronization sync the dependence distance is always a single block. For such applications to be able to use implicit synchronization we need to alternate between the iterations of two instances of the program for it to be beneficial. We had already seen some enhancement in performance for the Jacobi when we alternated between the iterations of two instances of Jacobi solving for a two right-hand side system of equations. However, that enhancement was modest again as a result of the good utilization profile for the original solution as can be shown in Figure 50.

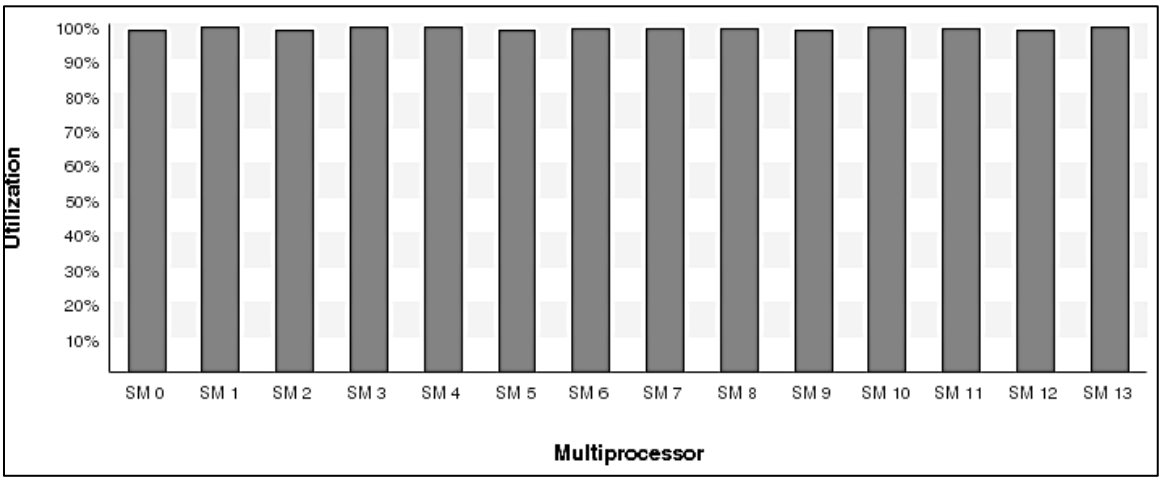

Figure 50: Multiprocessor utilization of a k20c GPU card by the Jacobi linear solver CPU based synchronization

4.8 Performance of Projective based approach in load-imbalanced applications

The reader could have noticed that in many linear algebra numerical simulations, there is generally a good load balance among thread blocks, i.e. they are load balanced. However, we wanted to test how the approach would perform under load imbalanced conditions. So, we injected an artificial load imbalance inside the Jacobi multiple right-hand side systems solver. To accomplish this, we enforced an extra random wait up to 30% of the original workload of each block.

The new load balance profile is shown in Figure 51. The speedup in Figure 52 shows that our approach reaches about 80% in some cases.

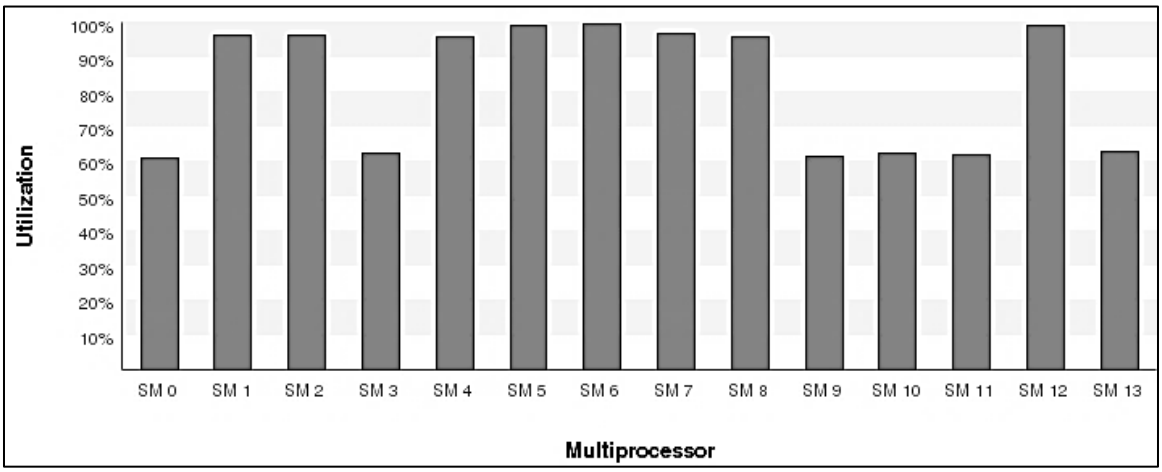

Figure 51: Utilization for Kernel exit-entry based MRHS with artificial load imbalance.

The reader may also have noticed that there are fluctuations and irregularity in the speedup. To analyze the reason for this irregularity, we study the original execution time data through Figure 53. We can see that there are irregular jumps in performance magnified in Figure 54. These partial irregularities and fluctuations introduce instability in the execution profile of the kernel exit-entry based approach.

To know the reason behind these irregularities, we graph the execution time against the number of blocks as shown in Figure 56. We can now clearly see that large jumps in execution time are happening exactly at cycles of 165 thread blocks. We can notice that this number represents the residency limit for the K40 GPU where it has 15 multiprocessors; $15 \times 11 = 165$. So in sizes in which blocks assigned to an iteration cycle are equally spread over the GPU board, we see a large performance decrease in CPU based Approach. The results of profiling for the utilization on the sizes with performance jumps shows that ti is indeed the case

Another set of partial irregularities in the kernel exit-entry approach is found on a smaller scale, magnified in Figure 57 shown on the graph of execution time against the number of blocks. Those irregularities are happening at an interval of 12 blocks, which is a change in the size of exactly 384 elements. The only link we could find is between distance between jumps and the memory interface of the GK110 chip being 384-bits.

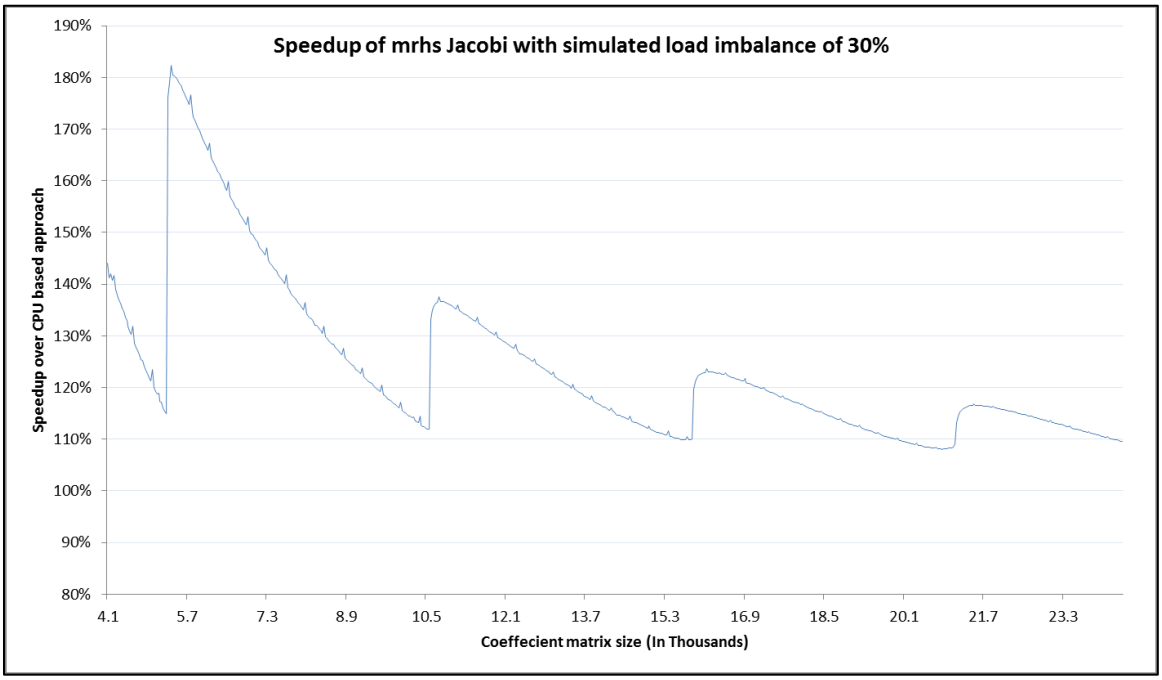

Figure 52: Speedup of Jacobi MRHS with simulated load imbalance of 30% between blocks of threads

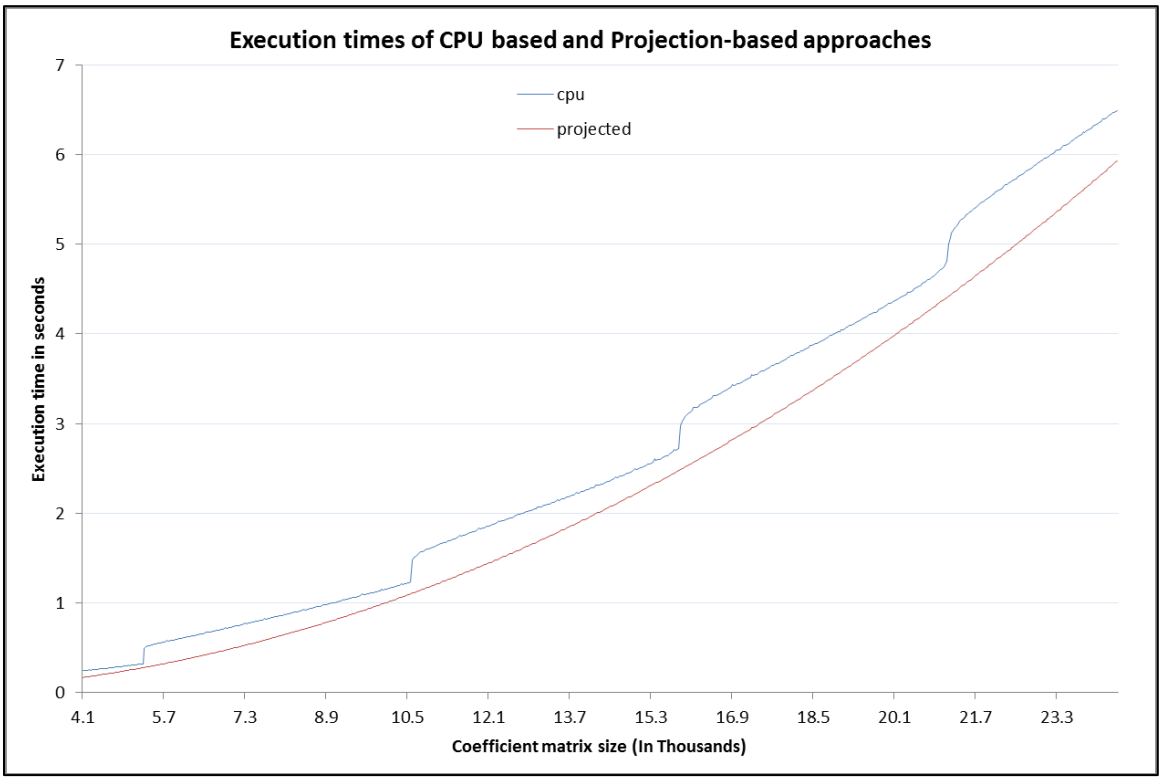

Figure 53: Execution times of both CPU based and projection-based approaches against coefficient matrix size

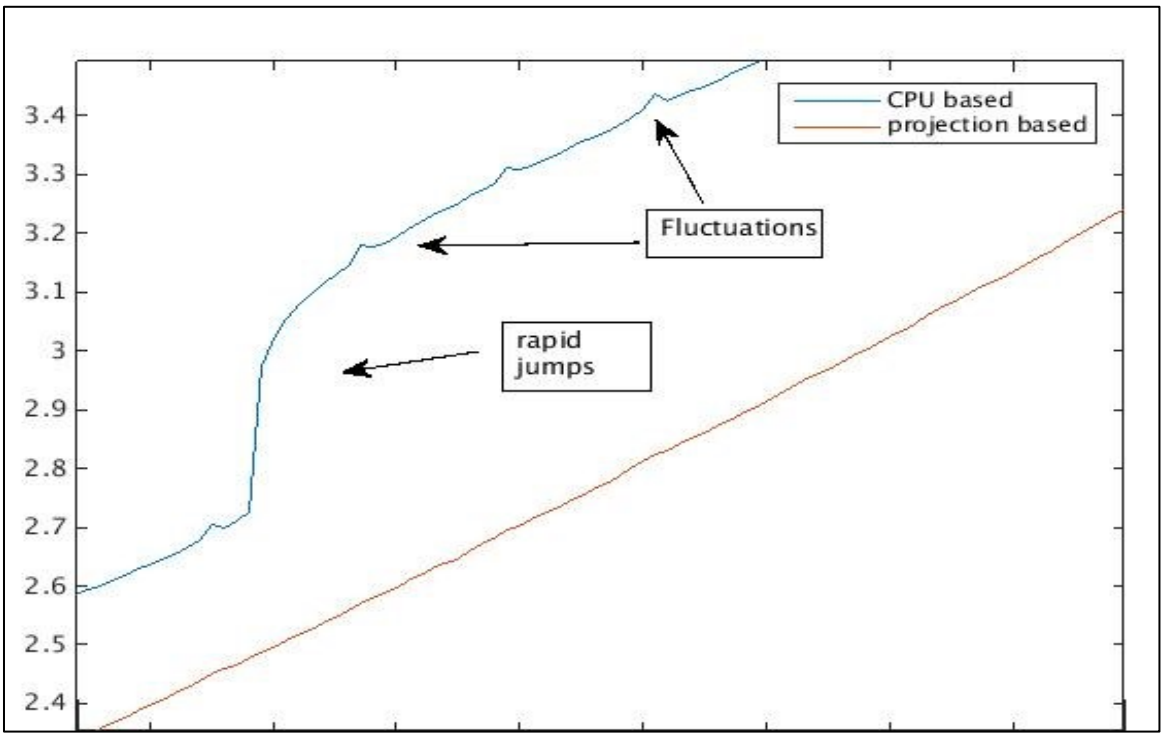

Figure 54: Partial irregularity of execution time in CPU based synchronization approach

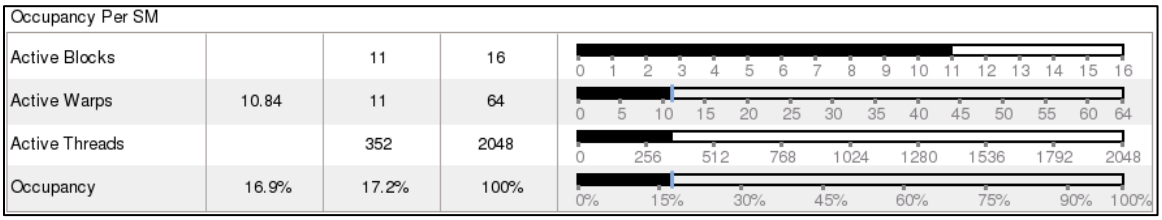

Figure 55: Occupancy profile of the Jacobi Kernel showing a residency of 11 blocks per SM (out of 16)

Finally, we can conclude that our approach is more appropriate for iterative applications that have load imbalance between blocks of threads. For applications to have a utilization of implicit dispatch based ordering, the inter-block dependence distance between iterations needs to be sufficiently large to surpass 1.5 times of the block residency of the target GPU. When this dependence distance cannot be met; multiple instances of the application need to be available to utilize implicit synchronization. Our speedup for Jacobi MRHS reached 80% over kernel exit-entry based approach. The speedup percentage in our approach reaches its maximum for relatively non-huge data sizes as discussed earlier.

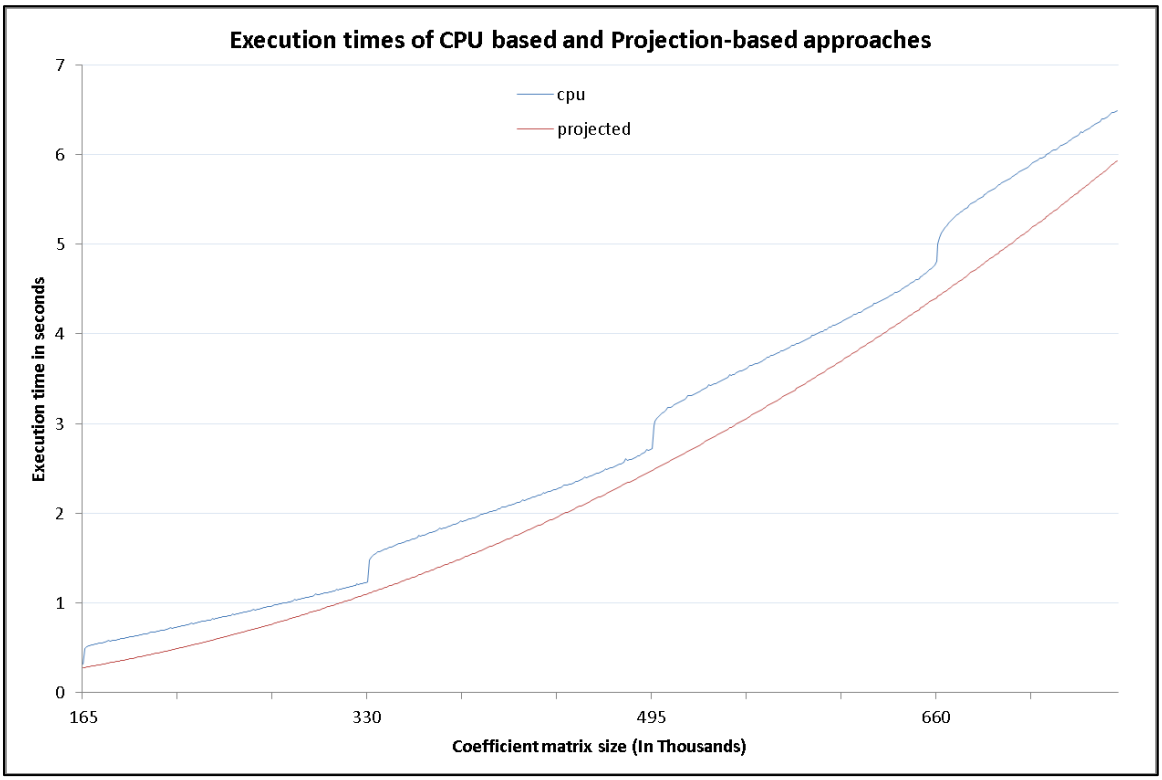

Figure 56: Execution time of Jacobi kernel exit-entry based approach and projected based approach against the number of blocks.

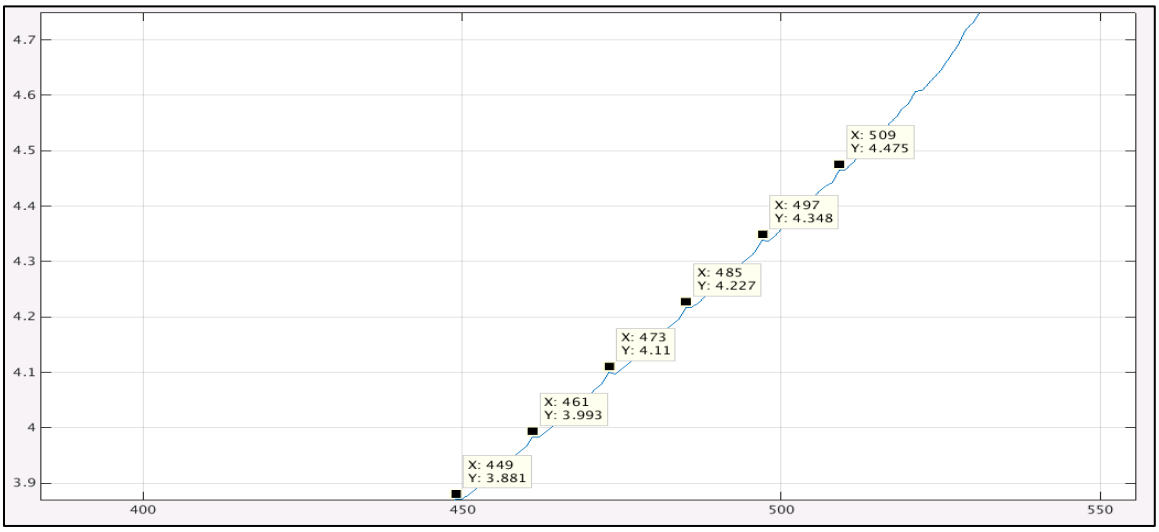

Figure 57: Partial irregularities in kernel exit-entry based execution time against number of GPU thread blocks

4.9 Discussion: Projection as a suggested automatic optimization

Since OpenACC Framework model depends on annotations over iterative serial code; it is a straightforward step to suggest iteration space projection into an OpenACC implementation. The Incorporation of the new optimization can be done using an additional #pragma acc project statement to allow projection optimization to occur inside the kernels or parallel construct. The following directive transformation would be used in such cases to eliminate serially repetitive kernel launches.

Listing 32: Suggested form for the addition of projection method as an optimization in OpenACC

```
01  #pragma acc kernels
02   {
03  for () // outer serial loop    {
04  #pragma acc loop gang
05  for () {
06     // application to be
    parallelized
07      } // end of gang loop
08  }/*outer loop*/
09  }//end of kernels block
```

```
01  #pragma acc kernels
02   {
03  #pragma acc loop project
04  for () // outer serial loop    {
05  #pragma acc loop gang
06  for () {
07     // application to be
    parallelized
08      } // end of gang loop
09  }/*outer loop*/
10  }//end of kernels block
11  }
```

Upon processing of *project* pragma statement, the compiler would allocate the outer loop as a gang loop while choosing the block ordering based on serialization of block chunks. Many cases are likely to introduce block interdependency in the projected output code. In cases where block interdependency distances that are less than the residency limit R_L of the produced kernel over the target architecture; Inter-block synchronization code shall be injected into output code in cases where that optimization is profitable. That output code shall be of suitable performance based on the specifics of the target architecture.

Notice, however, that the project pragma can be replaced by removing the restriction against nesting (gang) loop. A gang loop nest can be added where scheduling statements are put in place to control the ordering. Listing 33 shows how the addition to the statement can be applied.

Listing 33: Another form of adding loop projection in OpenACC, this form requires allowing "*loop gang*" statements to be nested

```
01  #pragma acc kernels
02   {
03  for () // outer serial loop    {
04  #pragma acc loop gang
05  for () {
06     // application to be
    parallelized
07      } // end of gang loop
08  }/*outer loop*/
09  }//end of  kernels block
```

```
01  #pragma acc kernels
02   {
03  #pragma acc loop gang
    schedule(project)
04  for () // outer serial loop    {
05  #pragma acc loop gang
06  for () {
07     // application to be
    parallelized
08      } // end of gang loop
09  }/*outer loop*/
10  }//end of  kernels block
```

4.10 Testing for correctness of the projection mechanism

It has been mentioned earlier that the block/gang scheduling approach is not discussed in official documentation. For example, the workings of the NVidia giga-thread scheduler are not discussed in official documentation. To establish confidence in the results we obtained and the new methods we introduced, we decided that the following elements are sufficient to insure correctness for the setup we used for testing environment: the first element is the absence of deadlocks in the experiments we conducted using our explicitly-synchronized projection approach. Since the numbers of blocks used in most runs were larger than the maximal residency of the GPU used to run the test. This is because the algorithm for synchronization ensures that the program either produces correct results or it deadlocks; because the wait operations in the synchronization code are blocking. The second element is the contrast between these results and the ones in section 4.5 from runs over the global synchronization : Even though we are launching a significant number of blocks and doing blocking wait synchronization on them, all runs halted successfully. This is contrasted to the results showing the existence of deadlocks in global rendezvous-style synchronization in section 4.5.

We chose the Jacobi implementation for this experiment since its largest possible sizes are fast relative to the matrix exponentiation. This choice allows us to run a lot of experiments for a large variation of sizes in a reasonable time.

We established a code that makes runs of projected approach and a corresponding runs on kernel exit-entry approach. The number of matches and mismatches of the vector result of the linear system is counted and reported by the application. For the projected approach, the size variations consisted of 28 samples with each sample being run for a variety of all even numbered iterations between 2 to 100. Each size-iteration variation was run for 15 times. In the first runs of the projected approach, the linear system matrix was kept constant for each size. The total number of the first round of runs for the projected approach was 21E3 times. The error counter is increased whenever a difference between corresponding numbers exceeds 1e-3.

The outcome of the experiment was that all results matched successfully. There were absolutely no mismatches between the projected approach and the kernel exit re-entry one.

A similar experiment was conducted for the Jacobi MRHS for sizes that provide safe dependence distance between blocks on k40 and k20 processors. In total, 26 different sizes with all even number iterations between 2 to 100 were run 15 times each for a total of 19500 runs with random input matrices appropriate for Jacobi linear solver. Absolutely all runs matched successfully with zero number of mismatches.

Another extra set of 840 instance runs was done on the Jacobi projected approach with an error tolerance of 1E-6, while again exact all-matching results had been obtained.

4.11 Conclusion

In conclusion, using inter-block synchronization techniques as recommended in previous literature[86] [78] produces programs that are prone to deadlocks if not used carefully. We have shown how the limitation on the number of blocks to avoid deadlocks reduces performance in comparison to other traditional approaches of repetitive kernel launches. In this chapter, we have deducted an upper limit for the number of blocks to be used in such manner (rendezvous limit) and a maximum upper bound that depends on GPU

model that we called Maximal residency. For that, we have provided a systematic methodology for the analysis of the relationship between SMs, blocks, and barriers. We see a future for this methodology in helping developers analyze the behavior of GPU programs coordinating blocks through globally shared atomic variables.

We also have analyzed and reverse engineered how the NVidia GPU block scheduler (Giga thread scheduler) schedules and dispatches blocks into multiprocessors. This analysis would help provide more understanding of the workings of the GPU block scheduler.

Based on the scheduler analysis, we have introduced a new novel method for implementing iterative solvers or any one-way producer-consumer algorithm without the need for kernel exit-entry. Our method projects successive iterations into successive blocks on the GPU and ensures their ordering through global atomics. Contrary to approaches in literature, our method does not suffer in limitations for the number of blocks to be launched by GPU kernel.

Based on the analysis of the GPU dispatcher, we shed light on cases where applications do not need synchronization at all using our approach. The performance increase is profitable in cases where the kernel exit-entry synchronization would cause the GPU multiprocessors are to be not fully utilized. Such cases would mostly occur in applications where there is no load imbalance between blocks.

Finally, we suggest that for the block scheduler behavior to be documented officially to provide a guarantee of compatibility for future generations of hardware.

CHAPTER 5
Conclusion and Future Work

This study was set out to explore and enhance optimizations for annotation based parallel compilers for GPUs for the goal of easier programming and better performance in terms of execution time. GPUs have gained a lot of attention in the HPC community; a lot of research was done on creating and optimizing language models that enable programming these devices. OpenACC standard has emerged to standardize the effort of creating a high-level directive based language extension for several conventional programming languages to enable easy programming of these devices. our study begun by confirming that Although OpenACC enables access to GPU computing to a wider audience; it's generated code usually falls behind in performance to less abstract but carefully programmed methods such as the ones in CUBLAS library. One of the features in OpenACC is the enablement of calling GPU functions/routines from within Compute region of the code; which should enable easy access to a set of highly performing GPU libraries. However, our study also confirmed that of some established libraries that are implemented on GPUs cannot be called from these compute regions. The reason behind that is lack of ability to interface to argument of opaque pointer type. We wanted to know whether using the device interface of a library would carry any advantage against calling these libraries from a CPU section in the program. therefore we created a layer that hides the opaque pointer data type arguments from CUBLAS' device interface which , when tested, allowed us to confirm that using the device interface indeed carries a an added value in terms of better structured OpenACC programs and more performance in some cases. To develop a systematic approach that enables access to a wide variety of device interfaced libraries to OpenACC developers. This was the reason for creating ***libimorph***, which is a system that takes a code template from the user and use it to generate a layer that hides opaque pointer arguments from the client code. The developer then needs to only develop a simple code. For our use of this system, we decided to manually add an extra internal management for CUBLAS handles through an internal array in device

memory to enhance performance. The layer we generated had better performance for many cases and less code size than using OpenACC alone. One of OpenACC implementers (PGI) actually added the capability to call CUBLAS device routines to their OpenACC implementation based on our request in the PGI forums.

In future, we would like to see our ***libimorph*** applied to a wider variety of library device routines. In particular, we would like to see this applied to the CURAND library for random number generation. We also would like to see advanced restructuring of the data composition based on the use of the device interface since a device interface allows more freedom in the algorithmic approach with less of the unneeded synchronization points that used to be put in place to be able to call GPU library routines from the CPU code.

Another problem that hinders performance In GPU programs is the method of synchronization between different thread blocks. In current programming models, synchronization is only possible through the host systems processor. Hence, it is only possible to implement synchronous iterations through repeated launches of GPU kernels for each iteration step. Some techniques were proposed to reduce that overhead using shared variables. However, these are prone to deadlocks and are error prone when implemented without correct enforcement of memory coherency. In this dissertation, we comprehensively analyzed the reasons and parameters involved in the deadlock-inducing behavior in such synchronization techniques, where we presented crisp upper limits for parameters involved to avoid deadlocks in these techniques. We also analyzed the scheduling behavior for different gangs inside the GPU. Based on the scheduler analysis, we have introduced a new novel method for implementing synchronized iterations or any one-way producer-consumer communication method without the need for synchronization through the host CPU and without limitations for the number of blocks involved in the synchronization. Our analyses showed that the performance of our method had a speedup of about 1.38 over inter-block synchronization methods used in literature. We also introduced analysis for the cases and parameters where it is possible to omitt the explicit shared variable synchronization all together and gaining performance similar to asynchronous algorithms via arranging for an implicit scheduling-based

synchronization for such cases to see a gain in speedup nearing 1.8 for applications where blocks do not share a balanced load among them. We also concluded that for a large majority of the cases where an explicit synchronization is needed, host-based synchronization is the best choice. We suggested adding our synchronization method into OpenACC standard and suggested the annotation structure for such addition.

In the future, we would like to see our method tested on a wider variety of GPU applications and applied to synchronization applications other than iteration handling.

Appendix A
AwCuBLAS API description

A.1 AwCuBLAS API listing and mapping

Here we list each CuBLAS function and the corresponding needed function call that we need to invoke. Notice, however, that to call CuBLAS from within OpenACC regions, we only need to invoke the function call. We do not need to transfer data, nor we need to create a CuBLAS context; just a single function call.

We list later the description of some of these functions that we deem relevant. For an explanation of these CuBLAS functions, please consult the CuBLAS manual on the web http://docs.nvidia.com/cuda/cublas/. The only difference is the change of ***cublasHandle_t*** arguments into integer ones; this difference of course is in addition to the name change.

CuBLAS functions that can be called from OpenACC compute regions, as they are in the CuBLAS version 6.5 that was transformed into AwCuBLAS.

```
01 invoke_Snrm2
02 invoke_Dnrm2
03 invoke_Scnrm2
04 invoke_Dznrm2
05 invoke_Sdot
06 invoke_Ddot
07 invoke_Cdotu
08 invoke_Cdotc
09 invoke_Zdotu
10 invoke_Zdotc
11 invoke_Sscal
12 invoke_Dscal
13 invoke_Cscal
14 invoke_Csscal
15 invoke_Zscal
16 invoke_Zdscal
17 invoke_Saxpy
18 invoke_Daxpy
19 invoke_Caxpy
20 invoke_Zaxpy
21 invoke_Scopy
22 invoke_Dcopy
23 invoke_Ccopy
24 invoke_Zcopy
25 invoke_Sswap
26 invoke_Dswap
27 invoke_Cswap
28 invoke_Zswap
29 invoke_Isamax
30 invoke_Idamax
31 invoke_Icamax
32 invoke_Izamax
33 invoke_Isamin
34 invoke_Idamin
35 invoke_Icamin
36 invoke_Izamin
37 invoke_Sasum
38 invoke_Dasum
39 invoke_Scasum
40 invoke_Dzasum
41 invoke_Srot
42 invoke_Drot
43 invoke_Crot
44 invoke_Csrot
45 invoke_Zrot
46 invoke_Zdrot
47 invoke_Srotg
48 invoke_Drotg
49 invoke_Crotg
50 invoke_Zrotg
51 invoke_Srotm
52 invoke_Drotm
53 invoke_Srotmg
54 invoke_Drotmg
55 invoke_Sgemv
56 invoke_Dgemv
57 invoke_Cgemv
58 invoke_Zgemv
59 invoke_Sgbmv
60 invoke_Dgbmv
61 invoke_Cgbmv
62 invoke_Zgbmv
63 invoke_Strmv
64 invoke_Dtrmv
65 invoke_Ctrmv
66 invoke_Ztrmv
```

```
67 invoke_Stbmv
68 invoke_Dtbmv
69 invoke_Ctbmv
70 invoke_Ztbmv
71 invoke_Stpmv
72 invoke_Dtpmv
73 invoke_Ctpmv
74 invoke_Ztpmv
75 invoke_Strsv
76 invoke_Dtrsv
77 invoke_Ctrsv
78 invoke_Ztrsv
79 invoke_Stpsv
80 invoke_Dtpsv
81 invoke_Ctpsv
82 invoke_Ztpsv
83 invoke_Stbsv
84 invoke_Dtbsv
85 invoke_Ctbsv
86 invoke_Ztbsv
87 invoke_Ssymv
88 invoke_Dsymv
89 invoke_Csymv
90 invoke_Zsymv
91 invoke_Chemv
92 invoke_Zhemv
93 invoke_Ssbmv
94 invoke_Dsbmv
95 invoke_Chbmv
96 invoke_Zhbmv
97 invoke_Sspmv
98 invoke_Dspmv
99 invoke_Chpmv
100 invoke_Zhpmv
101 invoke_Sger
102 invoke_Dger
103 invoke_Cgeru
104 invoke_Cgerc
105 invoke_Zgeru
106 invoke_Zgerc
107 invoke_Ssyr
108 invoke_Dsyr
109 invoke_Csyr
110 invoke_Zsyr
111 invoke_Cher
112 invoke_Zher
113 invoke_Sspr
114 invoke_Dspr
115 invoke_Chpr
116 invoke_Zhpr
117 invoke_Ssyr2
118 invoke_Dsyr2
119 invoke_Csyr2
120 invoke_Zsyr2
121 invoke_Cher2
122 invoke_Zher2
123 invoke_Sspr2
124 invoke_Dspr2
125 invoke_Chpr2
126 invoke_Zhpr2
127 invoke_Sgemm
128 invoke_Dgemm
129 invoke_Cgemm
130 invoke_Zgemm
131 invoke_Ssyrk
132 invoke_Dsyrk
133 invoke_Csyrk
134 invoke_Zsyrk
135 invoke_Cherk
136 invoke_Zherk
137 invoke_Ssyr2k
138 invoke_Dsyr2k
139 invoke_Csyr2k
140 invoke_Zsyr2k
141 invoke_Cher2k
142 invoke_Zher2k
143 invoke_Ssyrkx
144 invoke_Dsyrkx
145 invoke_Csyrkx
146 invoke_Zsyrkx
147 invoke_Cherkx
148 invoke_Zherkx
149 invoke_Ssymm
150 invoke_Dsymm
151 invoke_Csymm
152 invoke_Zsymm
153 invoke_Chemm
154 invoke_Zhemm
155 invoke_Strsm
156 invoke_Dtrsm
157 invoke_Ctrsm
158 invoke_Ztrsm
159 invoke_Strmm
160 invoke_Dtrmm
161 invoke_Ctrmm
162 invoke_Ztrmm
163 invoke_SgemmBatched
164 invoke_DgemmBatched
165 invoke_CgemmBatched
166 invoke_ZgemmBatched
167 invoke_Sgeam
168 invoke_Dgeam
169 invoke_Cgeam
170 invoke_Zgeam
171 invoke_SgetrfBatched
172 invoke_DgetrfBatched
173 invoke_CgetrfBatched
174 invoke_ZgetrfBatched
175 invoke_SgetriBatched
176 invoke_DgetriBatched
177 invoke_CgetriBatched
178 invoke_ZgetriBatched
179 invoke_StrsmBatched
180 invoke_DtrsmBatched
181 invoke_CtrsmBatched
182 invoke_ZtrsmBatched
183 invoke_SmatinvBatched
184 invoke_DmatinvBatched
185 invoke_CmatinvBatched
186 invoke_ZmatinvBatched
187 invoke_SgeqrfBatched
188 invoke_DgeqrfBatched
189 invoke_CgeqrfBatched
190 invoke_ZgeqrfBatched
191 invoke_SgelsBatched
192 invoke_DgelsBatched
193 invoke_CgelsBatched
194 invoke_ZgelsBatched
195 invoke_Sdgmm
196 invoke_Ddgmm
197 invoke_Cdgmm
198 invoke_Zdgmm
199 invoke_Stpttr
200 invoke_Dtpttr
201 invoke_Ctpttr
202 invoke_Ztpttr
203 invoke_Strttp
204 invoke_Dtrttp
205 invoke_Ctrttp
206 invoke_Ztrttp
```

A.2 AWCUBLAS API Description

The API consists, for each function name in CUBLAS, we precede it with the word "invoke_" (invoke suffixed with underscore) then we invoke it with the usual argument

list (but without the CublasHandle_t data type, as handles are automatically managed). In the following, we list some of the API function prototypes. For a complete list, it is best to consult the official CuBLAS toolkit documentation [70] while doing a similar mapping between AwCuBLAS and CuBLAS to the one done in Appendix A

A.2.1 invoke_<t>nrm2

A.2.1.1 Syntax

```
cublasStatus_t invoke_Snrm2(
                                int awHandle,  int n,  const float *x,
                                int incx,      float *result);
cublasStatus_t invoke_Dnrm2(
                                int awHandle,  int n,  const double *x,
                                int incx,      double *result);
cublasStatus_t invoke_Scnrm2(
                                int awHandle,   int n,  const cuComplex *x,
                                 int incx,      float *result);
cublasStatus_t invoke_Dznrm2(
                                int awHandle,   int n,
                                 const cuDoubleComplex *x,    int incx,
                                 double *result);
```

A.2.1.2 Description

This API function is responsible for computing Euclidean norm of a vector x in single, double, single complex and double complex precision respectively. Mathematically, the result is computed using the expression $\sqrt{\sum_{i=0}^{n-1}(x[j] \times x[j])}$ where j = (i-1) × incx

A.2.2 invoke_<t>dot

A.2.2.1 Syntax

```
cublasStatus_t invoke_Sdot (
                                int awHandle,   int n, const float *x,
                                 int incx,      const float *y,       int incy,
                                 float *result);
cublasStatus_t invoke_Ddot (
                                int awHandle,   int n, const double *x,
                                 int incx,      const double *y,      int incy,
                                 double *result);
cublasStatus_t invoke_Cdotu (
                                int awHandle,   int n, const cuComplex *x,
                                 int incx,      const cuComplex *y,   int incy,
                                 cuComplex *result);
cublasStatus_t invoke_Zdotu (
                                int awHandle,   int n, const cuDoubleComplex *x,
                                 int incx,      const cuDoubleComplex *y,
                                 int incy,      cuDoubleComplex *result);
```

A.2.2.2 Description

These API functions are responsible for computing dot product between vectors x and y in single, double, single complex and double complex respectively. The result can be expressed mathematically by $\sum_{i=0}^{n-1}(x[k] \times y[j])$, where k = i × incx and j = i× incy.

A.2.3 invoke_<t>scal

A.2.3.1 Syntax

```
cublasStatus_t invoke_Sscal(
                                int awHandle,  int n,  const float *alpha,
                                float *x,      int incx);
cublasStatus_t invoke_Dscal(
                                int awHandle,  int n,  const double *alpha,
                                double *x,     int incx);
cublasStatus_t invoke_Cscal(
                                int awHandle,  int n,  const cuComplex *alpha,
                                cuComplex *x,  int incx);
cublasStatus_t invoke_Csscal(
                                int awHandle,  int n,  const float *alpha,
                                cuComplex *x,  int incx);
```

```
cublasStatus_t invoke_Zscal(
                                int awHandle,  int n,  const cuDoubleComplex *alpha,
                                cuDoubleComplex *x,   int incx);
cublasStatus_t invoke_Zdscal(
                                int awHandle,  int n,  const double *alpha,
                                cuDoubleComplex *x,   int incx);
```

A.2.3.2 Description

These API functions are responsible for scaling the elements of vector x by constant alpha in single, double complex single and complex double precisions respectively. The complex functions can scale either by a complex or real constant alpha. The result can be expressed mathematically byα x[j], where I = 0 .. n-1 and j = i × incx.

A.2.4 invoke_<t>axpy

A.2.4.1 Syntax

```
cublasStatus_t invoke_Saxpy (
                                int awHandle,  int n,  const float *alpha,
                                const float *x,       int incx,      float *y,
                                int incy);
cublasStatus_t invoke_Daxpy (
                                int awHandle,  int n,  const double *alpha,
                                const double *x,      int incx,      double *y,
                                int incy);
cublasStatus_t invoke_Caxpy (
                                int awHandle,  int n,  const cuComplex *alpha,
                                const cuComplex *x,   int incx,
                                cuComplex *y,  int incy);
cublasStatus_t invoke_Zaxpy (
                                int awHandle,  int n,
                                const cuDoubleComplex *alpha,
                                const cuDoubleComplex *x,    int incx,
                                cuDoubleComplex *y,   int incy);
```

A.2.4.2 Description

These API functions are responsible for scaling the elements of vector x by constant α; and adding it to another vector y all in single, double, complex single and complex

double precisions respectively. The complex functions expect alpha to be complex too. The result can be expressed mathematically by $y[j] = \alpha x[k] + y[j]$, where i = 0, ... , n-1 , k = i × incx and j = i × incy.

A.2.5 invoke_<t>copy

A.2.5.1 Syntax

```
cublasStatus_t invoke_Scopy (
                                int awHandle,  int n,  const float *x,
                                int incx,      float *y,      int incy);
cublasStatus_t invoke_Dcopy (
                                int awHandle,  int n,  const double *x,
                                int incx,      double *y,     int incy);
cublasStatus_t invoke_Ccopy (
                                int awHandle,  int n,  const cuComplex *x,
                                int incx,      cuComplex *y,  int incy);
cublasStatus_t invoke_Zcopy (
                                int awHandle,  int n,
                                const cuDoubleComplex *x,    int incx,
                                cuDoubleComplex *y,   int incy);
```

A.2.5.2 Description

These API functions are responsible for copying (assigning) the elements of vector x into the elements of vector y in single, double, complex single and complex double precisions respectively. The result can be expressed mathematically by $y[j] = x[k]$where i = 0, ... , n-1 , k = i × incx and j = i × incy.

A.2.6 invoke_<t>swap

A.2.6.1 Syntax

```
cublasStatus_t invoke_Sswap (
                                int awHandle,  int n,  float *x,      int incx,
                                float *y,      int incy);
```

```
cublasStatus_t invoke_Dswap (
                                int awHandle, int n, double *x,     int incx,
                                double *y,    int incy);
cublasStatus_t invoke_Cswap (
                                int awHandle, int n, cuComplex *x, int incx,
                                cuComplex *y, int incy);
cublasStatus_t invoke_Zswap (
                                int awHandle, int n, cuDoubleComplex *x,
                                int incx,     cuDoubleComplex *y,   int incy);
```

A.2.6.2 Description

These API functions are responsible for swapping (interchanging) the elements of vector x with the elements of vector y in single, double, complex single and complex double precisions respectively. The result can be expressed mathematically by $y[j] \Leftrightarrow x[k]$where i = 0, ..., n-1 , k = i × incx and j = i × incy.

A.2.7 invoke_I<t>amax

A.2.7.1 Syntax

```
cublasStatus_t invoke_Isamax(
                                int awHandle, int n, const float *x,
                                int incx,     int *result);
cublasStatus_t invoke_Idamax(
                                int awHandle, int n, const double *x,
                                int incx,     int *result);
cublasStatus_t invoke_Icamax(
                                int awHandle, int n, const cuComplex *x,
                                int incx,     int *result);
cublasStatus_t invoke_Izamax(
                                int awHandle, int n,
                                const cuDoubleComplex *x,    int incx,
                                int *result);
```

A.2.7.2 Description

These API functions are responsible for finding the index of the largest element of the vector x in single, double, complex single and complex double precisions respectively. In

the case of multiple largest elements, these functions return the smallest (first) index among them.

A.2.8 invoke_<t>gemv

A.2.8.1 Syntax

```
cublasStatus_t invoke_Sgemv (
                                int awHandle, cublasOperation_t trans,
                                int m, int n, const float *alpha,
                                const float *A,       int lda,
                                const float *x,       int incx,
                                const float *beta,    float *y,      int incy);
cublasStatus_t invoke_Dgemv (
                                int awHandle, cublasOperation_t trans,
                                int m, int n, const double *alpha,
                                const double *A,      int lda,
                                const double *x,      int incx,
                                const double *beta,   double *y,     int incy);
cublasStatus_t invoke_Cgemv (
                                int awHandle, cublasOperation_t trans,
                                int m, int n, const cuComplex *alpha,
                                const cuComplex *A,   int lda,
                                const cuComplex *x,   int incx,
                                const cuComplex *beta,       cuComplex *y,
                                int incy);
cublasStatus_t invoke_Zgemv (
                                int awHandle, cublasOperation_t trans,
                                int m, int n,
                                const cuDoubleComplex *alpha,
                                const cuDoubleComplex *A,
                                int lda,      const cuDoubleComplex *x,
                                int incx,     const cuDoubleComplex *beta,
                                cuDoubleComplex *y,   int incy);
```

A.2.8.2 Description

These API functions are responsible for performing matrix-vector multiplication between an m×n matrix A and vector x in single, double, single complex and double complex precisions respectively, adding the result to the original values of vector y. Constants alpha and beta are used for scaling vector x and the original values of result vector y. Matrix A is considered to be stored in column-major formatting, where lda indicates the

size of actual storage size of the leading dimension of matrix A, as this could sometimes be bigger than actual dimensions of A.The final result can be expressed mathematically as $y = \alpha OP(A)x + \beta y$. OP(A) indicates any necessicity for transposing the matrix A befor operation as (CUBLAS_OP_N, CUBLAS_OP_T, CUBLAS_OP_H) for non transpos, transpose or conjugate transpose(for complecx matrices) respectively. In cases where a compiler has a problem processing the CUBLAS_OP enum values in compute regeions, these shall be passed as integer numbers into the API.

A.2.9 invoke_<t>gemm

A.2.9.1 Syntax

```
cublasStatus_t invoke_Sgemm (
                            int awHandle, cublasOperation_t transa,
                            cublasOperation_t transb, int m, int n, int k,
                            const float *alpha, const float *A, int lda,
                            const float *B, int ldb, const float *beta,
                            float *C, int ldc);
cublasStatus_t invoke_Dgemm (
                            int awHandle, cublasOperation_t transa,
                            cublasOperation_t transb, int m, int n, int k,
                            const double *alpha, const double *A, int lda,
                            const double *B, int ldb, const double *beta,
                            double *C, int ldc);
cublasStatus_t invoke_Cgemm (
                            int awHandle, cublasOperation_t transa,
                            cublasOperation_t transb, int m, int n, int k,
                            const cuComplex *alpha, const cuComplex *A,
                            int lda, const cuComplex *B, int ldb,
                            const cuComplex *beta, cuComplex *C, int ldc);
cublasStatus_t invoke_Zgemm (
                            int awHandle, cublasOperation_t transa,
                            cublasOperation_t transb, int m, int n, int k,
                            const cuDoubleComplex *alpha,
                            const cuDoubleComplex *A, int lda,
                            const cuDoubleComplex *B, int ldb,
                            const cuDoubleComplex *beta, cuDoubleComplex *C,
                            int ldc);
```

A.2.9.2 Description

These API functions are responsible for performing matrix-matrix multiplication between an m×k matrix A and k×n matrix B in single, double, single complex and double complex precisions respectively. The result is added to the original values of result matrix C. Constants alpha and beta are used for scaling matrix A and the original values of result matrix C respectively. Matrices A, B, and C are considered to be stored in column-major formatting, where lda , ldb and ldc indicate the size of actual storage sizes of the leading dimensions of the matrices, as they could sometimes be bigger than actual dimensions of the matrices.The final result can be expressed mathematically as $y = \alpha OP(A)OP(B) + \beta C$. OP(A) and OP(B) indicate any necessicity for transposing the matrices A and B before the multiplication operations (CUBLAS_OP_N, CUBLAS_OP_T, CUBLAS_OP_H) for no transpos, transpose or conjugate transpose(for complec matrices) respectively. Notice that the dimenstions m ,n and k refer to the dimensions of the multiplication operands after applying OP(A) and OP(B). In cases where a compiler has a problem processing the enum values in compute regeions, these shall be passed as integer numbers into the API.

A.2.10invoke_<t>geam

A.2.10.1 Syntax

```
cublasStatus_t invoke_Sgeam(
                        int awHandle,  cublasOperation_t transa,
                        cublasOperation_t transb,    int m, int n,
                        const float *alpha,   const float *A,       int lda,
                        const float *beta ,   const float *B,       int ldb,
                        float *C,      int ldc);
cublasStatus_t invoke_Dgeam(
                        int awHandle,  cublasOperation_t transa,
                        cublasOperation_t transb,    int m, int n,
                        const double *alpha,  const double *A,      int lda,
                        const double *beta,   const double *B,
                        int ldb,       double *C,     int ldc);
```

```
cublasStatus_t invoke_Cgeam(
                          int awHandle,  cublasOperation_t transa,
                          cublasOperation_t transb,    int m, int n,
                          const cuComplex *alpha,      const cuComplex *A,
                          int lda,       const cuComplex *beta,
                          const cuComplex *B,   int ldb,       cuComplex *C,
                          int ldc);
cublasStatus_t invoke_Zgeam(
                          int awHandle,  cublasOperation_t transa,
                          cublasOperation_t transb,    int m, int n,
                          const cuDoubleComplex *alpha,
                          const cuDoubleComplex *A,    int lda,
                          const cuDoubleComplex *beta,
                          const cuDoubleComplex *B,    int ldb,
                          cuDoubleComplex *C,   int ldc);
```

A.2.10.2 Description

These API functions are responsible for performing matrix-matrix addition and/or transposition in single, double, single complex and double complex precisions respectively. Constants alpha and beta are used for scaling matrices A and B. Matrices A, B and C are considered to be stored in column-major formatting, where lda , ldb, and ldc indicate the size of actual storage sizes of the leading dimensions of the matrices , as they could sometimes be bigger than actual dimensions of the matrices. The final result can be expressed mathematically as $C = \alpha OP(A) + \beta OP(B)$. However, these functions support in-place operations where C = A or C = B. in such cases, the operation can be expressed mathematically as $C = \alpha C + \beta OP(B)$ and $C = \alpha OP(A) + \beta C$ respectively. OP(A) and OP(B) indicate any necessity for transposing the matrices A and B before the multiplication operations (CUBLAS_OP_N, CUBLAS_OP_T, CUBLAS_OP_H) for no transpose, transpose or conjugate transpose (for complex matrices) respectively. Notice that the dimensions m and n refer to the dimensions of the addition operands after applying OP(A) and OP(B). In cases where a compiler has a problem processing the enum values in compute regions, these shall be passed as integer numbers into the API.

References

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr Dobb's J.*, vol. 30, no. 3, pp. 202–210, 2005.

[2] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[3] J. S. Vetter *et al.*, "Keeneland: Bringing heterogeneous gpu computing to the computational science community," *Comput. Sci. Eng.*, vol. 13, no. 5, pp. 90–95, 2011.

[4] J. Dongarra, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, p. 1094342010391989, 2011.

[5] I. Buck *et al.*, "Brook for GPUs: stream computing on graphics hardware," presented at the ACM Transactions on Graphics (TOG), 2004, vol. 23, pp. 777–786.

[6] S. Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and computation transformations for Brook streaming applications on multiprocessors," presented at the Code Generation and Optimization, 2006. CGO 2006. International Symposium on, 2006, p. 12–pp.

[7] M. Peercy, M. Segal, and D. Gerstmann, "A Performance-oriented Data Parallel Virtual Machine for GPUs," in *ACM SIGGRAPH 2006 Sketches*, New York, NY, USA, 2006.

[8] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[9] Khronos OpenCL Working Group, "The opencl specification," *version*, vol. 1, no. 29, p. 8, 2008.

[10] A. R. B. OpenMP, "OpenMP application program interface, v. 3.0," *OpenMP Archit. Rev. Board*, 2008.

[11] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs, SC'10: Proceedings of the 2010 ACM," in *IEEE conference on Supercomputing*, 2010.

[12] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 78–90, 2011.

[13] OpenACC Working Group, "The OpenACC Application Programming Interface," 2011.

[14] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, New York, NY, USA, 2010, pp. 43–50.

[15] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "OpenMP for Accelerators," in *OpenMP in the Petascale Era*, B. M. Chapman, W. D. Gropp, K. Kumaran, and M. S. Müller, Eds. Springer Berlin Heidelberg, 2011, pp. 108–121.

[16] A. Leung *et al.*, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," presented at the Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 51–61.

[17] E. Wang *et al.*, "Intel Math Kernel Library," in *High-Performance Computing on the Intel® Xeon Phi™*, Springer International Publishing, 2014, pp. 167–188.

[18] C. Nvidia, “Cublas library,” *NVIDIA Corp. St. Clara Calif.*, vol. 15, 2008.
[19] C. Nvidia, “CUFFT library,” 2010.
[20] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.
[21] Q. Hou, K. Zhou, and B. Guo, “BSGP: Bulk-synchronous GPU Programming,” in *ACM SIGGRAPH 2008 Papers*, New York, NY, USA, 2008, p. 19:1–19:12.
[22] B. Leback, D. Miles, and M. Wolfe, “Tesla vs. Xeon Phi vs. Radeon A Compiler Writer’s Perspective,” *Tech. News Portland Group*, 2013.
[23] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2008, pp. 73–82.
[24] T. D. Han and T. S. Abdelrahman, “*hi*CUDA: a high-level directive-based language for GPU programming,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, New York, NY, USA, 2009, pp. 52–61.
[25] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, “accULL: An OpenACC Implementation with CUDA and OpenCL Support,” in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Springer Berlin Heidelberg, 2012, pp. 871–882.
[26] D. Luebke *et al.*, “GPGPU: General-purpose Computation on Graphics Hardware,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2006.
[27] S. Lee and J. S. Vetter, “Early evaluation of directive-based GPU programming models for productive exascale computing,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Los Alamitos, CA, USA, 2012, p. 23:1–23:11.
[28] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu, “CUDA-Lite: Reducing GPU Programming Complexity,” in *Languages and Compilers for Parallel Computing*, J. N. Amaral, Ed. Springer Berlin Heidelberg, 2008, pp. 1–15.
[29] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.
[30] S.-I. Lee, T. A. Johnson, and R. Eigenmann, “Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation,” in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed. Springer Berlin Heidelberg, 2004, pp. 539–553.
[31] R. Dolbeau, S. Bihan, and F. Bodin, “HMPP: A hybrid multi-core parallel programming environment,” presented at the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), 2007.
[32] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming*. Newnes, 2013.
[33] R. Reyes, I. Lopez, J. J. Fumero, and F. de Sande, “Directive-based Programming for GPUs: A Comparative Study,” in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International*

Conference on Embedded Software and Systems (HPCC-ICESS), 2012, pp. 410–417.

[34] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2008, pp. 101–113.

[35] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame, "A Programming Language Interface to Describe Transformations and Code Generation," in *Languages and Compilers for Parallel Computing*, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Springer Berlin Heidelberg, 2010, pp. 136–150.

[36] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," *U South. Calif. Tech Rep*, pp. 08-897, 2008.

[37] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting Polyhedral Loop Transformations to Work," in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed. Springer Berlin Heidelberg, 2004, pp. 209–225.

[38] M. M. Z. M. Khan, "Autotuning, code generation and optimizing compiler technology for GPUs," 2012.

[39] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop Transformation Recipes for Code Generation and Auto-Tuning," in *Languages and Compilers for Parallel Computing*, G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds. Springer Berlin Heidelberg, 2010, pp. 50–64.

[40] S. Donadio *et al.*, "A Language for the Compact Representation of Multiple Program Versions," in *Languages and Compilers for Parallel Computing*, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds. Springer Berlin Heidelberg, 2006, pp. 136–151.

[41] C. Chen, J. Chame, and M. Hall, "Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy," in *International Symposium on Code Generation and Optimization, 2005. CGO 2005*, 2005, pp. 111–122.

[42] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, 2008, pp. 1–11.

[43] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A Script-based Autotuning Compiler System to Generate High-performance CUDA Code," *ACM Trans Arch. Code Optim*, vol. 9, no. 4, p. 31:1–31:25, Jan. 2013.

[44] Y. Zhang and F. Mueller, "Hidp: A hierarchical data parallel language," in *2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–11.

[45] L. Chen *et al.*, "Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation," in *Languages and Compilers for Parallel Computing*, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Springer Berlin Heidelberg, 2010, pp. 151–165.

[46] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, 2005.

[47] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzaran, and D. Padua, "Performance Portability with the Chapel Language," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 582–594.
[48] A. Sidelnik, B. L. Chamberlain, M. J. Garzaran, and D. Padua, "Using the High Productivity Language Chapel to Target GPGPU Architectures," Apr. 2011.
[49] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.
[50] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, 1968, pp. 88–98.
[51] J. Darlington *et al.*, "Parallel programming using skeleton functions," in *PARLE '93 Parallel Architectures and Languages Europe*, A. Bode, M. Reeve, and G. Wolf, Eds. Springer Berlin Heidelberg, 1993, pp. 146–160.
[52] J. Liu, "Compiler Optimizations for Simd/Gpu/Multicore Architectures," Pennsylvania State University, University Park, PA, USA, 2013.
[53] A. Cohen, T. Grosser, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles to Reconcile Parallelism and Locality, avoiding Divergence and Load Imbalance," presented at the GPGPU 6 - Sixth Workshop on General Purpose Processing Using GPUs, 2013.
[54] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral Parallel Code Generation for CUDA," *ACM Trans Arch. Code Optim*, vol. 9, no. 4, p. 54:1–54:23, Jan. 2013.
[55] M. Elteir, H. Lin, and W. Feng, "Performance Characterization and Optimization of Atomic Operations on AMD GPUs," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, 2011, pp. 234–243.
[56] R. Duarte, R. Sendag, and F. J. Vetter, "On the performance and energy-efficiency of multi-core SIMD CPUs and CUDA-enabled GPUs," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 174–184.
[57] Y. YANG and H. ZHOU, "DEVELOPING A HIGH PERFORMANCE GPGPU COMPILER USING CETUS," 2011.
[58] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2010, pp. 86–97.
[59] A. Qasem, "Architectural Considerations for Compiler-guided Unroll-and-Jam of CUDA Kernels," *Am. J. Comput. Archit.*, vol. 1, no. 2, pp. 12–20, 2012.
[60] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 1991, pp. 63–74.
[61] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation," in

Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, 2012, pp. 107–118.
[62] A. Abdelfattah, J. Dongarra, D. Keyes, and H. Ltaief, "Optimizing Memory-Bound SYMV Kernel on GPU Hardware Accelerators," in *High Performance Computing for Computational Science-VECPAR 2012*, Springer, 2013, pp. 72–79.
[63] J. C. Wileden and A. Kaplan, "Software Interoperability: Principles and Practice," in *Proceedings of the 21st International Conference on Software Engineering*, New York, NY, USA, 1999, pp. 675–676.
[64] L. S. Blackford *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, 2002.
[65] J. Dongarra, "Preface: basic linear algebra subprograms technical (blast) forum standard," *Int. J. High Perform. Comput. Appl.*, vol. 16, no. 2, pp. 115–115, 2002.
[66] S. Jones, "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation S*, 2012, vol. 338.
[67] S. C. Rennich, D. Stosic, and T. A. Davis, "Accelerating sparse cholesky factorization on GPUs," presented at the Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms, 2014, pp. 9–16.
[68] E. Anderson *et al.*, "LAPACK: A Portable Linear Algebra Library for High-performance Computers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Los Alamitos, CA, USA, 1990, pp. 2–11.
[69] L. Grillo, F. de Sande, and R. Reyes, "Performance Evaluation of OpenACC Compilers," in *2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2014, pp. 656–663.
[70] nVidia, "CuBLAS CUDA TOOLKIT DOCUMENTATION," *NVIDIA Developer*, 01-Sep-2015. [Online]. Available: https://developer.nvidia.com/cublas. [Accessed: 14-Jan-2016].
[71] nVidia, "CUDA Toolkit Documentation," 01-Sep-2015. [Online]. Available: http://docs.nvidia.com/cuda/index.html. [Accessed: 15-Jan-2016].
[72] J. R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*. Carnegie-Mellon University. Department of Computer Science, 1994.
[73] K. Rupp, J. Weinbub, A. Jüngel, and T. Grasser, "Pipelined Iterative Solvers with Kernel Fusion for Graphics Processing Units," *ArXiv14104054 Cs*, Oct. 2014.
[74] R. Barrett *et al.*, *Templates for the solution of linear systems: building blocks for iterative methods*, vol. 43. Siam, 1994.
[75] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, vol. 49. National Bureau of Standards Washington, DC, 1952.
[76] M. Reddy, *API Design for C++*. Elsevier, 2011.
[77] H. Sutter, *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison-Wesley Professional, 2000.
[78] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.
[79] A. ul Hasan Khan, M. Al-Mouhamed, and L. A. Firdaus, "Evaluation of global synchronization for iterative algebra algorithms on many-core," in *2015 16th*

IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015, pp. 1–6.

[80] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with Relaxed Synchronization," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, New York, NY, USA, 2012, pp. 41–50.

[81] D. A. Rivera-Polanco, *Collective Communication and Barrier Synchronization on NVIDIA GPU*. University of Kentucky Libraries, 2009.

[82] G. Barlas, *Multicore and GPU Programming: An Integrated Approach*. Elsevier, 2014.

[83] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," *BMC Res. Notes*, vol. 2, p. 73, 2009.

[84] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel Iterative Algorithms: From Sequential to Grid Computing (Chapman & Hall/Crc Numerical Analy & Scient Comp. Series)*. Chapman & Hall/CRC, 2007.

[85] Z. Zhang, Q. Miao, and Y. Wang, "CUDA-based Jacobi's iterative method," in *Computer Science-Technology and Applications, 2009. IFCSTA'09. International Forum on*, 2009, vol. 1, pp. 259–262.

[86] W. Feng and S. Xiao, "To GPU synchronize or not GPU synchronize?," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2010, pp. 3801–3804.

Index

Vitae

Name : Anas Ali Almousa

Nationality : Syrian

Date of Birth : 2/22/1982

Email : Anas.Ali@gmail.com

Address : Eastern province – Saudi Arabia

Academic Background : Anas had got his BSc and MSc degrees in Computer Engineering from Jordan University of Science and technology. During his PhD at King Fahd University of Petroleum and Minerals, he worked on General Purpose Programming of Graphics Processing Units. He presented a seminar titled “Enablement and use of device routines from within OpenACC compute regions “at the third annual workshop for accelerating scientific Applications using GPUs in King Abdullah University of Science and Technology.