ACCELERATION OF CFD AND DATA ANALYSIS USING GRAPHICS PROCESSORS

A Dissertation Outline Presented

by

ALI KHAJEH-SAEED

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2012

Mechanical and Industrial Engineering

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

ACCELERATION OF CFD AND DATA ANALYSIS USING GRAPHICS PROCESSORS

A Dissertation Outline Presented

by

ALI KHAJEH-SAEED

Approved as to style and content by:

J. Blair Perot, Chair

Stephen de Bruyn Kops, Member

Rui Wang, Member

Hans Johnston, Member

Donald L. Fisher, Department Head
Mechanical and Industrial Engineering

ACKNOWLEDGMENTS

This thesis would not have been possible without the aid and support of countless people over the past four years. I must first express my gratitude towards my advisor, Professor J. Blair Perot. I would also like to thank the members of my graduate committee, Professor Stephen de Bruyn Kops, Professor Rui Wang and Professor Hans Johnston for their guidance and suggestions.

Special thanks to my friend Michael B Martell JR for his invaluable assistance with all things Linux, CFD and Windows. I would like to thank my friend Timothy McGuiness for his invaluable help with CUDA.

It is a pleasure to thank Shivasubramanian Gopalakrishnan, Sandeep Menon, Kshitij Neroorkar, Dnyanesh Digraskar, Nat Trask, Michael Colarossi, Thomas Furlong, Kyle Mooney, Chris Zusi, Brad Shields, Maija Benitz and Saba Almalkie for making the lab a fun place to work.

The author would also like to thank the Department of Defense and Oak Ridge National Lab (ORNL) for their support. Some of the development work occurred on the NSF Teragrid/XSEDE supercomputer, Lincoln and Forge, located at National Center for Supercomputing Applications (NCSA) and Keeneland supercomputer, located at National Institute for Computational Science (NICS).

The author would like thank the NVIDIA and AMD for generous donation of 2 Tesla C2070s and 2 ATI FirePro V7800s graphic cards.

ABSTRACT

ACCELERATION OF CFD AND DATA ANALYSIS USING GRAPHICS PROCESSORS

FEBRUARY 2012

ALI KHAJEH-SAEED

B.Sc., SHARIF UNIVERSITY OF TECHNOLOGY

M.Sc., SHARIF UNIVERSITY OF TECHNOLOGY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Blair Perot

Graphics processing units function well as high performance computing devices for scientific computing. The non-standard processor architecture and high memory bandwidth allow graphics processing units (GPUs) to provide some of the best performance in terms of FLOPS per dollar. Recently these capabilities became accessible for general purpose computations with the CUDA programming environment on NVIDIA GPUs and ATI Stream Computing environment on ATI GPUs. Many applications in computational science are constrained by memory access speeds and can be accelerated significantly by using GPUs as the compute engine. Using graphics processing units as a compute engine gives the personal desktop computer a processing capacity that competes with supercomputers. Graphics Processing Units represent an energy efficient architecture for high performance computing in flow simulations and many other fields. This document reviews the graphic processing unit and its features and limitations.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Central Processing Unit (CPU)

Intel and AMD produce microprocessors based on several Central Processing Unit (CPU) cores that are efficient and cost-effective. Generally the speed of these microprocessors is roughly doubled every three years. This rate of speed increase occurred before 2003 and with increasing the power of the CPU, the number of calculations per second increased. But there is a limit for increasing the power (overheating issues), so Intel and AMD decided to increase the number of cores in the microprocessors rather than the clock frequency. Before multi-core microprocessors most programs ran sequentially. The switch to multicore processors has had a tremendous effect on the software and hardware developer communities. But it has not brought speed increases to high performance computing (HPC), because speed is now bottlenecked by the memory subsystem not the processor [13].

1.2 Graphics Processing Unit (GPU)

After 2003, a new class of many-core processors called Graphics Processing Units (GPUs), were introduced to the scientific community. AMD (ATI) and NVIDIA replaced multi-core microprocessors with many-core processors. This phenomenon changed the performance from giga-bits per seconds to tera-bits per second (see Figure 1.1). Every six months they introduce new hardware with the performance increased over the previous generation of the hardware [2]. As of 2010, the ratio of peak (and actual) floating-point operation per second between GPUs and CPUs is

close to 10. New GPUs and CPUs have theoretical speeds of 1350 gigaflops and 140 gigaflops respectively. NVIDIA introduced new software to the scientific community that allows the G80 microprocessors to be easily programmed. In June 2008, NVIDIA introduced a new series of microprocessors (Tesla 10-series GPUs) to the user community that support double precision for the first time, which delivers more than 1,000 peak gigaflops of single precision and close to 100 peak gigaflops of double precision performance. In September 2009, NVIDIA introduced the next-generation GPU architecture codenamed "Fermi", the Tesla 20-series family, which delivers more than 1,300 gigaflops of single precision and approximately 515 gigaflops of double precision performance, which is 5 times faster than the previous generation of their GPU architecture [2, 1, 14].

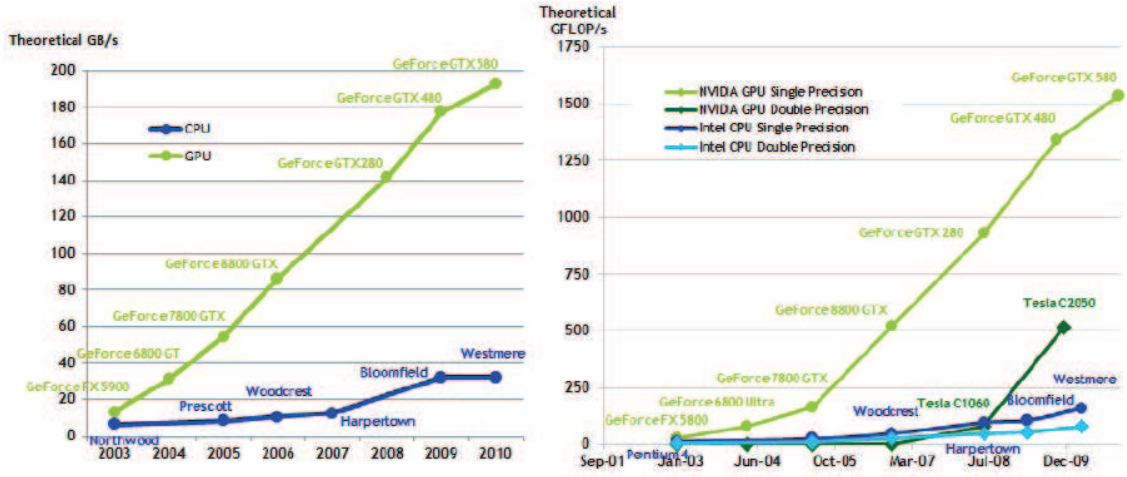


Figure 1.1. FLOPS and memory bandwidth for the CPU and GPU [2]

The main reason for such a large bandwidth and GFLOPs gap between GPU and CPU microprocessors is in the differences in the fundamental design strategies between the two types of microprocessors, as illustrated in Figure 1.2 [2].

The CPU is usually designed and optimized for sequential code performance and not for parallel tasks. This philosophy makes use of a complicated control logic unit to let sequential instructions execute in parallel while keeping the appearance of a

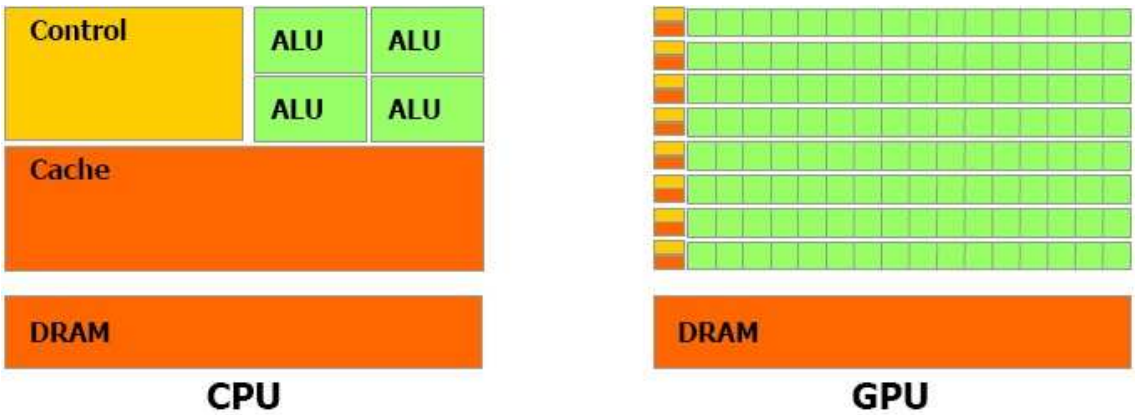


Figure 1.2. The GPU devotes more transistors to data processing [2]

sequential program. In contrast, the GPU is designed in a way that more transistors are dedicated to data processing and fewer for the flow control and data caching. In addition, the GPU designers are aware of the memory wall and routinely use newer and faster memory in their designs.

Although the GPU is well suited for scientific computing, it is still important to understand the hardware and its limitations if high performance is desired. The main strategy for GPU design is to optimize for the execution of an enormous number of threads at the same time without any overhead. The GPU takes advantage of a large number of execution threads to find work to do when some of the threads are idle or waiting for memory latency. Memory bandwidth is another important parameter in the GPU. In late 2006, the G80 delivered about 80 gigabytes per second (GB/S) into the main DRAM (DDR3) and in 2010, the GPU bandwidth reached to 180 GB/s (DDR5). A small amount of cache memory (shared memory) is provided to help control the bandwidth requirements of these applications. But this fast memory (unlike the CPU cache) has to be managed explicitly. So with this fast memory multiple threads can access the data with a low latency and do not need to all go to the GPU memory. In newer generations of GPUs, small L1 and L2 caches (like a CPU) are also added to hide long memory latency [2].

It is clear that the GPU is designed as a data computing engine and it will not perform well on some tasks that CPUs are designed to perform well on like sequential tasks with small data streams. Fortunately, every GPU is hosted by a computer that has a CPU, so it isn't necessary to use a GPU if the task is not well suited to that hardware. The GPU is a supplement not a substitute for the CPU. In fact, most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numeric intensive parts on the GPU for better performance. Every communication must be through the relatively slow PCI-Express bus. Although hardware developers increased this communication speed by changing PCI-Express 1 ($\times 8$) to new PCI-Express 2 ($\times 16$), this update is still far from the main memory speed (5 GB/s vs. 180 GB/s). PCI-Express 3 (maximum theoretical bandwidth of 16 GB/s) is expected to available last quarter of 2011 [15].

NVIDIA introduced first version of the GPUDirect technology in June 2010. The first GPUDirect technology accelerated communication with network and storage devices from Mellanox and QLogic [3]. In the first version in order to used pinned memory (zerocopy), there is a copy from system memory (pinned memory) to another memory (both memories in the same system). Figure 1.3 shows how the communication was done with and without GPUDirect technology.

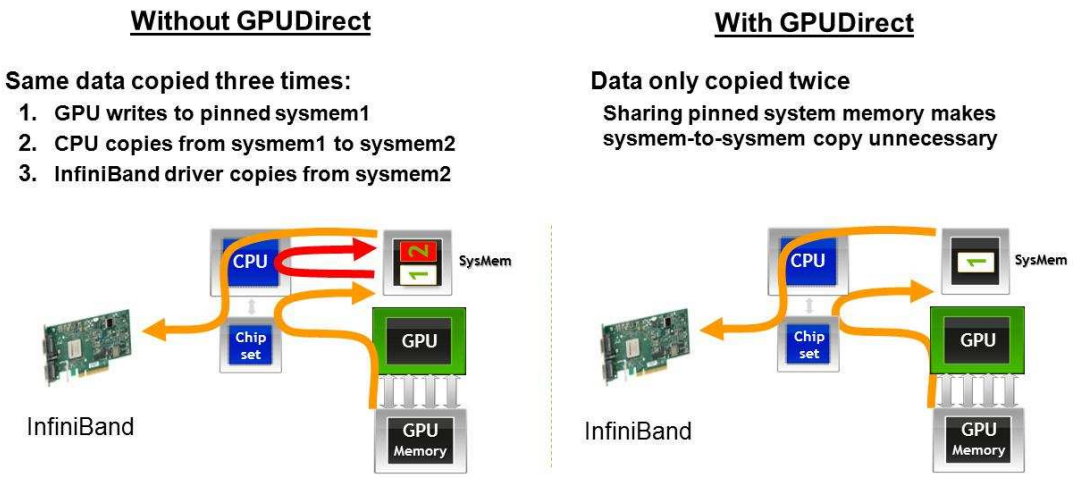


Figure 1.3. Acceleration of communication with network and storage devices with NVIDIA GPUDirect *I* [3]

The latest generation of GPUs form NVIDIA support *GPUDirect II* Technology. In the second generation of the NVIDIA *GPUDirect*, all GPUs in the same node or motherboard can access each others memories via PCI-e without going to CPU memory. Also it is possible to copy data between GPUs without going to the CPU. This technology eliminates system memory allocation and copy overhead. Figure 1.4 shows how the communication was done with two GPUs in the same node or motherboard that connected with PCI-e using *GPUDirect* technology.

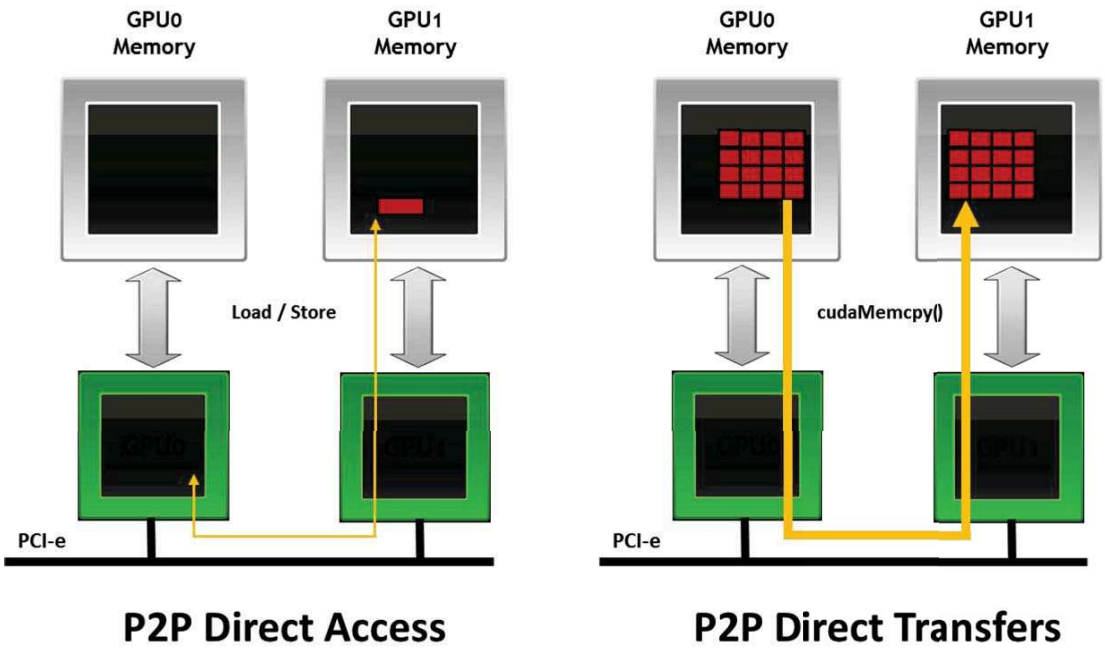


Figure 1.4. Acceleration of communication between GPUs in the same node or motherboard with NVIDIA *GPUDirect II* [3]

1.3 Architecture of a Modern GPU

Figure 1.5 shows the new generation of GPUs (Fermi based GPUs). It has almost 3.0 billion transistors, with 16 multiprocessors each of them with 32 cores (512 total). The Fermi multiprocessor core executes a single or double floating point or integer instruction per clock for a thread [16].

The Fermi based GPU has six 64-bit memory partitions, for a 384-bit memory interface, with 3 or 6 GB of GDDR5 DRAM memory based on the GPU's model. The GPU is connected to the CPU via a PCI-Express slot. The GigaThread global scheduler unit distributes the thread blocks to Stream Multiprocessor (SM) thread schedulers. Each stream multiprocessor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Previous generations of the GPUs used IEEE 754-1985 floating point arithmetic. But the Fermi architecture applies the new

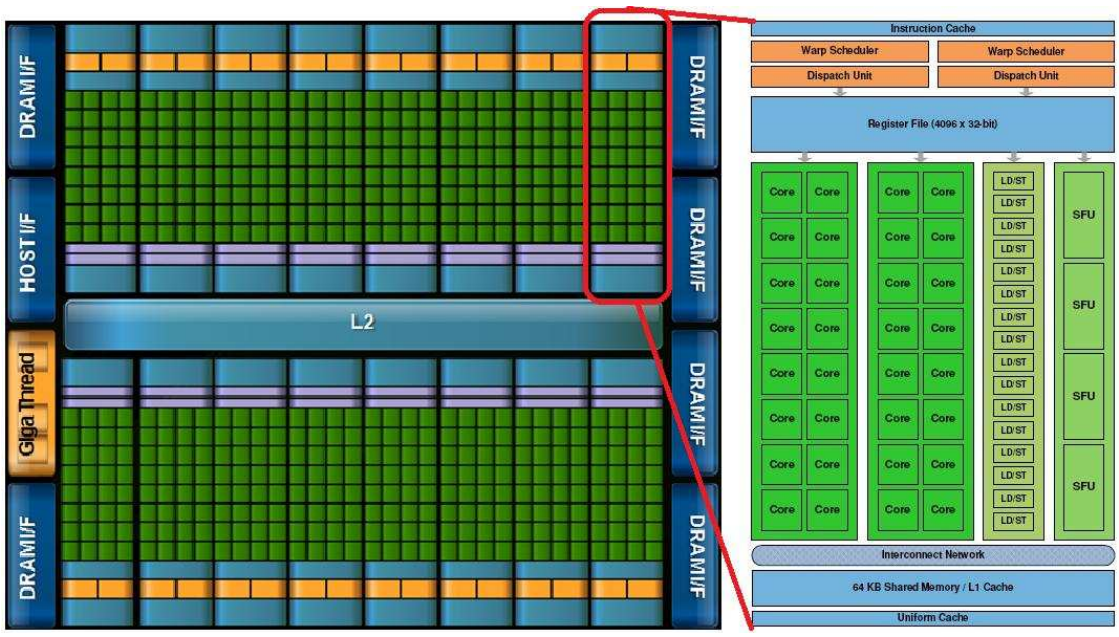


Figure 1.5. Fermi architecture with 16 SM of 32 cores each and a block diagram of a single SM [4]

IEEE 754-2008 floating-point standard, using the fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA has better performance over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding. Because FMA has just a single rounding step this makes it more accurate than performing the operations separately in comparison with MAD. Four Special Function Units (SFUs) (hardware) in each multiprocessor execute instructions such as cosine, sine, reciprocal, logarithm and square root. Each SFU executes one instruction per thread, per clock; a single warp (group of 32 parallel threads) executes over eight clocks. Double precision arithmetic is necessary for most scientific applications such as linear algebra, numerical simulation, and quantum physics and chemistry. The Fermi architecture has been specifically designed to offer much better performance in double precision; up to 16 double precision FMA operations can be performed per SM, per clock, a enormous improvement over the

previous architecture (GT200) [4]. But in practice this has only a moderate affect of scientific computations which are invariably memory and not computation limited.

Table 1.1 gives the features and technical specifications associated to each GPU architecture [1].

Table 1.1. Features and technical specifications for different GPU architectures [1]

GPU	**G80 (2006)**	**GT200 (2008)**	**Fermi (2010)**
Transistors	681 Million	1.4 Billion	3.0 Billion
Total number of cores	128	240	512
Double Precision (ops/clock)	None	30 FMA	256 FMA
Single Precision (ops/clock)	128 MAD	240 MAD	512 FMA
Warp schedulers/SM	1	1	2
Special Function Units/SM	2	2	4
Shared Memory/SM (Configurable)	16 KB	16 KB	48 KB or 16 KB
L1 Cache/SM (Configurable)	None	None	16 KB or 48 KB
L2 Cache/SM	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit
Max number of threads/block	512	512	1024
Max number of threads/SM	768	1024	1536
Number of 32-bit registers/SM	8 K	16 K	32 K

1.4 Compute Unified Device Architecture (CUDA)

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture that uses NVIDIA GPUs as a parallel compute engine to solve many complicated computational problems in a more efficient and faster way than on a CPU [17]. CUDA enables this high performance via standard application programming interfaces (APIs) like DirectCompute, OpenCL, and high level programming environments such as C/C++, Java, Python, Fortran (the Portland Group) and the Microsoft.NET Framework. CUDA provides both a low and high level API. The initial CUDA SDK was published on 15 February 2007, for Microsoft Windows and Linux operating systems. Mac OS X support was later added in version 2.0 on February 14, 2008. CUDA supports all NVIDIA GPUs from the G80 series onwards, includ-

ing GeForce GTX, Quadro FX and the high performance Tesla GPUs [17]. NVIDIA mentions that programs written for the GeForce 8 series will also work without any modification on all future NVIDIA video cards, due to binary compatibility standard (backward compatibility). CUDA provides developers access to the memories of the GPU and native instruction set. On the other hand, CUDA makes the NVIDIA GPUs effectively become open architectures like CPUs. Because the GPUs have the parallel many-core architecture, each multiprocessor can execute thousands of threads simultaneously and the GPU can deliver high performance computations to the desktop personal computers. With the CUDA architecture and tools, software developers are achieving tremendous speedups in fields such as computational fluid dynamics (CFD), molecular dynamics (MD), finance, signal processing, computational biology, medical imaging and natural resource exploration, and creating breakthrough applications in areas such as image recognition, ray tracing, real-time HD video playback and encoding [2].

1.5 Computational Fluid Dynamics (CFDs) and Parallel Processing

The development of computational fluid dynamics (CFD) started first with the appearance of the digital computer in early 1950s. Finite difference methods (FDMs) and finite element methods (FEMs), which are the basic tools used in the solution of partial differential equations (PDEs) in general and especially in CFD, have different histories. At the Royal Society of London, Richardson implemented the first FDM solution to analyze the stress of a masonry dam in 1910. In contrast, the first FEM work was published in the Aeronautical Science Journal by Turner Clough, Martin and Topp to analysis aircraft stress in 1956. After that time, both methods have been developed and modified dramatically for use in fluid dynamics, heat transfer, combustion and other related areas. There are some benefits accruing from the com-

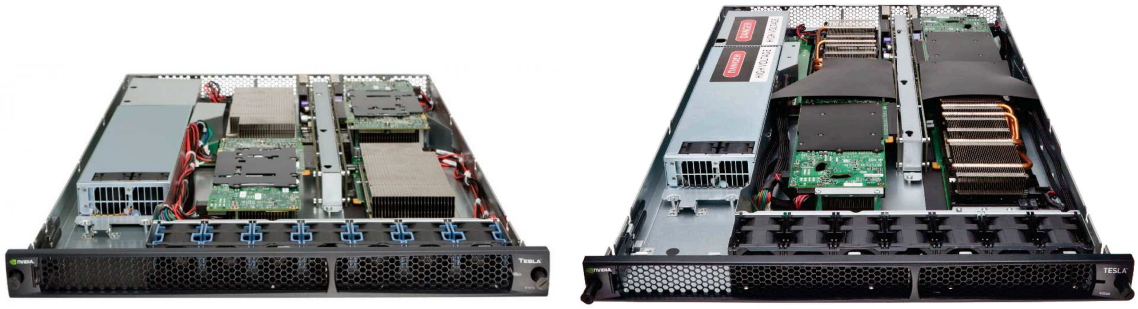
bination of both FDM and FEM. The Finite volume method (FVM) is a combination of the both FDM and FEM methods and has become popular in recent years because of its conservative form and easy applicability to unstructured meshes [18].

Scientific programs, especially CFD codes, are usually run on computers that have large memories and high processing speeds. In addition, massive data storage systems must be provided to store and analyze the computed results and it is necessary to have methods to transmit and examine the enormous amounts of data and the computed results [18].

Supercomputers were presented in the 1960s and designed for the first time by Seymour Cray at Control Data Corporation (CDC), which became available to the market in the 1970s [19]. Cray supercomputers were the top ranked supercomputers between 1985 and 1990. Nowadays, most supercomputers are essentially PC clusters that are assembled by well-known companies such as Dell, SuperMicro, Cray, IBM and Hewlett-Packard [19]. As of November 2010, the Cray Jaguar is the fastest CPU based supercomputer and second in the world according to TOP500 [20]. Jaguar is a petascale supercomputer built by Cray at Oak Ridge National Laboratory in Oak Ridge, Tennessee [21]. It has a max (maximal LINPACK performance achieved) and peak (theoretical peak performance) performance of almost 1759 and 2331 teraflops respectively. Jaguar has 224,162 Opteron processor cores, and a version of Linux called Cray Linux Environment installed on it [21].

GPUs that are essentially used for graphics rendering have become massively-parallel "co-processors" to the CPUs nowadays. The new generation of NVIDIA GPUs beat Intel and AMD CPUs on single and double floating point performance by a factor of six and four respectively and memory bandwidth by a factor of roughly five. Nowadays small desktop supercomputers with GPUs can deliver high performance at the price of conventional workstations.

Not only are GPUs cost-effective multi-core accelerators, but they also have reduced space, power, and cooling demands. In support of this goal, NVIDIA has begun producing commercially available Tesla GPU accelerators for use in scientific supercomputer clusters. The Tesla GPUs are available in standard single GPU with single video output for Fermi architecture and without any video outputs for 200 series based Tesla, or in 1U rack mount cases with four GPU devices inside the unit. Each 1U rack mount has its own power and cooling system [5, 6]. Figure 1.6 shows a Tesla S1070 and Tesla S2070 that is equipped with four GPUs in a single 1U rack mount.



(a) Tesla S1070 (b) Tesla S2070

Figure 1.6. NVIDIA Tesla S1070 [5] and Tesla S2070 [6] includes four GT200 and GF100 GPUs respectively in the single 1U rack mount

Several GPU clusters have been set up in the last decade. However, most GPU clusters were used as visualization systems and not for scientific computation. Only in last five years, scientific computation GPU clusters have been deployed. The first two GPU clusters were a 160-node GPU cluster at Los Alamos National Lab (LANL) [22] and a 16-node GPU cluster at National Center for Supercomputing Applications (NCSA) [23]. NVIDIA QuadroPlex GPUs are installed on both of the clusters. But NVIDIA QuadroPlex GPUs are suitable for visualization applications. The main reason for such installations is for experimentation. GPU clusters specifically built for scientific computation are still rare. At NCSA two GPU clusters with the NVIDIA

Tesla S1070 Computing System have been built: a 96 1U rack mount Tesla S1070 (384 total GPUs) cluster "Lincoln" [24] and an experimental 16 1U rack mounts Tesla S1070 (64 total GPUs) cluster "AC" [25] in 2009. There are three important components that should be matched in a GPU cluster: host nodes (CPU nodes), GPUs, and the interconnect between host and the GPUs. Considering the GPUs are going to perform the majority of the calculations, main memory (RAM), PCI-e bus, and network interconnect performance characteristics need to be matched with the GPU performance in order to get high performance. Eminently, high performance GPUs, like the NVIDIA Tesla, desire full-bandwidth PCI-e Gen 2 $\times$16 slots that do not reduce to $\times$8 speeds even when multiple GPUs are used in a single PCI-e slot. Additionally, InfiniBand QDR interconnect is highly desired to match the GPU-to-CPU bandwidth when multi-GPUs are used for computation. A single CPU core per GPU may be desirable to simplify the development of MPI-based applications because CUDA can stall a CPU core. Like CPU parallel programs, there is an option to decide between MPI and OpenMP or pthreads. OpenMP or pthreads only can execute programs on the single node (same motherboard). For running the programs on clusters, MPI is necessary [26]. As of November 2010, the first, third and fourth fastest supercomputers are GPU based supercomputers.

China's new Tianhe-1A (Milky Way in English) is the fastest supercomputer in the world according to TOP500 with a max and peak performance of almost 2,566 and 4,701 teraflops, respectively [20]. Tianhe-1A overtook the Jaguar supercomputer with 2.566 petaFLOPS. Also Tianhe-1A consumes 4,040 KW and Jaguar consumes 6,950.60 KW (40% less than Jaguar). Tianhe-1A has 186,368 Xeon X5670 processor cores and 7168 NVIDIA Tesla M2050 GPUs. Each Tesla M2050 is a single GPU with 3GB of memory and with passive heatsink cooled by host system airflow. The Tianhe-1A had a cost $88 million to build and requires roughly $20 million for annual energy and operating costs [27].

China's Nebulae Supercomputer is built from a Dawning TC3600 Blade system. Nebulae has 55,680 Xeon X5650 processors cores and 4,640 Tesla C2050 GPUs (Fermi architecture) [28]. Nebulae is now the second fastest GPU system worldwide in theoretical peak performance at 2.98 PFlop/s and 1.271 PFlop/s with a Linpack performance. Nebulae, which is located at the newly built National Supercomputing Centre in Shenzhen, holds the number 3 spot on the TOP500 list of supercomputers behind Tianhe-1A (2.566 petaFLOPS) and Jaguar (1.75 petaFLOPS) [28].

1.6 Supercomputers and Specifications

In this project, four different machines were used for computations, Orion, Lincoln, Forge and Keeneland. Orion and Forge hold AMD CPUs but Lincoln and Keeneland hold Intel CPUs. In the results section, it is shown that Intel and AMD work in different ways. Also Orion, Forge and Keeneland have equipped with new Tesla 20 series (Fermi) GPUs. But Lincoln uses Tesla 10 series. Below each supercomputer specifications are explained in details.

1.6.1 Orion In-House Supercomputer

Orion contains an AMD quad-core Phenom II X4 CPU, operating at 3.2 GHz, with 4×512 KB of L2 cache, 6 MB of L3 cache and 8 GB of RAM. In terms of GPUs, Orion contains four NVIDIA GTX 295 cards (occupying four PCI-e ×16 slots) which each come as two GPU cards sharing a single PCI-e slot (so 8 GPUs in total in Orion). When it is refered to a single GTX 295 GPU, it is referred to one of the processors located on the single 295 card. Each GPU has 240 cores and a memory bandwidth of 111.9 GB/s. Also the first and second GPUs are replaced with GTX 480 and Tesla C2070 cards in order to run some cases with these new GPUs. All code was written in C++ with NVIDIA's CUDA language extensions for the GPU. Results on Orion were compiled using Microsoft Visual Studio 2005 (VS 8) under Windows XP Professional

x64. The bulk of NVIDIA SDK examples use this configuration. Orion employs six fans for cooling purposes. Figure 1.7 shows Orion's configuration and fan locations.

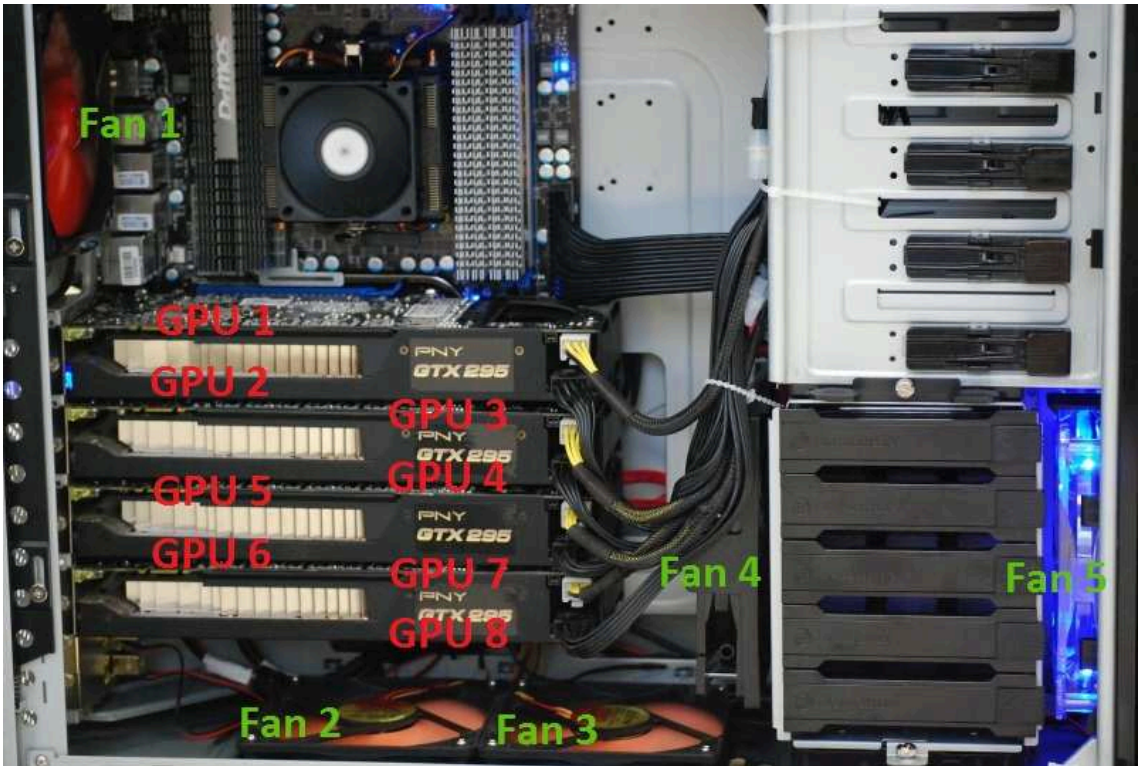


Figure 1.7. Orion in-house supercomputer with AMD quad-core Phenom II X4 CPU and four GTX 295 cards

1.6.2 Lincoln Supercomputer

Lincoln is a Teragrid/XSEDE GPU cluster located at NCSA (see figure 1.8). Lincoln has 96 Tesla S1070 servers (384 GPUs). Lincoln's 192 servers each hold two Intel 64 (Harpertown) 2.33 GHz dual socket quad-core processors with 2×6 MB L2 cache and 2 GB of RAM per core. Each server is connected to 2 Tesla processors via PCI-e Gen2 X8 slots. The Lincoln results were compiled using Red Hat Enterprise Linux 4 (Linux 2.6.19) and the gcc compiler [24]. Lincoln has one processor (4 cores) for each GPU.

Figure 1.8. Lincoln Teragrid/XSEDE GPU cluster with 384 Tesla 10 series GPUs (from [7])

1.6.3 Forge Supercomputer

Lincoln was replaced with Forge. Forge is a new Teragrid/XSEDE GPU cluster located at NCSA (see figure 1.9). Forge has 288 Tesla M2070 NVIDIA Fermi GPUs that each comes with 6 GB DDR5 memory. Forge's 36 servers each hold two AMD Opteron Magny-Cours 6136 with 2.4 GHz dual-socket eight-core and 3 GB of RAM per core. Each server is connected to 8 Tesla processors via PCI-e Gen2 $\times$16 slots. The Forge results were compiled using Red Hat Enterprise Linux 6 (Linux 2.6.32) and the GNU compiler [29].

1.6.4 Keeneland Supercomputer

Keeneland Initial Delivery (KID) is a another Teragrid/XSEDE GPU cluster located at National Institute for Computational Science (NICS) (see figure 1.10). KID has 360 Tesla M2070 NVIDIA Fermi GPUs that each comes with 6 GB DDR5 mem-

Figure 1.9. Forge Teragrid/XSEDE GPU cluster with 288 Tesla 20 series GPUs (from [8])

ory. KID's 120 servers each hold two hex-core Intel Xeon (Westmere-EP) 2.93 GHz (11.72 GFlops) and 2 GB of DDR3 RAM per CPU core. Each server is connected to 3 Tesla processors via PCI-e Gen2 $\times$16 slots. Also nodes are connected by an $\times$8 InfiniBand QDR (single rail) network [30].

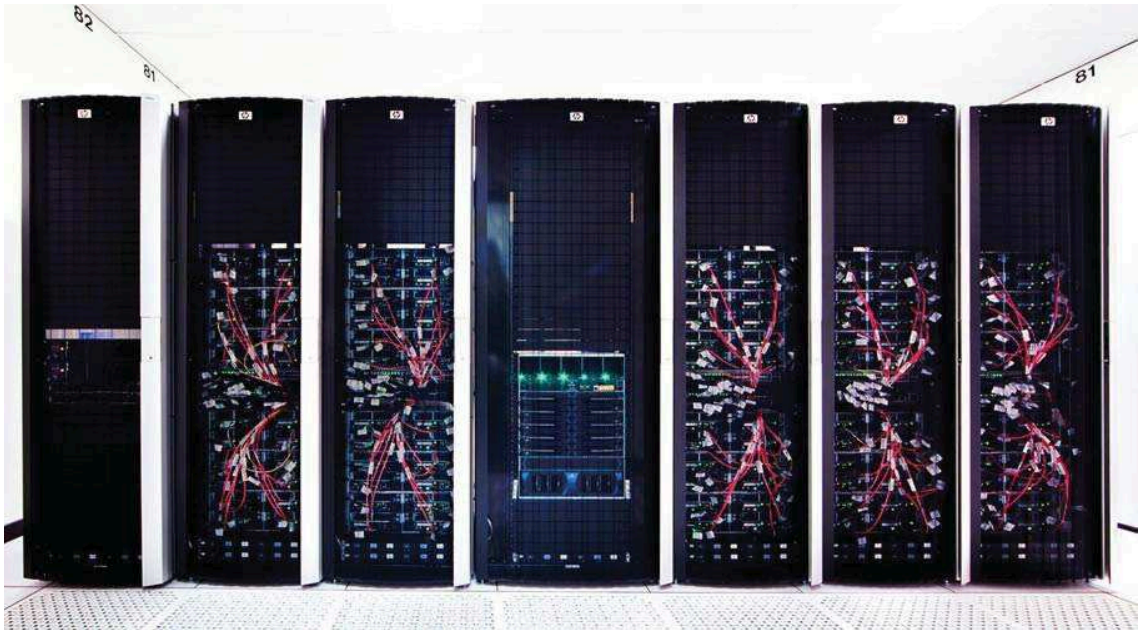


Figure 1.10. Keeneland Initial Delivery Teragrid/XSEDE GPU cluster with 360 Tesla 20 series GPUs (from [9])

CHAPTER 2

AN INTRODUCTION TO GPU PROGRAMMING

2.1 Basic GPU Programming Model and Interface

Before tackling more advanced topics, it is necessary to clarify some basic terms in the CUDA programming model:

Thread: is a ready-for-execution/running instance of a kernel. Each thread has its own instruction address counter and register state. Each thread can be identified by a combination of its three-dimensional thread index and three-dimensional thread-block index. CUDA threads are lightweight. This means that creating and destroying a thread is very fast.

Warp: is a group of 32 parallel threads. The multiprocessors create and execute warps in order. All threads in a warp start together but each of them can have its own instruction and registers.

Block: is a group of Warps. A block is executed on one multiprocessor. Every block has its own shared memory and registers in the multiprocessor.

Grid: is a group of blocks. There should be at least as many blocks as multiprocessors (and preferably at least 2× as many).

Host: is the CPU in CUDA applications.

Device: is the GPU in CUDA applications.

SIMT: stands for Single-Instruction, Multiple-Thread and is identical to the more commonly used term - SIMD. A multiprocessor can execute hundreds of threads concurrently (1536 threads in the Fermi architecture). There is special hardware in the GPU architecture to create, mange, schedule and execute such a large number of threads.

2.2 Device Memories

In the CUDA programming paradigm, the host and device have their own separate memories. This means that each device is typically a hardware card that comes with its own Dynamics Random Access Memory (DRAM). For example, the NVIDIA GeForce GTX 480 card comes with 1.5 GB of DRAM. In order to execute the code on the GPU, the programmer needs to allocate memory on the device and copy data from host the device. Also after execution, data should be copied back to the host and free up the device memory [13].

Figure 2.1 demonstrates a schematic of CUDA device memory model. There are the memories that the host or device can write to and read from. There are also read-only memories. The description and features of each memory type will be discussed in detail in next section.

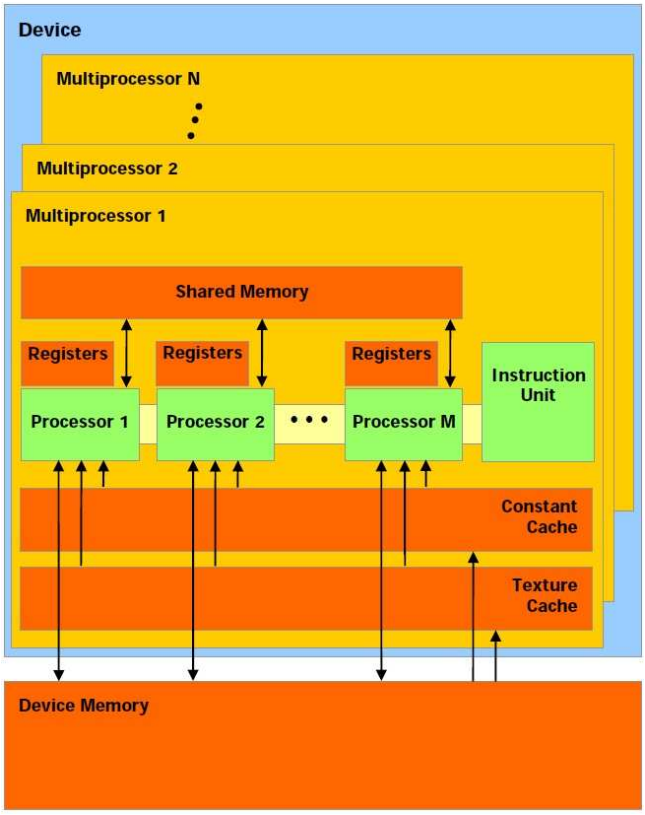


Figure 2.1. GPU hardware model (from [2])

2.2.1 Global Memory (Device Memory)

Global memory resides in device memory and is the largest and primary memory on the device. Global memory has a high latency so reading data from global memory or writing to it, is slow, taking 400 to 600 cycles [2]. The access pattern to the device memory is a key factor to hiding the latency and increasing the speed. A global memory request for a single warp is divided in to two memory request. Each request of a half-warp (16 threads) can be issued independently. Depending on the device the memory access of the threads are coalesced to one or more memory transactions. In order to maximize the device bandwidth, it is therefore important to maximize coalescing access. Global memory has its own advantages:

- Large amount of memory available for a CUDA application

- Accessible from host and device
- Possibility to hide memory latency when coalescing access is applied

On the other hand there are some disadvantages for global memory

- High memory latency
- The accessing pattern must be coalesced (spatially local) to achieve high bandwidth

2.2.2 Local Memory

Local memory resides in the device memory and is per thread memory. Local memory is 'overflow' memory and it is slow and should be avoided. If the kernel requires too many register values, the compiler uses local memory as an additional memory store [2]. Because local memory is essentially just overflow registers being stored in global memory it has a very high latency. By default, short arrays are saved in local memory.

2.2.3 Texture Memory

CUDA supports texture memory that is a subset of the texturing hardware, and a holdover from graphics processing (where all memory was texture memory). The texture memory is read-only and cached and resides in device memory. If memory accesses are not spatially local - but are temporally local, then the caching of this memory can improve the performance of the application [2]. Reading data from texture memory instead of global memory has several performance advantages:

- Increase the performance by reading data from cache if coalesced access is hard to achieve when using global memory
- Broadcasting packed data to separate variables in a single operation (decrease the number of accesses to the global memory)

- Converting 8-bit and 16-bit to 32-bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] for free. (color processing)
- Can allocate large texture memory on the device
- The texture cache is specially optimized for 2D rectilinear spatial locality (like a screen display).
- Linear, bilinear, and tri-linear interpolation are almost free of charge (using dedicated hardware for interpolations)

On the other hand using texture memory has some disadvantages:

- So far CUDA can only support up to 32-bit floating-point values for texture memory (No double precision).
- If coalesced access to global is easy to achieve, then using texture memory doesn't have any performance advantages and in some cases decreases the performance by initializing the texture memory.
- Because the texture cache is optimized for 2D rectilinear spatial locality, its utility is limited for anything but graphics.

2.2.4 Shared Memory

The shared memory is on-chip and has much faster memory access speeds than the local and global memory. In order to increase the bandwidth, shared memory is divided into 32 equally-sized memory modules (in the Fermi architecture), called banks, which can be accessed simultaneously by all threads in the same warp. If there are no bank conflicts for all threads of a warp then access to the shared memory is very fast. Bank conflicts happen when two threads in a same warp try to read from or write to the same bank. If a bank conflict happens the access (read or write) has to be serialized. But if all threads try to read from the same address (broadcasting)

from shared memory there is no bank conflict [2]. Using shared memory has some benefits:

- Low latency compared with global memory
- Supports single and double precision
- Synchronize read and write for all threads in the same block

On the other hand shared memory has some disadvantages:

- Limited size of the memory (48KB per multiprocessor for Fermi architecture)
- Avoiding bank conflicts in some cases is hard to achieve.
- Different blocks can't access each other's shared memory even if they are running on the same multiprocessor

2.2.5 Constant Memory

The constant memory space resides in device memory but has limited size (64KB) and is cached in the constant cache. Constant cache is 6KB or 8KB per multiprocessor. Like texture memory, constant memory is read-only memory. Access to constant memory has one cycle latency when there is a cache hit and hundreds of cycles when there is a cache miss [2]. Constant memory has some benefits:

- Usually fast
- Saves registers and procedure arguments
- Possibility to decrease the amount of shared memory
- Decreases the loading kernel over head to the device

Also constant memory has some disadvantages:

- Limited size of constant memory
- High memory latency for first time accesses

2.2.6 Page-Locked Host Memory

There is another memory allocation in CUDA called Page-Locked Host Memory. This is a type of memory on the CPU (that can not be moved about by the OS). Page-locked memory has its own benefits:

- It is possible to copy between page-locked memory and device memory while running a kernel on that device at the same time
- CUDA can map page-locked memory into the address space of the memory on the device. This eliminates the need for a copy command from the device to the host or vice versa
- Bandwidth between page-locked memory and the device is high

Page-locked memory has some disadvantages too:

- Using more page-locked memory decrease overall system performance of the OS
- Reading from some page-locked memories (like write-combining memory) from host is prohibitively slow

There are four different types of page-locked memory available for CUDA applications. The flags parameter in page-locked memory enables different options to be specified that affect the allocation. All of these flags are orthogonal to one another: a programmer may allocate memory that is portable, mapped and/or write-combined with no restrictions. So far the benefits of all these options are marginal (20% faster).

2.2.6.1 Pinned Memory

This is the default type of memory for the page-locked memory. In this type of memory copying data between the host and the device or vice versa should be done by the program. By using different streams for kernel launch and copy command, it is possible to overlap and therefore hide the copying time. For using this type of

memory the programmer should pass the *cudaHostAllocDefault* flag to page-locked memory definition [2].

2.2.6.2 Portable Memory

Page-locked memory can only be used by the host thread that allocated the memory. By passing *cudaHostAllPortable* flag to page-locked memory, CUDA makes page-locked memory available for all host threads [2].

2.2.6.3 Write-Combining Memory

By default, page-locked memory is allocated as cacheable memory. Write-combining memory frees up L1 and L2 cache resources and causes more cache to be available for the rest of the application. Also, transfer rates can be improved up to 40% when using write-combining memory. But reading write-combining memory from the host is extremely slow. Write-combining memory should be only used when the host writes to that memory and device reads from that memory. By passing the *cudaHostAllocWriteCombined* flag to page-locked memory, CUDA treats page-locked memory as a write-combined memory [2].

2.2.6.4 Mapped Memory

A page-locked host memory can be mapped into the address space of the device memory by passing *cudaHostAllocMapped* flag to the page-locked memory. Therefore there are two addresses for that memory: one on the host side and another one is on the device side [2]. Mapped memory has several benefits:

- Copying data between the host and the device is done implicitly when the data is needed by the kernel
- Because CUDA is implicitly handling the copy, there is no need to use streams to overlap the copy execution.

Because every device call is an asynchronous call by default, it is necessary to synchronize the device kernel to avoid any read-after-write, write-after-read and write-after-write hazards.

2.3 Performance Optimization Strategies

There are some optimization techniques that should be considered in order to achieve high performance on GPUs [2, 31, 32].

- Minimize the data transfer between host and device
- Maximize coalesced access to the global memory where possible
- Minimize the use of the global memory. Use shared memory or constant memory instead of global memory where possible
- Use suitable memory type for saving data (Texture, constant, mapped or write-combining memory)
- Overlap copying data with kernel launches where possible
- Run kernels concurrently where possible
- Minimize CUDA synchronization call
- Use single large copy instead of many small copies
- Minimize thread divergence
- Avoid bank conflicts when using the shared memory
- Maximize the occupancy where possible. In general, occupancy should be greater than 25%
- Launch the blocks with multiples of 32 threads.

- Use faster and specialized math functions instead of slow and accurate math functions where possible
- Use page-locked memory where possible
- Avoid atomic functions. For example if two threads perform an atomic operation at the same memory address at the same time, those operations will be serialized. The order in which the operations complete is undefined, which is fine, but the serialization can be quite costly.
- Avoid integer division modulo operation (close to 20 instructions) and use shift operation where possible

2.4 Multi-GPU Optimization Strategies

Hiding the communication time on the GPU is much more complicated than on the CPU. There are three steps that should be overlapped with the computation to completely conceal the communication time; loading data from GPU to the CPU, sending and receiving the data (between CPUs) with MPI and finally copying data back to the GPU. The bandwidth between the CPU and GPU is so slow (5-10 GB/s). On the other hand, the bandwidth for QDR InfiniBand is close to 10 GB/s. Comparing with high bandwidth GPUs (178 GB/s), hiding communication time is a key factor in multi-GPU implementations. Currently, all graphic cards support concurrent copy and kernel execution. The idea behind the hiding is to keep the GPU working on data and copy data from it and send/receive boundary values to/from other nodes with MPI and copy back the data to the GPU.

CPUs always have better performance when the problem is small. Because all data can easy fit into a CPU memory's cache. In contrast, GPUs work best when the problem size is large. When the problem size is large enough, this has many benefits. First, the cost of initialization is relatively small, and it is efficient that copy/send/receive

large data once instead of copy/send/receive many small data packets. Second, large data means it is easy to keep the GPU busy, when communication is needed. Details of the implementation and the effect of these parameters is discussed in chapter 5.

CHAPTER 3

DATA ANALYSIS ON THE GPUS

3.1 STREAM Benchmark

3.1.1 Problem Statement

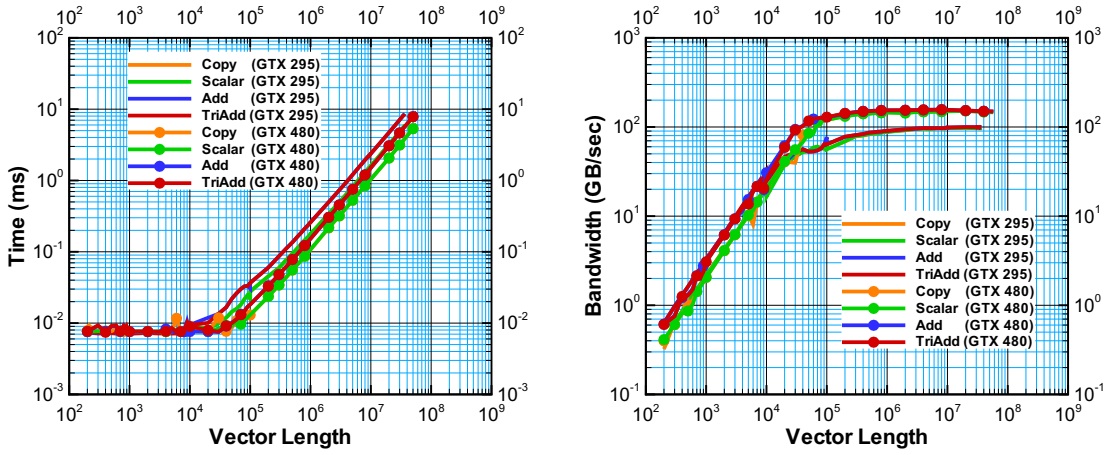
The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. The STREAM benchmark is composed of four kernels. In the first kernel one vector is copied to another within the same device ($a = b$, one read and one write). For the second kernel, vector entries are multiplied by a constant number and the results is written to another vector ($a = \alpha b$, one read and one write). Kernel 3 adds two vectors ($a = b + c$, two reads and one write). Finally, Kernel 4 is a combination of Kernels two and three, sometimes referred to as a DAXPY operation ($a = \alpha b + c$, two reads and one write). The benchmark was computed using a single GPU to operate on different vector sizes. Table 3.1 shows the specifications for four types of NVIDIA GPUs. The GTX 480, Tesla C2070, GTX 295 and Tesla S1070 actually house single, single, two and four GPUs in a single box, respectively. Accordingly, the hardware specifications from NVIDIA shown below are for one of these GPUs (1/2 of the GTX 295 and 1/4 of the Tesla S1070) [33, 34, 35, 36]. The test shown below used 64k blocks with 256 threads each, operating on double precision vectors. Each GPU kernel was called 100 times, and each kernel performs the STREAM operation 100 times in that kernel. The timings below are reported for a single STREAM operation (total time / 10,000).

Table 3.1. NVIDIA hardware specifications for GTX 295, GTX 480, Tesla S1070 and Tesla C2070

Model	Cores	Memory (MB)	Theoretical Bandwidth (GB/s)	Memory Interface Width (bit)	Max Power (W)
GTX 295	240	896	119.9	448	145
GTX 480	480	1500	177.4	384	250
Tesla S1070	240	4000	102	512	200
Tesla C2070	448	6000	144	384	238

3.1.2 Single GPU

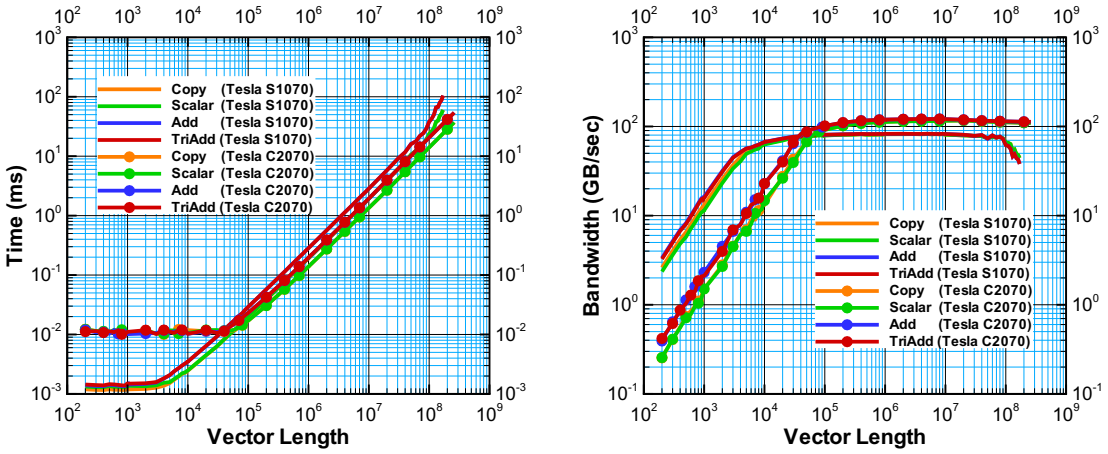
Figures 3.1 and 3.2 show the single-GPU execution time and bandwidth for the GTX 295, GTX 480, Tesla S1070 and Tesla C2070 GPUs, respectively. For vectors with lengths less than 10^5 elements, the time is nearly constant and the bandwidth is less than the maximum value for 10 series and 20 series respectively. However, for vector sizes larger than 10^5 the bandwidth is close too the maximum value and execution time increases linearly with vector length. This shows that the startup cost of simply initiating a GPU kernel is high, and large vector lengths are required for good GPU performance



(a) Time (b) Bandwidth

Figure 3.1. Time and bandwidth for single NVIDIA GTX 295 and GTX 480 for four different STREAM kernels

The Tesla S1070 has more memory (4 GB) than the GTX 295 (896 MB). However, figure 3.2 shows that for large vector lengths (greater than 5×10^7) bandwidth begins to decrease. For the largest possible lengths on the Tesla S1070, bandwidth is approximately 50% of the maximum value. However, there is no efficiency loses for Tesla C2070 when using large vector sizes. Also figure 3.2 shows that for small problem sizes Tesla S1070 has better performance than C2070. The kernel startup time for the C2070 is an order of magnitude larger than its predecessor. The bandwidth achieved by the STREAM benchmark is consistently about 20% below the theoretical peak predicted by NVIDIA for all GPUs.



(a) Time

(b) Bandwidth

Figure 3.2. Time and bandwidth for single NVIDIA Tesla S1070 and Tesla C2070 for four different STREAM kernels

3.1.3 Weak Scaling on Many GPUs

In the weak scaling case, the vector length is constant per GPU (at 2M double precision elements) as the number of GPUs increases. This makes the number of operations constant per GPU as the number of GPUs increases.

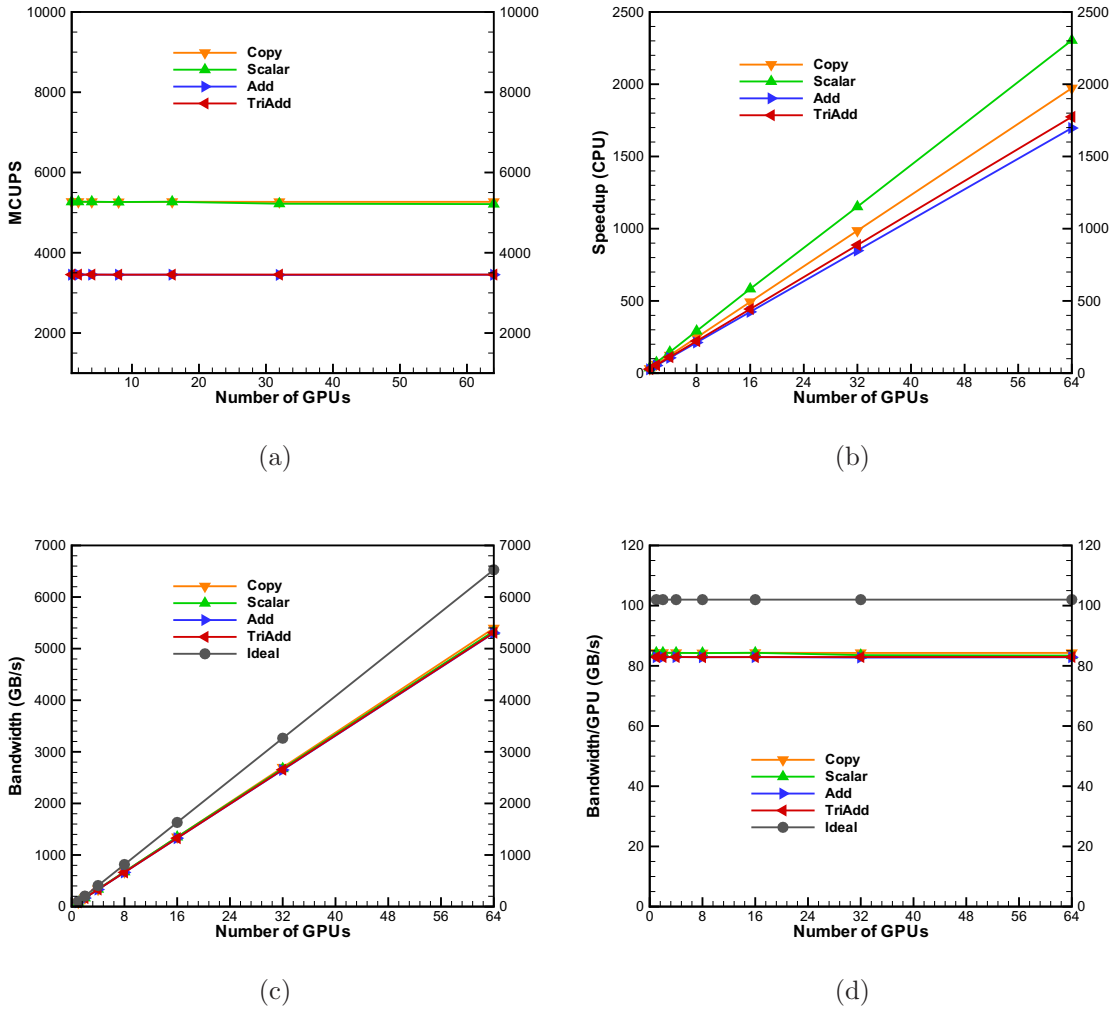


Figure 3.3. Results for weak scaling of the four STREAM benchmark kernels on Lincoln with 2M elements per GPU, (a) MCUPS, (b) speedup comparing with a single core of the AMD processor on Orion, (c) Actual and ideal bandwidth, (d) bandwidth per GPU for various numbers of GPUs

Figure 3.3 shows the results for weak scaling. Figure 3.3(a) shows millions of cell updates per second (MCUPS) for different numbers of GPUs while Figure 3.3(b) shows the speedup compared with a single core of the AMD CPU. Figures 3.3(c) and 3.3(d) show total bandwidth as well as bandwidth per GPU for various numbers of GPUs. Because 2M is large enough for a single GPU to be efficient, the bandwidth is close to the maximum value and the speedup is linear.

3.1.4 Strong Scaling on Many GPUs

Figure 3.4 shows the strong scaling results. In the strong scaling case, the vector length remains constant as the number of processors varies. This means that an increased number of GPUs decreases the vector length per GPU. The total vector length used for the strong scaling case was 32M. This results in 0.5M elements per GPU when 64 GPUs are used for the computation.

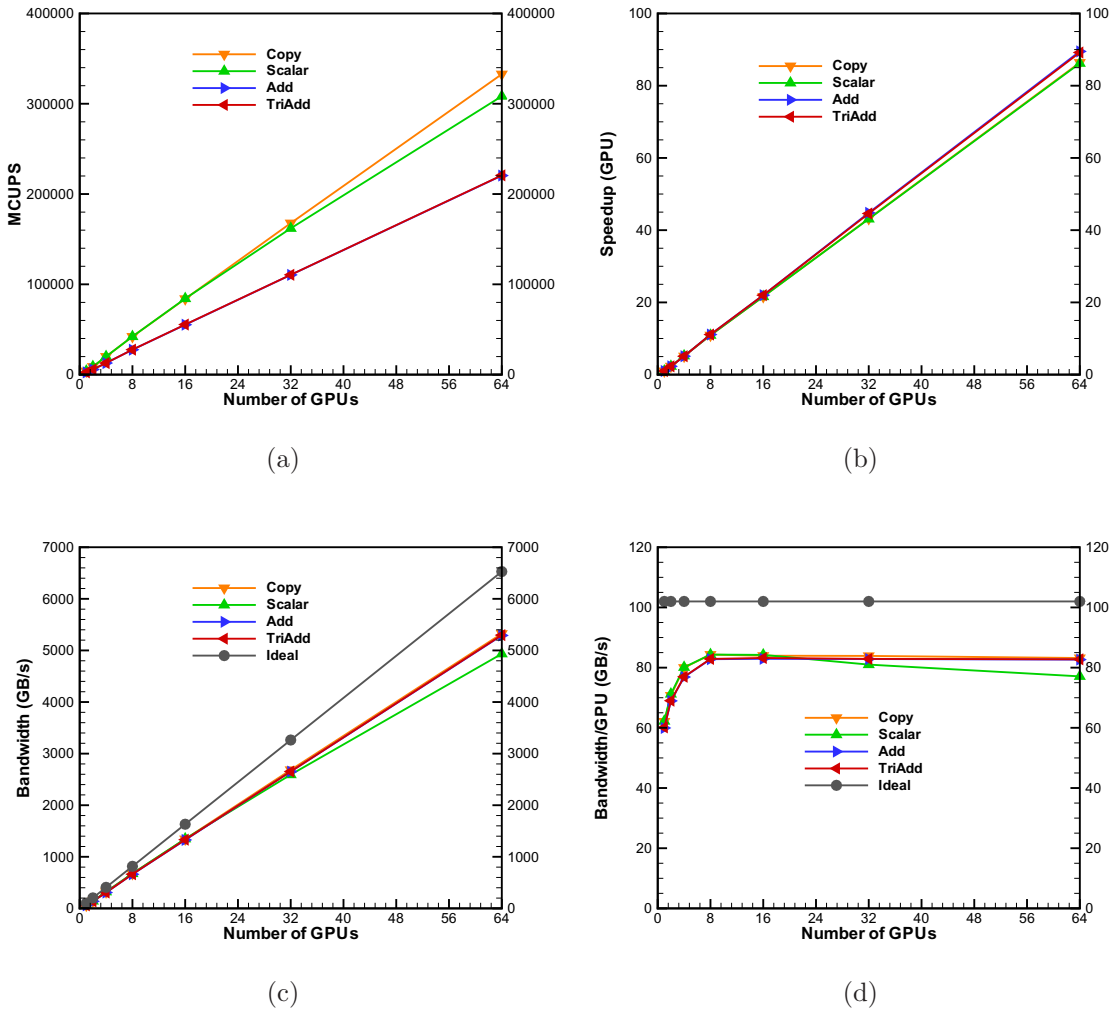
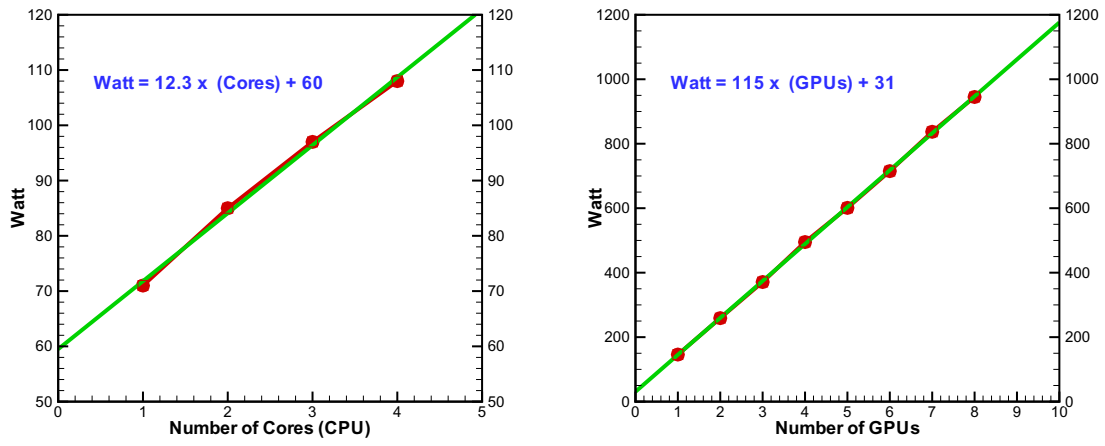


Figure 3.4. Results of strong scaling of the four STREAM benchmark kernels on the Lincoln with 32M total elements,(a) MCUPS, (b) speedup comparing with single Tesla S1070 GPU, (c) Actual and ideal bandwidth, (d) bandwidth per GPU for various numbers of GPUs

Figure 3.4(a) shows the MCUPS while figure 3.4(b) shows the speedup for strong scaling comparing with a single GPU. Figures 3.4(c) and 3.4(d) show the total bandwidth and bandwidth per GPU for various numbers of GPUs, respectively. The superlinear speedup in figure 3.4(b) is an artifact of the single GPU case being relatively slow because the vector length is so large (greater than 10^7, see figure 3.2(b)) that the performance is suboptimal for the single GPU case.

3.1.5 Power Consumption

Figures 3.5(a) and 3.5(b) show power consumption for the AMD quad-core Phenom II X4, operating at 3.2 GHz, and for the GTX 295 GPUs, respectively. This test involved the STREAM Benchmark operating on double precision vectors of length 2M per GPU or per CPU core.



(a) (b)

Figure 3.5. Power consumption for the weak scaling STREAM benchmark with the AMD CPU and GTX 295 GPUs

Each GPU uses approximately 115 W over idle consumption when running the STREAM Benchmark. Also, it was found that the idle power for one GTX 295 (2 GPUs) is 71 W (or about 30 W per GPU). This was ascertained by physically

removing GPUs from the machine, and re-running the code. The NVIDIA hardware specifications imply that each Tesla S1070 GPU uses more power than the GTX 295 GPUs (200 W vs. 145 W maximum). One reason for this difference could be the amount of memory. The Tesla S1070 has 4000 MB per GPU but the GTX 295 has only 896 MB per GPU. The Tesla also contains its own power supply and cooling system, which may be playing a role in the difference in consumption levels. Unfortunately, it wasn't direct access to the Tesla hardware (at NCSA) to actually measure its power consumption directly.

3.1.6 GPU Temperature

The STREAM benchmark was computed with GTX 295 GPUs to obtain a measurement of each GPU's temperature. The code was run with 8 GPUs for 331 seconds. Table 3.2 shows the initial and maximum temperatures for each GPU, as the STREAM benchmark was running for 331 seconds. The maximum allowable temperature for the GTX 295 listed on NVIDIA's website is 105 $°C$ [35]. The maximum GPU temperatures range between 88-96 $°C$ when running for this prolonged period of time. Since most GPGPU applications involve many short kernel calls rather than this type of extended exertion, GPU temperatures are typically lower than these maximum measured values.

Table 3.2. Temperatures for GTX 295 cards, running for 331 second STREAM benchmark on Orion (see figure 1.7 for fans and GPUs configuration)

Temperature	**GPU 1**	**GPU 2**	**GPU 3**	**GPU 4**	**GPU 5**	**GPU 6**	**GPU 7**	**GPU 8**
Initial Temp $°C$	66	64	72	68	72	67	63	61
Max Temp $°C$	90	88	96	92	96	93	91	88
ΔT	24	24	24	24	24	26	28	27

3.2 Unbalanced Tree Search

3.2.1 Problem Statement

The Unbalanced Tree Search (UTS) benchmark performs an exhaustive search on an unbalanced tree. The tree is generated on the fly using a splittable random number generator (RNG) that allows the random stream to be split and processed in parallel while still producing a deterministic tree. There are two kinds of trees evaluated in this work, binary trees and geometric trees. The binary tree is based on a probability that each node can have children (or not). For the binary tree, each node either has 8 children or none at all. For the geometric tree, each node has 1 to 4 children with the number of children being assigned randomly (see figure 3.6). Geometric trees are terminated at a predefined level. Nodes greater than the terminating level have no children.

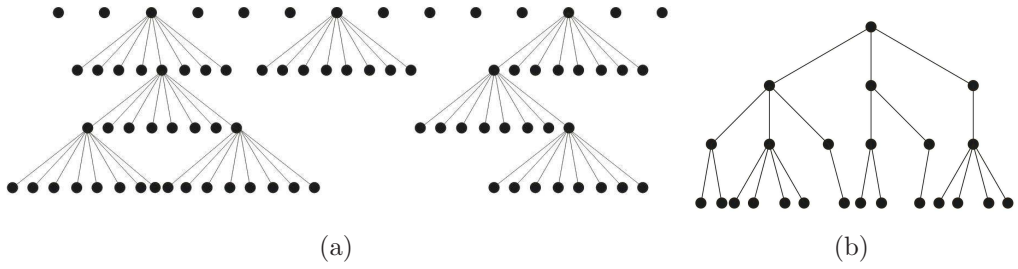


Figure 3.6. Representations of (a) a binary tree, with nodes having 0 or 8 children, and (b) a geometric tree, with 1-4 randomly assigned children

There are two well-known schemes for dynamic load balancing: work sharing and work stealing. In a work sharing approach there is a global shared queue, and each processor has its own chunk of data. If the number of nodes on a processor increases beyond a fixed number, then it will start to write the extra data to the shared queue. Similarly, if there are not enough nodes on a processor, it will start to read nodes from the queue.

In work stealing there is no central queue (which can be a bottleneck). When there are not enough nodes for a processor to work on, it takes nodes directly from processors that have too many. The advantage of work stealing is that there is no communication when all processors are working on their own data set, making this an efficient scheme. In contrast, work sharing can be inefficient because it requires load balancing messages to be sent even when all the processors have work to do [37]. Unfortunately, neither approach is particularly well-suited for the GPU because it is not possible to communicate directly between two GPUs. All communication must go through the CPU (via MPI). Copying data from the GPU to the CPU or vice versa is expensive and should be avoided. To circumvent these problems, load balancing is divided into two parts, load balancing between the CPUs and load balancing on the GPU.

Each computational team is comprised of one CPU and one GPU - each with its own queue. Each GPU works on the computation while the CPUs perform the load balancing via MPI. After launching the GPU kernel, the CPUs begin load balancing amongst themselves. For this CPU balancing, they rank themselves by nodes per process, and then begin sending work to one another based on this ranked list. The protocol for load redistribution is that the CPU with the most work shares with the CPU with the least amount of work, the CPU with the second most work shares with the CPU with the second least work, and so on. When the GPU finishes its work, it begins reading new data from the CPU. If a GPU has too much work, it will periodically stop and deliver that data to the CPU to be redistributed elsewhere. The GPU memory copies to and from the CPU are easily overlapped with GPU computation using the *cudaMemcpyAsync* command [32].

The last box in figure 3.7 represents two check conditions. The whole UTS algorithm is inside the while loop. If the number of nodes in every GPU is equal to zero the program is terminated.

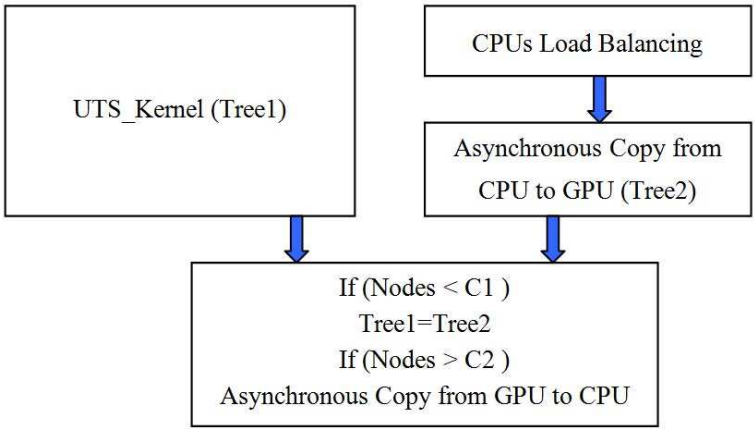
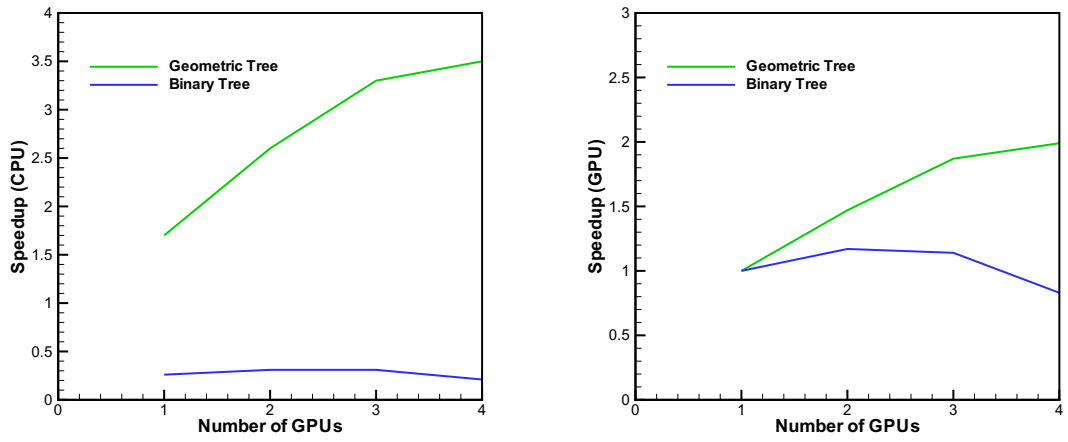


Figure 3.7. Implemented algorithm for UTS

3.2.2 Results for Unbalanced Tree Search

Figure 3.8 shows results for two different tree searches using various numbers of GPUs. The binary tree is started with 5000 nodes and the maximum depth of the tree is 653. The total number of nodes is 1,098,896. The geometric tree is started with four nodes and has a maximum depth of 12.



(a) (b)

Figure 3.8. Strong scaling speedups for the unbalanced tree search relative to (a) a single CPU and (b) a single GPU using GTX 295 GPUs (Orion)

Because the binary tree starts with only 5,000 nodes, there is not enough work initially for even a single GPU. Also, the random number generator is so fast that it is not possible to successfully hide the load balancing. For the geometric tree, again, there is not enough work to require multiple cards until the tree grows beyond level 7. Additionally, once this level is reached, the geometric tree begins to grow very fast. Finally, like the binary tree, the random number generator is very fast compared to data transfers via MPI. For all binary tree searches and for the first 7 levels of the geometric tree, the CPU is capable of putting all data in its cache. For load balancing, the GPU has to exit the kernel and write data to global memory that is 100 times slower than the CPU's cache memory.

CHAPTER 4

SEQUENCE MATCHING

4.1 Introduction

This chapter describes the solution of a single very large pattern matching search using a supercomputing cluster of GPUs. The objective is to compare a query sequence which has a length on the order of $10^2 - 10^6$ with a 'database' sequence which has a size on the order of roughly 10^8, and find the locations where the test sequence best matches parts of the database sequence. Finding the optimal matches that can account for gaps and mismatches in the sequences is a problem that is non-trivial to parallelize. This study examines how to achieve efficient parallelism for this problem on a single GPU and between the GPUs when they have a relatively slow interconnect.

Pattern matching in the presence of noise and uncertainty is an important computational problem in a variety of fields. It is widely used in computational biology and bioinformatics. In that context DNA or amino acid sequences are typically compared to a genetic database. More recently optimal pattern matching algorithms have been applied to voice and image analysis, and to the data mining of scientific simulations of physical phenomena.

The Smith-Waterman algorithm is a dynamic programming algorithm for finding the optimal alignment between two sequences once the relative penalty for mismatches and gaps in the sequences is specified. For example, given two RNA sequences, *CAGCCUCGCUUAG* (the database) and *AAUGCCAUUGCCGG* (the test sequence), the optimal region of overlap (when the gap penalties are specified accord-

ing to the Scalable Synthetic Compact Application No. 1 (SSCA#1) benchmark) is shown below in bold

CA**GCC–UCGC**UUAG

AAU**GCCAUUGC**CGG

The optimal match between these two sequences is eight items long and requires the insertion of one gap in the database and the toleration of one mismatch (the third to last character in the match) as well as a shifting of the starting point of the query sequence. The Smith-Waterman algorithm determines the optimal alignment by constructing a table that involves an entry for every item of the query sequence and in the database sequence. When either sequence is large, constructing this table is a computationally intensive task. In general, it is also a relatively difficult algorithm to parallelize, because every item in the table depends on all the values above it and to its left.

This chapter discusses the algorithm changes necessary to solve a single very large Smith-Waterman problem on many internet connected GPUs in a way that is scalable to any number of GPUs and to any problem size. Prior work on the Smith-Waterman algorithm on GPUs [38] has focused on the quite different problem of solving many (roughly 400,000) entirely independent small Smith-Waterman problems of different sizes (but averaging about 1K by 1K) on a single GPU [39].

Within each GPU this chapter shows how to reformulate the Smith-Waterman algorithm so that it uses a memory efficient parallel scan to circumvent the inherent dependencies. Between GPUs, the algorithm is modified so as to reduce inter-GPU communication.

4.2 The Smith-Waterman Algorithm

The Smith-Waterman algorithm is a well-known dynamic programming method for finding similarity between DNA, RNA, nucleotide or protein sequences. The main idea behind the dynamic programming is to divide the problem into small segments and solve each segment separately. In the end, the results are combined to complete the solution. The algorithm was first proposed in 1981 by Smith and Waterman and identifies similar regions between sequences by searching for optimal local alignments [40]. To find the best local alignment between two sequences, an appropriate scoring system including a set of specified gap penalties and similarity values is required. The Smith-Waterman algorithm is built to find segments of all possible lengths between two sequences that are similar to each other based on a defined scoring system. This means that the Smith-Waterman algorithm is very accurate and finds an optimal alignment between two sequences. Unfortunately, the Smith-Waterman algorithm is both time consuming and CPU intensive. Because this algorithm is time consuming, there has been a lot of development and suggestions for optimizations. One example is the well-known Basic Local Alignment Search Tool, BLAST that works based on heuristic acceleration algorithms [41, 42].

It might be difficult to identify any good alignments between two sequences that are only distantly related, as they sometimes have regions of low similarity. The local alignment searches are useful for comparing global alignment. Identifying the best local alignment between two sequences is essentially what the Smith-Waterman algorithm is looking for [43].

Contrary to the Needleman-Wunsch algorithm [44], on which the Smith-Waterman algorithm is built, the Smith-Waterman algorithm is searching for the best local alignments, not global or multiple alignments, considering segments of all possible lengths to find the similarities between sequences [43].

The Smith-Waterman algorithm uses individual pair-wise comparisons between characters as:

$$H_{i,j} = \max \begin{cases} \max(H_{i-1,j-1} + S_{i,j}, 0) & \\ \max(H_{i-k,j} - (G_s + kG_e)) & 0 < k < i \\ \max(H_{i,j-k} - (G_s + kG_e)) & 0 < k < j \end{cases} \tag{4.1}$$

Here, G_s and G_e are the gap start-penalty and gap extension-penalty, respectively. The similarity score S, is given by the user or particular application. In realistic biological applications the gap penalties and similarity scores can be quite complex. In this work a simple scoring system is used from the HPC SSCA#1 (Scalable Synthetic Compact Application) benchmark in which matching items get a score of 5 and dissimilar items have a score of -3, the gap start-penalty is 8 and gap extension-penalty is 1.

The first data sequence (the database) is usually placed along the top row, and the second sequence (the query sequence) is usually placed in the first column. The table values are called $H_{i,j}$ where i is the row index and j is the column index (see figure 4.1). The algorithm is initiated by placing zeros in the first row and column of the table. The other entries in the table are then set via the equation 4.1.

	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
A	*0*	0	5	0	0	0	0	0	0					
A	*0*	0	5	2	0	0	0	0	0					
U	*0*	0	0	2	0	0	5	0	0					
G	*0*	0	0	5	0	0	0	2	5					
C	*0*	5	0	0	10	5	0	5	0					
C	*0*	5	2	0	5	15	6	5	4					
A														
U														
U														
G														
C														
C														
G														
G														

Figure 4.1. Dependency of the values in the Smith-Waterman table

The algorithm assigns a score to each residue comparison between two sequences. By assigning scores for matches or substitutions and insertions/deletions, the comparison of each pair of characters is weighted into a matrix by calculation of every possible path for a given cell. In any matrix cell the value represents the score of the optimal alignment ending at these coordinates and the matrix reports the highest scoring alignment as the optimal alignment (see figure 4.2). For constructing the optimal local alignment from the matrix, the starting point is the highest scoring matrix cell. The path is then traced back through the array until a cell scoring zero is met. Because the score in each cell is the maximum possible score for an alignment of

any length ending at the coordinates of this specific cell, aligning this highest scoring segment will yield the optimal local alignment.

	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
A	*0*	0	5	0	0	0	0	0	0	0	0	0	5	0
A	*0*	0	5	2	0	0	0	0	0	0	0	0	5	2
U	*0*	0	0	2	0	0	5	0	0	0	5	5	0	2
G	*0*	0	0	***5***	0	0	0	2	5	0	0	2	2	5
C	*0*	5	0	0	***10***	5	0	5	0	10	1	0	0	0
C	*0*	5	2	0	5	***15***	6	5	4	5	7	1	0	0
A	*0*	0	10	1	0	***6***	12	3	2	1	2	4	6	0
U	*0*	0	1	7	0	5	***11***	9	1	0	6	7	1	3
U	*0*	0	0	0	4	4	10	***8***	6	0	5	11	4	1
G	*0*	0	0	5	0	3	1	7	***13***	4	3	2	8	9
C	*0*	5	0	0	10	5	0	6	4	***18***	9	8	7	6
C	*0*	5	2	0	5	15	6	5	4	9	15	6	5	4
G	*0*	0	2	7	0	6	12	3	10	8	6	12	3	10
G	*0*	0	0	7	4	5	3	9	8	7	5	3	9	8

Figure 4.2. Similarity matrix and best matching for two small sequences *CAGC-CUCGCUUAG* (top) and *AAUGCCAUUGCCGG* (left). The best alignment is: *GCC-UCGC* and *GCCAUUGC* which adds one gap to the test subsequence and which has one dissimilarity (3rd to last unit).

The Smith-Waterman algorithm is quite time demanding because of the search for optimal local alignments, and it also imposes some requirements on the computer's memory resources as the comparison takes place on a character-to-character basis.

The fact that similarity searches using the Smith-Waterman algorithm take a lot of time often prevents this from being the first choice, even though it is the most

precise algorithm for identifying similarity regions between sequences. BLAST [41] and FastA [45, 46] are heuristic approximations of the Needleman-Wunsch and Smith-Waterman algorithms. These approximations are less sensitive and do not guarantee to find the best alignment between two sequences. However, these methods are not as time-consuming as they reduce computation time and CPU usage.

Today's research requires fast and effective data analysis. Algorithms like BLAST have therefore largely replaced Smith-Waterman searches as demands on the time of handling large amounts of data are still getting stronger. On the other hand, database searches are getting more and intensive and researchers are becoming more and more concerned about the time and accuracy of searching algorithms. As a matter of fact, the efficient implementation of the Smith-Waterman algorithm is again becoming an active area of research [43].

The Smith-Waterman algorithm can be accelerated based on FPGA (Field-Programmable Gate Array) chips or by using the SIMD technology (Single Instruction, Multiple Data) or with GPUs which parallelize and thereby accelerate the computations.

4.3 The Optimized Smith-Waterman Algorithm

4.3.1 Reduced Dependency

If equation 4.1 is naively implemented, then the column maximum and row maximum (second and third items in equation 4.1 are repetitively calculated for each table entry causing $O(L_1L_2(L_1 + L_2))$ computational work (L_1 and L_2 are the length of the database and query sequences). This is alleviated by retaining the previous row and column sums [47], called $F_{i,j}$ and $E_{i,j}$ respectively. This increases the storage required by the algorithm by a factor of 3 but reduces the work significantly. The Smith-Waterman algorithm is now given by,

$$
\begin{aligned}
E_{i,j} &= \max(E_{i,j-1}, H_{i,j-1} - G_s) - G_e \\
F_{i,j} &= \max(F_{i-1,j}, H_{i-1,j} - G_s) - G_e \\
H_{i,j} &= \max(H_{i-1,j-1} + S_{i,j}, E_{i,j}, F_{i,j}, 0)
\end{aligned} \tag{4.2}
$$

4.3.2 Anti-Diagonal Approach

The Smith-Waterman algorithm can be parallelized by operating along the anti-diagonal [48]. Figure 4.3 shows (in grey) the cells that can be updated in parallel. If the 3-variable approach is used, then the entire table does not need to be stored. An anti-diagonal row can be updated from the previous E and F anti-diagonal row, and the previous two anti-diagonal rows of H. The algorithm can then store the i and j location of the maximum H value so far. When the entire table has been searched it is then possible to return to the maximum location and rebuild the small portion of the table necessary to reconstruct the alignment sub-sequences.

The anti-diagonal method has startup and shutdown issues, that make it somewhat unattractive to program. It is inefficient (by a factor of roughly 50%) if the two input sequences have similar sizes.

For very dissimilar sized input sequences, the maximum length of the anti-diagonal row is the minimum of the two input sequence lengths. On a GPU we would like to operate with roughly 128 to 256 threads per block and at least two blocks per multiprocessor (32-60 blocks). This means that we need query sequences of length 104 or greater to efficiently occupy the GPU using the anti-diagonal method. This is an order of magnitude larger than a typical query sequence.

The anti-diagonal method can not operate entirely with just registers and shared memory. The inherent algorithm dependencies still require extensive communication of the values of H, F, and E between the threads.

	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	5	0	0	0	0	0	0	0	?			
A	0	0	5	2	0	0	0	0	0	?				
U	0	0	0	2	0	0	5	0	?					
G	0	0	0	5	0	0	0	?						
C	0	5	0	0	10	5	?							
C	0	5	2	0	5	?								
A	0	0	10	1	?									
U	0	0	1	?										
U	0	0	?											
G	0	?												
C	0													
C	0													
G	0													
G	0													

Figure 4.3. Anti-diagonal method and dependency of the cells

4.3.3 Parallel Scan Smith-Waterman Algorithm

The NVIDIA G200 series require a minimum of roughly 30K active threads for 100% occupancy in order to perform efficiently. This means the diagonal algorithm will only perform efficiently on a single GPU if the minimum of the two sequence lengths is at least 30K. In biological applications test sequence lengths of this size are rare, but this is not the real problem with the diagonal algorithm. The primary issue with the diagonal approach is how it accesses memory. Any hardware which can accelerate scientific computations is necessarily effective because it accelerates memory accesses. On the GPU, memory access is enhanced in hardware by using

memory banks and memory streaming. On the GPU this means that it is very efficient to access up to 32 consecutive memory locations, and relatively ($5 - 10\times$ slower) inefficient to access random memory locations such as those dispersed along the anti-diagonal (and its neighbors).

To circumvent this problem it is shown below how the Smith-Waterman algorithm can be reformulated so that the calculations can be performed in parallel one row (or column) at a time. Row (or column) calculations allow the GPU memory accesses to be consecutive and therefore fast. To create a parallel row-access Smith-Waterman algorithm, the typical algorithm (Eqn. 4.2) is decomposed into three parts. The first part neglects the influence of the row sum, $E_{i,j}$ and calculates a temporary variable $\widetilde{H}_{i,j}$.

$$\begin{aligned} F_{i,j} &= \max(F_{i-1,j}, H_{i-1,j} - G_s) - G_e \\ \widetilde{H}_{i,j} &= \max(H_{i-1,j-1} + S_{i,j}, F_{i,j}, 0) \end{aligned} \tag{4.3}$$

This calculation depends only on data in the row above, and each item in the current row can therefore be calculated independently and in parallel. It is then necessary to calculate the row sums.

$$\widetilde{E}_{i,j} = \max(\widetilde{H}_{i,j-k} - kG_e) \quad 1 < k < j \tag{4.4}$$

These row sums are performed on $\widetilde{H}_{i,j}$ not $H_{i,j}$, since the later is not yet available anywhere on the row. The key to the reformulated algorithm is noting that these row sums look dependent and inherently serial, but in fact they can be computed rapidly in parallel using a variation of a parallel maximum scan. Finally we compute the true $H_{i,j}$ values via the simple formula.

$$H_{i,j} = \max(\widetilde{H}_{i,j}, \widetilde{E}_{i,j} - G_s) \tag{4.5}$$

It was shown in [49] that this decomposition is mathematically equivalent to Eqn. 4.2. The storage requirements are the same. Each step of this algorithm is completely parallel, and even more importantly for the GPU, has a contiguous memory access pattern. Figure 4.4 shows a graphical representation of the row-access parallel Smith-Waterman algorithm.

		0	**C**	**A**	**G**	**C**	**C**	**U**	**C**	**G**	**C**	**U**	**U**	**A**	**G**
H	**C**	*0*	5	0	0	10	5	0	5	0	10	1	0	0	0
F		*0*	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4
$\tilde{H}$	**C**	*0*	5	2	0	5	15	2	5	2	5	7	0	0	0
$\tilde{E}$		*0*	-1	4	3	2	4	14	13	12	11	10	9	8	7
H		*0*	5	2	0	5	15	6	5	4	5	7	1	0	0
F		*0*	-5	-5	-4	-5	-5	-5	-5	-4	-5	-5	-5	-5	-4

Figure 4.4. Graphical representation of row-access for each step in row-parallel scan Smith-Waterman algorithm. This example is calculating the 6^{th} row from the 5^{th} row (of the example problem shown in Figure 4.1).

4.3.4 Overlapping Search

If an upper bound is known a priori for the length of the alignment sub-sequences that will result from the search, it is possible to start the Smith-Waterman search algorithm somewhere in the middle of the table. This allows an entirely different approach to parallelism. Figure 4.5 shows an example where the database sequence has been broken into 4 overlapping parts. If the region of overlap is wider than any sub-sequence that is found then each part of the database can be searched independently. The algorithm will start each of the 4 searches with zeros in the left column. This is correct for the first section, but not for the other sections. Nevertheless, by the time

that the end of the overlap region has been reached, the table values computed by the other sections will be the correct values.

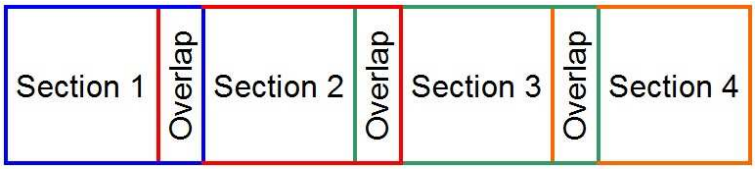


Figure 4.5. Example of the overlapping approach to parallelism.

This means that in the overlap region the values computed from the section on the overlap region's left are the correct values. The values computed at the beginning of the section that extends to the right are incorrect and are ignored (in the overlap region) for the purposes of determining best alignment sub-sequences.

This approach to parallelism is attractive for partitioning the problem onto the different GPUs. It does not require any communication at all between the GPUs except a transfer of the overlapping part of the database sequence at the beginning of the computation, and a collection (and sorting) of the best alignments at the very end. This approach also partitions a large database into much more easily handled/accessed database sections. The cost to be paid lies in the duplicate computation that is occurring in the overlap regions, and the fact that the amount of overlap must be estimated before the computation occurs.

For a 10^8 long database on 100 GPUs, each GPU handles a database section of a million items. If the expected subsequence alignments are less than 10^4 long the overlap is less than 1% of the total computation. However, if we were to try to extend this approach to produce parallelism at the block level (with 100 blocks on each GPU) the amount of overlap being computed would be 100% which would be quite inefficient. Thread level parallelism using this approach (assuming 128 threads per block and no more than 10% overlap) would require that sub-sequences be guaranteed to be less than eight items long. This is far too small, so thread level parallelism is

not possible using overlapping. The approach used in this chapter, which only uses overlapping at the GPU level, can efficiently handle potential sub-sequence alignments that are comparable to the size of the database section being computed on each GPU (sizes up to 10^6 in out current example). For larger alignments, the GPU parallel scan would be necessary.

4.3.5 Data Packing

Because the Smith-Waterman algorithm is memory constrained, one simple method to improve the performance significantly is to pack the sequence data and the intermediate table values. For example, when working with DNA or RNA sequences, there are only 4 possibilities (2 bits) for each item in the sequence. 16 RNA items can therefore be stored in a single 32-bit integer, increasing the performance of the memory reads to the sequence data by a factor of 16. Similarly for protein problems 5 bits are sufficient to represent one amino acid resulting in 6 amino acids per 32-bit integer read (or write).

The bits required for the table values are limited by the maximum expected length of the sub-sequence alignments and by the similarity scoring system used for a particular problem. The SSCA#1 benchmark sets the similarity value to 5 for a match, so the maximum table values will be less than or equal to 5×(the alignment length). 16 bits are therefore sufficient for sequences of length up to 10^4.

Data packing is problem dependent, so we have not implemented it in our code distribution [50], or used it to improve the speed of the results presented below.

4.3.6 Hash Table

Some commonly used Smith-Waterman derivatives (such as FASTA and BLAST) use heuristics to improve the search speed. One common heuristic is to insist upon perfect matching of the first n items before a sub-sequence is evaluated as a potential possibility for an optimal alignment. This and some preprocessing of the database

allows the first n items of each query sequence to be used as a hash table to reduce the search space.

For example, if the first 4 items are required to match perfectly, then there are 16 possibilities for the first 4 items. The database is then preprocessed into a linked list with 16 heads. Each linked list head points to the first location in the database of that particular 4 item string, and each location in the database points to the next location of that same type.

In theory this approach can reduce the computational work by roughly a factor of 2^n. It also eliminates the optimality of the Smith-Waterman algorithm. If the first 8 items of a 64 item final alignment must match exactly there is roughly a 12.5% chance that the alignment found is actually suboptimal. On a single CPU, the BLAST and FASTA programs typically go about 60× faster than optimal Smith-Waterman.

Because this type of optimization is very problem dependent and is difficult to parallelize, we have not included this type of optimization in our code or the following results.

4.4 Literature Review

There are already a number of efficient implementations of the Smith-Waterman (SW) algorithm on different CPU architectures [51, 52, 53, 54, 55, 56] and GPU architectures [38, 39, 57, 58, 59, 60, 61, 62, 63].

Liu et al. [57] have implemented the SW algorithm on the GPU on the OpenGL environment. They achieved a speedup of sixteen over available straightforward and optimized CPU SW implementation. The basic idea is to compute the dynamic programming matrix in anti-diagonal order. All elements in the same anti-diagonal of the Smith-Waterman matrix can be computed independent of each other in parallel. The anti-diagonals are stored as textures in the texture memory. Fragment programs are then used to apply the arithmetic operations specified by the recurrence relation.

The main problem with this approach is how to access the memory (read and write). For calculating one diagonal two previous diagonals are required. In order to keep the method memory efficient complicated memory management is needed.

Liu et al. [58] implemented double affine SW (DASW) algorithm which compares and aligns genomic sequences such as DNA and proteins. They perform their calculation with the NVIDIA GeForce 7800 GTX graphics card using the OpenGL API, and the GL shading language (GLSL). Their implementation involved two stages from the OpenGL rendering pipeline: geometry transformation and fragment rasterization. The geometry acts as the proxy that arranges the pipeline for the dynamic programming computation and sets the area of computation. In the beginning query and database sequences are copied to the texture memory of the GPU. After this step, the geometry transformation stage for each diagonal is passed a quadrilateral that can keep the dynamic programming cells of the diagonal. A unique texture coordinate address is set to each fragment from the quadrilateral. The resulting pixel values are saved into an image buffer in the GPU texture memory, to be reused in next passes. They precede this computation loop until all diagonals have been updated. They have achieved 0.25 giga cell updated per second (GCUPS) for their implementation.

Liu et al. [59] introduced two streaming algorithms for dynamic programming-based biological sequence alignment that efficiently mapped onto the graphics hardware. They reformulated their algorithms as streaming algorithms that are efficient when implemented on graphics processing unit. Their experimental results show that the dynamic programming-based approach obtains speedups of more than one order of magnitude compared with optimized CPU implementations.

Manavski and Valle [60] also implemented the SW algorithm on single and dual GPUs. They are achieved more than 1.8 GCUPS for single GPU and 3.5 GCUPS for two GeForce 8800 GTX. The strategy adopted in their implementation in CUDA was to make each thread in the GPU perform the whole alignment of the query sequence

with one database sequence. They sorted the database based on their length in order to make each thread in the block compute same amount of work for query sequences. So when each block launched, the threads in the block will need to align the test sequence with two database queries having the closest possible sizes. Because there are so many query sequences in the database, the whole approach is inherently parallel. But it is still necessary to be careful about the memory access pattern.

Liu et al. [61] implemented SW sequence database searches in single GTX 280 and dual-GPU GTX 295 graphics cards. Their code can support query sequences of length up to 59K that is longer than the maximum sequence length 35,213 in Swiss-Prot release 56.6. The query length is limited in their application because of constant memory in the GPU (64K). For the single-GPU version, they achieved consistent performance for query sequences of length varying from 144 to 5,478, where the performance figures vary from a low of 9.039 GCUPS to a high of 9.660 GCUPS, with an average performance of 9.509 GCUPS. They also executed their code with multi-GPU and obtained better performance. With increasing the sequences length they attained better performance from a low of 10.660 GCUPS to a high of 16.087 GCUPS, with an average performance of 10.660 GCUPS.

Striemer et al. [62] and Akoglu et al. [63] also applied SW algorithm on the GPU. Like other works, they have used a single thread block to find best sequence for a single sequence in the database. They have done whole sequence matchings on a single GPU and void to using the CPU. They have achieved an average speedup of 9 comparing with SSEARCH software and same order of speedup comparing with Farrar [53] (efficient CPU implementation code).

Ligowski and Rudnicki [39] were executed SW algorithm on CPU and GPU. The control loop resides on CPU. Program loads sorted database to the GPU memory, and waits for queries. Each thread in the block compares a query sequence with one of the database sequences. The query sequence is shared by all threads in the block.

Since all threads in the block are synchronized, the length of the database sequences processed by a single block should be close to each other in the same block. Otherwise all threads in the block have to wait for the one searching alignment of the longest database sequence. All calculations within the loop are performed in fast shared memory and registers. The bandwidth is limited by availability of shared memory. Based on this approach they achieved 6 GCUPS for a single GPU and 12 GCUPS for dual GPUs with 9800 GX2.

Liu et al. [38] optimized their Smith-Waterman code (CUDASW++). They investigated a partitioned vectorized SW algorithm using CUDA based on the virtualized SIMD abstraction of the GPUs. They found out the optimized single-instruction multi-thread (SIMT) and the partitioned vectorized algorithms have similar performance characteristics when benchmarked by searching the Swiss-Prot release 56.6 database. They also illustrated that the optimized SIMT algorithm produced reasonably stable performance, while the partitioned vectorized algorithm sometimes has small fluctuations around the average performance for some gap penalties. Fluctuations increase with the higher gap open and gap extension penalties. On the other hand, CUDASW++ 2.0 supports multiple GPU devices installed in a single host. It has high performance comparing with CUDASW++ 1.0 using either the optimized SIMT algorithm or the partitioned vectorized algorithm. They achieved the highest performance of 17 GCUPS on GTX 280 and 30 GCUPS on GTX 295. Although the optimal alignment scores of the Smith-Waterman algorithm can be used to find similar sequences, the scores are changed by sequence length and composition.

4.5 Results

Timings were performed on 4 different GPU architectures and a high end CPU. The NVIDIA 9800 GT has 112 cores and its memory bandwidth is 57.6 GB/s. The NVIDIA GTX 9800 has 128 cores and its memory bandwidth is 70.4 GB/s. The

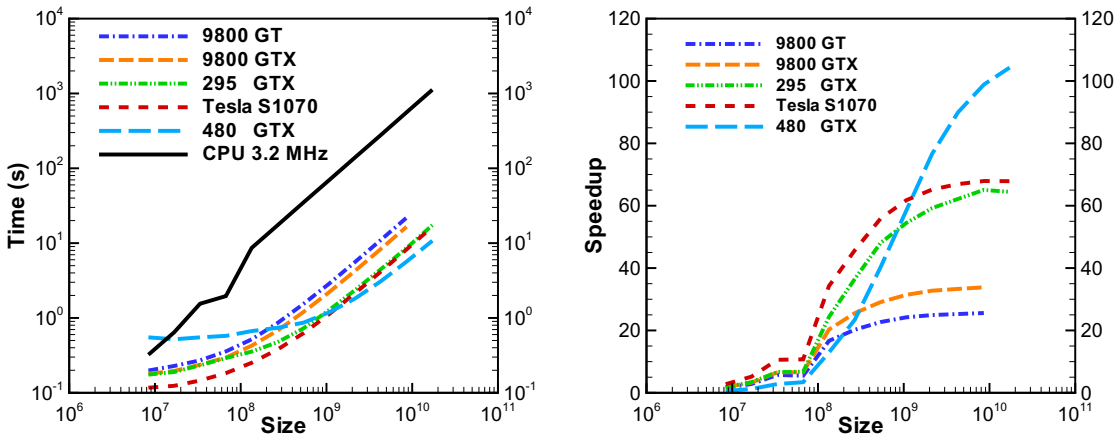
newer generation NVIDIA GTX 295 comes as two GPU cards that share a PCI-e slot. When we refer to a single GTX 295 we refer to one of these two cards, which has 240 cores and a memory bandwidth of 111.9 GB/s. The CPU is an AMD quad-core Phenom II X4, operating at 3.2 GHz, it has 4×512 KB of L2 cache and 6 MB of L3 cache. For the parallel timings a PC that contains 4 NVIDIA GTX 295 cards (occupying 2 PCI-e $\times 16$ slots) was used. All code was written in C++ with NVIDIA's CUDA language extensions for the GPU, and compiled using Microsoft Visual Studio 2005 (VS 8) under Windows XP. The bulk of NVIDIA SDK examples use this configuration.

The SSCA#1 benchmark was developed as a part of the High Productivity Computer Systems (HPCS) [64] benchmark suite that was designed to evaluate the true performance capabilities of supercomputers. It consists of 5 kernels that are various permutations of the Smith-Waterman algorithm.

4.5.1 Single GPU

In the first kernel the Smith-Waterman table is computed and the largest table values (200 of them) and their locations in the table are saved. These are the end points of well aligned sequences, but the sequences themselves are not constructed or saved. The sequence construction is performed in kernel 2. Because the traceback step is not performed in kernel 1 only a minimal amount of data in the table needs to be stored at any time. For example, in the row-parallel version, only the data from the previous row needs to be retained. This means that kernel 1 is both memory efficient and has a high degree of fine scale parallelism. Single processor timings and speedup for kernel 1 are shown in figure 4.6(a) and 4.6(b) for different table sizes and different processors. The table size is the length of database multiplied by the length of the test sequence (which was always 1024 for the kernel 1 tests). For smaller size problems, the overhead associated with saving and keeping 200 best matches becomes

a significant burden for the GPUs and their efficiency decreases. For a finely parallel (mostly SIMT) algorithm (kernel 1) the latest GPU (GTX 480) is over 100 times faster than a single core of the latest CPU. This performance benefit might be reduced to roughly 25× faster if all four cores of the (Phenom II) CPU were put to use. But the factor of 100× can also be regained by putting 4 GPUs on the motherboard. Many motherboards now have two (or even four) PCI-e ×16 slots.

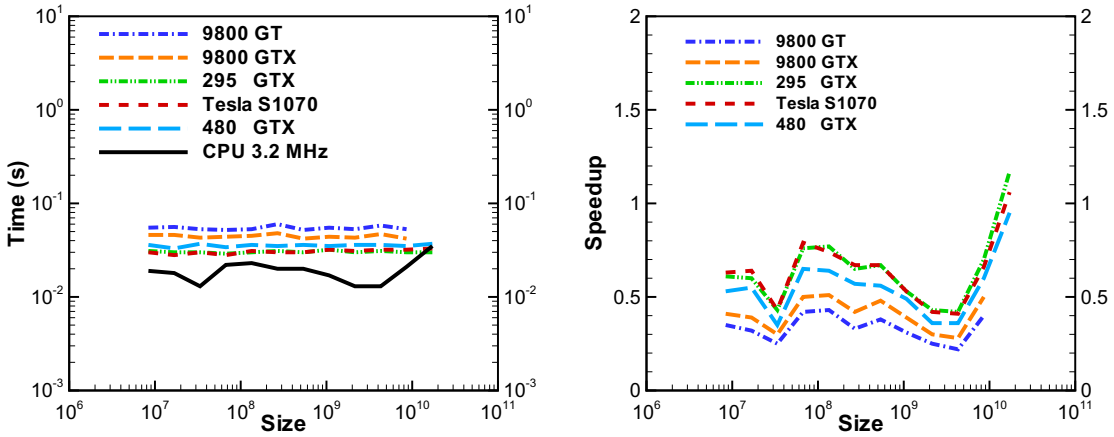


(a) (b)

Figure 4.6. (a) Time and (b) speedup for kernel 1, for different problem sizes and different processors

The second kernel in the SSCA#1 benchmark is very fast compared to kernel 1. In the second kernel, subsequences are constructed from the (kernel 1) 200 endpoint locations. Since the table has not been stored during the first kernel this means small parts of the table need to be recomputed. This reconstruction happens by applying the Smith-Waterman 'backwards', since the algorithm works just as well either direction. Starting at the end point location the corresponding row and column are zeroed. The algorithm is now applied moving upwards and to the left (rather than downwards and to the right). When the endpoint value is obtained in the table, this is the starting point of the sequence. The traceback process then occurs by tracing

to the right and down. In our GPU implementation groups of 64 threads work on each of these small sub-tables, and the sub-tables are all computed in parallel. Single processor timings and speedup for kernel 2 are shown in figure 4.7(a) and 4.7(b) for different table sizes and different processors.



(a) (b)

Figure 4.7. (a) Time and (b) speedup for kernel 2, for different problem sizes and different processors

The traceback procedure is an inherently serial process and difficult to implement well on the GPU. A GPU operates by assigning threads into a single-instruction, multiple-thread (SIMT) group called a block. In kernel 2 a single block is assigned to compute the small sub-table for each endpoint. 64 threads are assigned to each block (32 is the minimum possible on the GPU), so the sub-table is constructed in 64×64 chunks. In SSCA#1, a single 64×64 block is almost always sufficient to capture the alignment sequence, but it is almost always wasteful as well. The 64 threads assigned to a sub-table use the anti-diagonal method to construct each 64×64 chunk. As expected the performance is not a function of the problem size. Kernel 2 always works on 200 sequences. It does not matter how large the table is that they originally came from.

Timings and speedup for kernel 3 of the SSCA#1 benchmark are shown in figures 4.8(a) and 4.8(b). Kernel 3 is similar to kernel 1 in that it is another Smith-Waterman calculation. However in this case the sequences are required to be computed at the same time (in a single pass) as the table evaluation. The table is, of course, too large to store in its entirety as it is calculated, so only 'enough' of the table to perform a traceback is kept. Kernel 3 therefore requires the maximum possible subsequence length to be known beforehand.

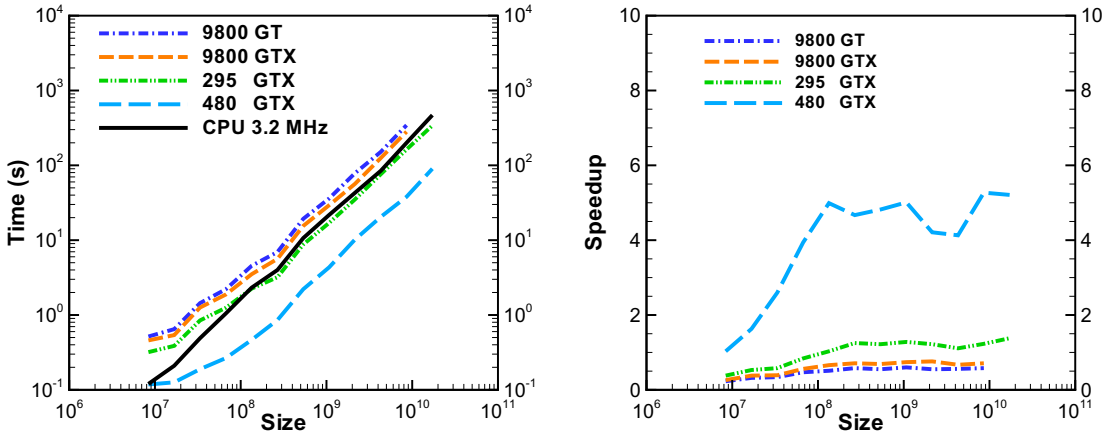


Figure 4.8. (a) Time and (b) speedup for kernel 3, for different problem sizes and different processors

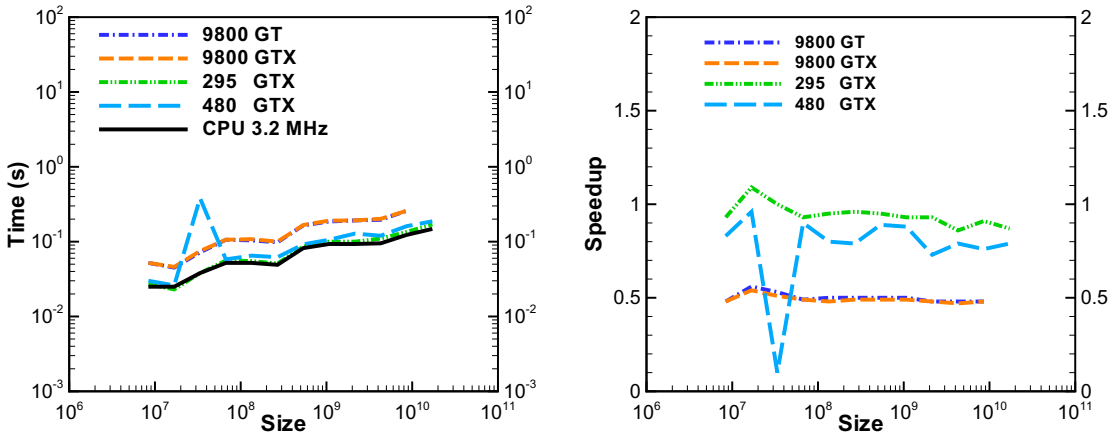
In SSCA#1 this is possible because the test sequences in kernel 3 are the 100 most interesting subsequences (100 highest scoring sequences) found in kernel 2. In kernel 3, the 100 best matches for each of the 100 test sequences (10,000 matches) must be computed. By assigning groups of threads into a SIMT block, a single block is assigned to find 100 best matches for one test sequence. The results of kernel 3 are 100×100 subsequences. Kernel 3 therefore has coarse grained parallelism that might be used to make it faster than kernel 1 on general purpose hardware. However, the kernel 3 algorithm requires different threads and different thread blocks to be doing

different things at the same time, which is ill-suited to the GPU (essentially SIMT) architecture. The performance of the GPUs for this kernel is therefore roughly the same as one core of a (large cache) CPU.

The fourth kernel goes through each set of matches (100×100 best matches) found by the third kernel, and performs a global pairwise alignment for each pair. The output of this kernel is $100 \times 100 \times (100-1)/2$ global alignment scores. The global alignment of two (same length) subsequences is given by equation 4.6 below. The global similarity, dissimilarity and gap penalty (given in Eqn. 4.6) are dictated by SSCA#1 but could be arbitrary.

$$H_{i,j} = \min \begin{cases} H_{i-1,j-1} & +0 \quad A_j = B_i \\ H_{i-1,j-1} & +1 \quad A_j \neq B_i \\ H_{i-1,j} & +2 \\ H_{i,j-1} & +2 \end{cases} \tag{4.6}$$

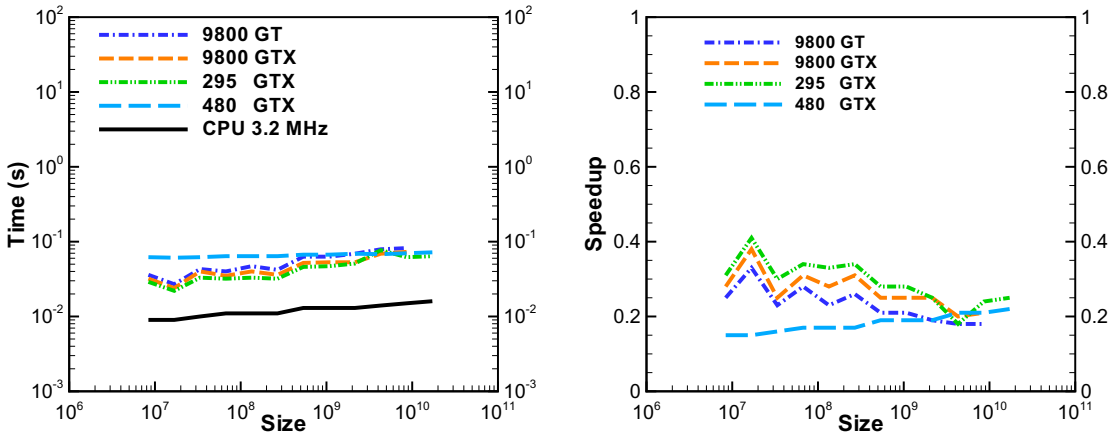
Global alignment is also performed by creating a table. The global pairwise alignment score of two subsequences is the number in the right bottom corner of table. The table is started with $H_{i,0} = i \times 2,\ H_{0,j} = j \times 2$. On the GPU, each thread performs a single global pairwise alignment for each pair. Every table construction is completely independent and so this kernel is highly parallel and very fast. Figures 4.9(a) and 4.9(b) show the timing and speedup for this kernel. Kernel 4 is plotted on a log scale of the problem size because the sequence lengths get slightly longer as the database gets larger (due to the probability of extreme events increasing)



(a) (b)

Figure 4.9. (a) Time and (b) speedup for kernel 4, for different problem sizes and different processors

The fifth kernel performs multiple sequence alignment on each of the sets of alignments generated by the third kernel, using the simple, approximate center star algorithm [47]. The center sequence is chosen based on maximum score obtained in the fourth kernel. Each block performs single multiple sequence alignment per sets of alignments generated by the third kernel. Timing and speedup for this kernel are shown in figure 4.10(a) and 4.10(b) for different problem sizes respectively. This kernel does not map well to the GPU but is very fast nonetheless. It involves 10,000 independent tasks. Kernel 5 is plotted on a log scale of the problem size too.



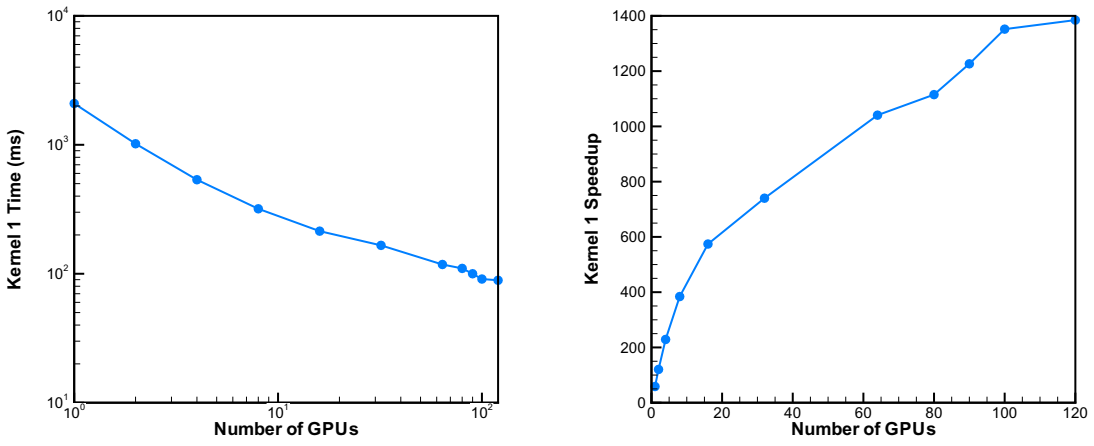
(a) (b)

Figure 4.10. (a) Time and (b) speedup for kernel 5, for different problem sizes and different processors

Figures 4.6(b) and 4.8(b) show that the performance of the GPU tends to be strongly problem size dependent. Like a vector supercomputer from the 1990's the startup costs are high for processing a vector. On a GPU the highest efficiencies are found when computing 10^5 or more elements at a time. Because the test sequence is 1024 a problem size of 10^9 translates into an entire row of (10^6) database elements being processed during each pass of the algorithm. These figures also show that the relative performance of the GPU cards on computally intensive problems is proportional to their hardware capabilities (number of cores and memory bandwidth) irrespective of whether the algorithm works efficiently on the GPU. It is probably the memory bandwidth that dictates the performance for the memory bound Smith-Waterman problem, but this is difficult to prove since the GPU architectures tend to scale up the number of cores and bandwidth simultaneously. Of course, the constant of proportionality is a strong function of the algorithm structure and the problem size.

4.5.2 Strong Scaling on Many GPUs

When the problem size is held constant and the number of processors is varied we obtain strong scaling results. Figures 4.11(a) and 4.11(b) show the timing speedup obtained when a 16M long database is searched with a 128 long query sequence on Lincoln, respectively. Ideally, the speedup in this case should be linear with the number of GPUs. In practice, as the number of GPUs increases, the problem size per GPU decreases, and small problem sizes are less efficient on the GPU. When 120 GPUs are working on this problem, each GPU is only constructing a sub-table of roughly size 10^7. From figure 4.6(b) it is clear that this size is an order of magnitude too small to be reasonably efficient on the GPU.



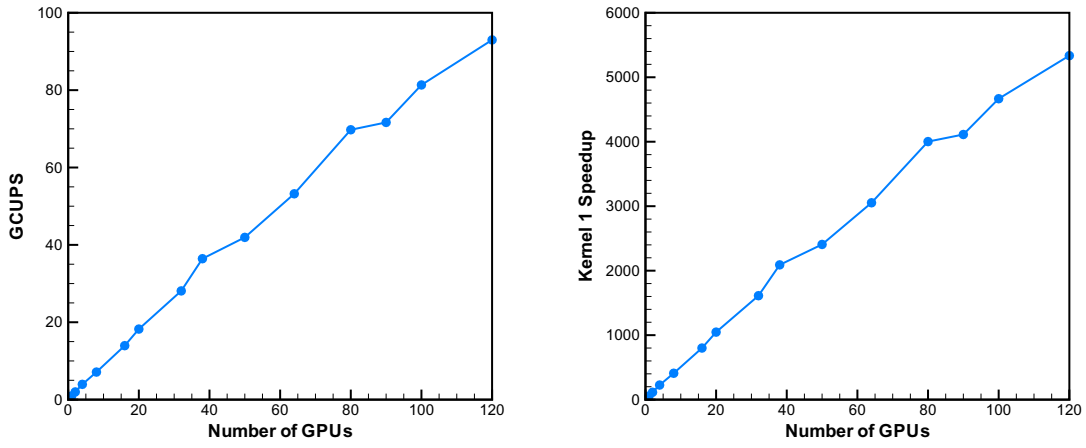
(a) (b)

Figure 4.11. Strong scaling timings with 16M for database and 128 for test sequence (a) Time (b) Speedup verses one core of a 3.2 GHz AMD quad-core Phenom II X4 CPU

4.5.3 Weak Scaling on Many GPUs

Weak scaling allows the problem size to grow proportionally with the number of GPUs so that the work per GPU remains constant. The GCUPS for this weak scaled problem are shown in figure 4.12(a), when 2M elements per GPU are used for

the database size, and a 128-element query sequence is used. Since the amount of work being performed increases proportionally with the number of GPUs used the GCUPS should be linear as the number of GPUs is increased. However, the time increases slightly because of the increased communication burden between GPUs as their number is increased. The total time varies from $270ms$ to $350ms$, and it increases fairly slowly with the number of GPUs used. Figure 4.12(b) shows speedups for Kernel 1 on the Lincoln supercomputer compared to a single-CPU core of an AMD quad-core. Compared to a single CPU core, the speedup is almost 56× for a single GPU, and 5335× for 120 GPUs (44× faster per GPU).



(a) (b)

Figure 4.12. Weak scaling GCUPS (a) and speedups (b) for Kernel 1 using various numbers of GPUs on Lincoln with 2M elements per GPU for the database size, and a 128-element test sequence

The scan is the most expensive part of the three steps. This scan was adapted from an SDK distribution that accounts for bank conflicts and is probably performing at close to the peak possible efficiency. If a single Smith-Waterman problem is being solved on the GPU, then the problem is inherently coupled. The row-parallel algorithm isolates the dependencies in the Smith-Waterman problem as much as is

possible, and solves for those dependencies as efficiently as possible, using the parallel scan.

The maximum memory bandwidth that can be achieved on Lincoln is close to 85 GB/s for a single GPU. Based on our algorithm (7 memory accesses per cell) the maximum theoretical GCUPS is close to 3.0 (85/4/7) for a single GPU. Since this code obtains 1.04 GCUPS per GPU, the algorithm is reasonably efficient (34% of the peak memory throughput). Higher GCUPS rates for a single GPU would be possible by using problem dependent optimizations such as data packing or hash tables.

The row parallel algorithm doesn't have any limitation on the query or database sequence sizes. For example, a database length of 4M per GPU was compared with a query sequence of 1M. This query took 4229 seconds on Lincoln and 2748 seconds on the GTX 480. This works out to 1.03 and 1.6 GCUPS per GPU for the Tesla S1070 and GTX 480 GPUs respectively. This is very similar to the performance obtained for smaller test sequences, because the algorithm works row by row and is therefore independent of the number of rows (which equals the query or test sequence size).

CHAPTER 5

HIGH PERFORMANCE COMPUTING ON THE 64-CORE TILERA PROCESSOR

5.1 Introduction

Processors with a large number of on chip cores are now becoming commonly commercially available. Tilera processors have been available since late 2007. They currently come with 64 cores, and 100 cores are announced for late 2011. Intel released a 48-core prototype to University researchers in April 2011, and AMD has been selling a 12-core Opteron since that time. One important aspect of these many-core designs is their low power consumption per core. The Intel chip reportedly draws no more than 125 W for all 48 cores. The Tilera-64 draws only about 50 W for 64 cores. Since power consumption, and the resultant cooling costs, can dominate supercomputer costs it is some interest to evaluate the potential of these many core chips for typical supercomputing applications. Our evaluation of these many-core architectures will focus on scalability and performance per Watt when applied to scientific computing applications.

The Tilera and Intel marketing plans are focused on commercial server farms, not supercomputers for scientific computing. However, high performance computing (HPC) is one potential application for these processors. GPUs provide similar computing benefits and are also now being incorporated into recent supercomputer designs. This project will evaluate the potential performance attributes of many-core processors on a few select and hopefully reasonably representative scientific computing benchmarks.

In 2007, Tilera launched its first many-core architecture with 64-core on a single chip. The main goals in Tilera architecture are to provide high performance cores that communicating via cache-coherent iMesh interconnect network architecture and low power hardware [65]. Figure 5.1 shows Tile processor hardware architecture with detail of an individual tile's structure.

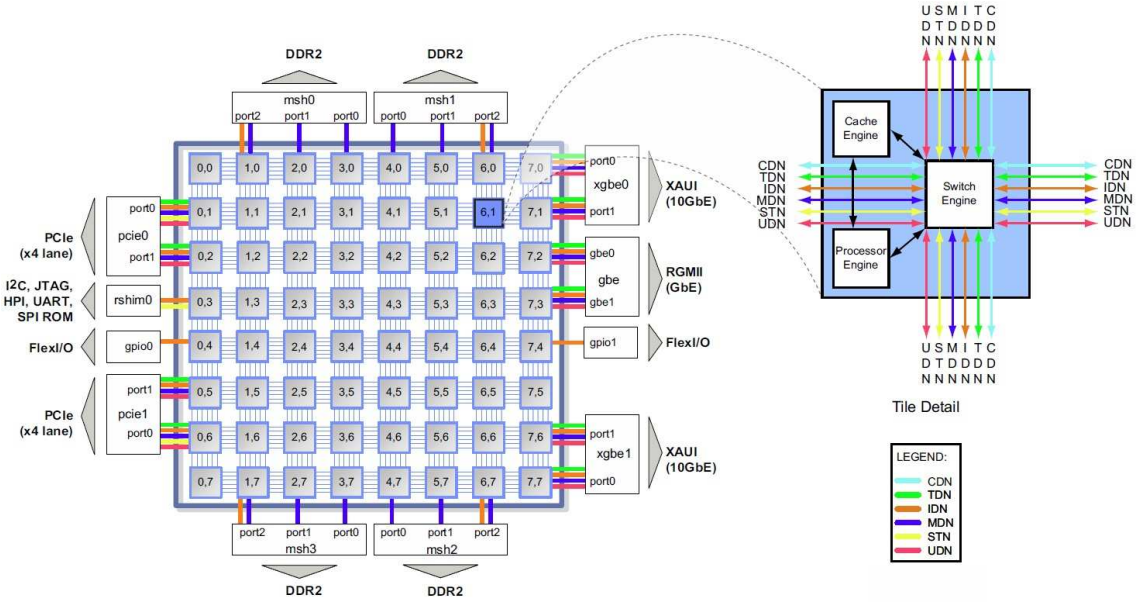


Figure 5.1. Tile processor hardware architecture with detail of an individual tile's structure (Figure from Tilera data sheet [10])

The iMesh interconnect architecture provides high bandwidth and low latency communication between tiles. Each tile is a powerful, full-featured computing system that can independently run an entire symmetric multi-processing operating system. Each tile operates at 866 MHz and implements a 32-bit integer processor engine utilizing a three-way Very Long Instruction Word (VLIW) architecture with 64-bit instruction bundle. Each tile has 16K L1 instruction cache, 8K L1 data cache and unified 64K L2 cache. Also there is 4MByte L3 cache distributed across tiles. There are four on-chip-memory controllers, each supporting 64-bit DDR2 DRAM (with optional ECC protection) and operating at 6.4 GB/s, for a peak memory bandwidth

of 25.6 GB/s. Tilera also has two full-duplex XAUI-based 10Gb Ethernet, two 4-lane PCI Express ports and two onboard 10/100/1000 Ethernet MACs with RGMII (Reduced Gigabit Media Independent Interface) interfaces [10].

There are number of previous benchmark implementations on the Tilera [66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77].

Bornstein et al. [72] implemented image analysis onboard a Mars rover. But Tilera processors don't support floating point form hardware so all floating point operations are emulated in software that causes reduced performance for Tilera. They reported that a conversion from floating point input data to integer increased the speed by a factor of 5. When using many cores they achieved 8 and 9.7 times the speed of a single tile (single core) when using 16 and 32 tiles respectively.

Ha et al. [73] studied the scalability problem for dynamic analysis on TILE64 processor. They obtained a benefit from using the fast inter-core communication between tiles for dynamic analysis.

Berezecki et al. [74] implemented key-value store Memcached on TILEPro64 and compared results with a 4-core Intel Xeon L5520 and an 8-core AMD Opteron 6128 HE. They achieved 5 times speedup compared to a single core of CPUs and almost 2 times speed up compared to all the cores of CPUs. They also reported that the TILEPro had 3 to 4 times better performance/Watt compared to the CPUs.

Yan et al. [75] implemented the accelerated deblocking filter of the H.264/AVC. They achieved an overall decoding speedup of 1.5 and 2 times for the HD and SD videos.

Richardson et al. [76] implemented some space applications on TILEPro64 processor. They achieved 23 times speedup compared to the single tile when 32 tiles were used for the sum of the absolute difference. They also reported linear speedup up to 8 tiles.

Ulmer et al. [77] applied a text document similarity benchmark based on the Term Frequency Inverse Document metric on the Tilera and an FPGA and compared results with sequential CPU runs. They achieved 4 times speedup for the Tilera when compared to the single core of 2.2 GHz x86 processor.

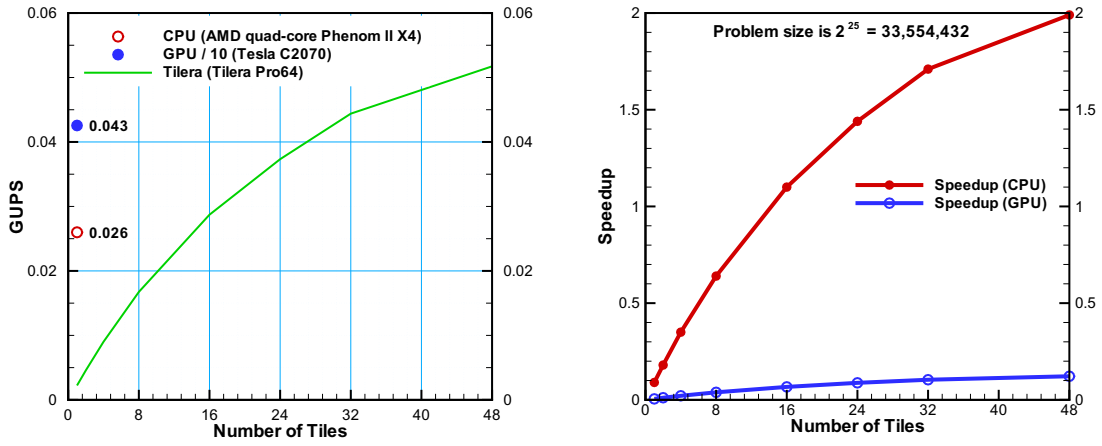
In this chapter we are interested in HPC benchmarks related to scientific computations and the scalability, performance, and power consumption of the Tilera processor. The benchmark cases are: GUPS, large unstructured sparse matrix multiply the Smith-Waterman algorithm (SSCA#1 kernels 1-2) and 3D FFT.

5.2 Giga Update Per Seconds (GUPS)

The GUPS benchmark [78] was ported to the Tilera. The main goal in this benchmark is to test the random memory access speed of the hardware. The CPU and GPU version of the GUPS benchmark were also used in order to compare the Tilera results with other known architectures. The computations were iterated 10 times in order to get accurate timings. The results presented herein are based on the average of the 10 iterations.

5.2.1 Strong Scaling

Figure 5.2(a) and table 5.1 show the GUPS (giga-updates per second) for the strong scaling case (a constant problem size with varying numbers of tiles). Figure 5.2(b) shows the relative speedup of the Tilera compared to one core of an AMD quad-core Phenom II X4 (red line) and a Tesla C2070 GPU (blue line). This figure shows that the speedup is linear with the number of cores only up to 4 cores (because there are 4 memory channels on the Tilera). With larger numbers of cores, the speedup is below the linear ideal performance. With 48 cores active (out of a maximum of 63) the Tilera is roughly 2 times faster than one core of the CPU and 8 times slower than GPU (the GPU performance is divided by 10 to more easily place it on the graph).



(a) (b)

Figure 5.2. (a) GUPS vs. number of tiles (b) Speedup for GUPS benchmark compared to a single core of an AMD quad-core Phenom II X4 (red line) and compared to a Tesla C2070 GPU (blue line). This case uses a 2^{25} 64-bit integer table size (256 MB) with 2^{27} updates

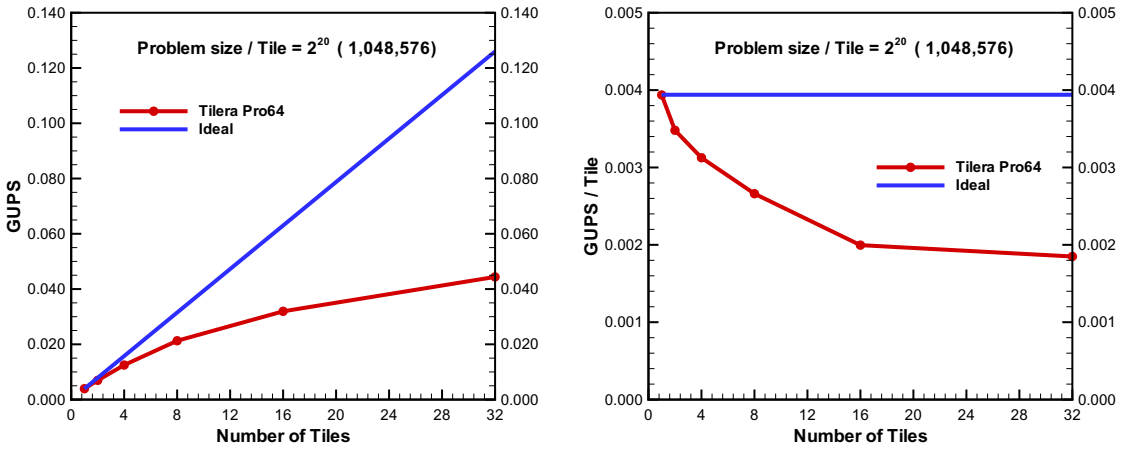
Because the Tilera code is parallel, there is a possibility of a read before write conflict. The Tilera's results were compared with the CPU results (which do not have this issue). The error rate was negligible and a low level of error is said to be acceptable in the GUPS benchmark specification.

Table 5.1. GUPS for strong scaling case with 2^{25} 64-bit integer unknowns (256 MB)

CPU	GPU	Tiles	TILERA	Speedup (CPU)	Speedup (GPU)
0.02599	0.42555	1	0.00223	0.09	0.01
		2	0.00458	0.18	0.01
		4	0.00901	0.35	0.02
		8	0.01674	0.64	0.04
		16	0.02872	1.10	0.07
		24	0.03733	1.44	0.09
		32	0.04439	1.71	0.10
		48	0.05173	1.99	0.12

5.2.2 Weak Scaling

Figure 5.3 shows the GUPS and GUPS/Tile for weak scaling (problem size per tile is kept constant at 1M). As the number of tiles increases the GUPS/Tile decreases. This occurs because the tiles must share the limited memory bandwidth. This contention is also what causes the deviation from an ideal linear speedup that is shown in figure 5.3(a).



(a) (b)

Figure 5.3. GUPS for weak scaling for the Tilera Pro64 (a) GUPS vs. number of tiles (b) GUPS/Tile vs. number of tiles

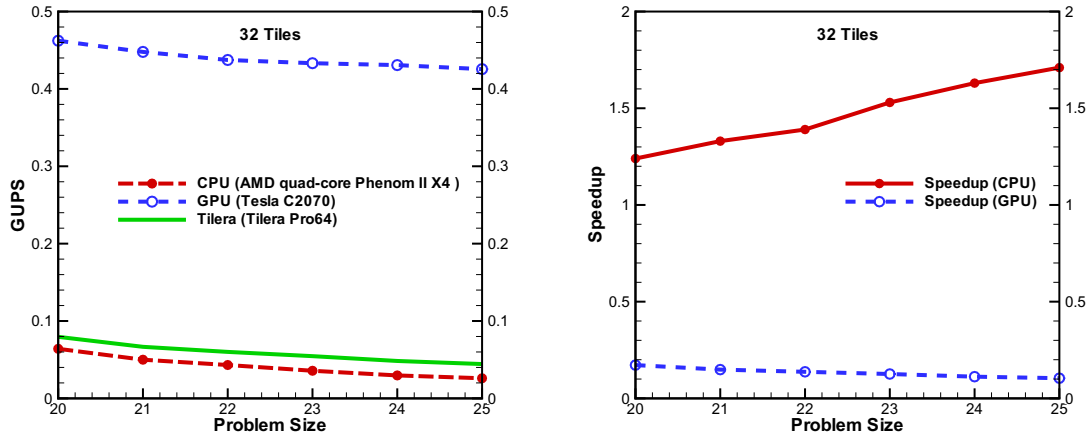
Table 5.2 shows the results for weak scaling for the CPU, GPU and Tilera. CPU results are just for a single core. The Tesla C2070 GPU is used to run the GUPS benchmark on the GPU. The CPU and GPU results are for the corresponding problem size (the number of tiles is irrelevant for those processors). The results for the week scaling are essentially the same as the strong scaling

Table 5.2. Weak scaling for GUPS benchmark. Problem size is 2^{20}/Tile. The largest problem size uses 32 tiles and the smallest size uses 1 tile.

Problem Size	CPU	GPU	TILERA	Speedup (CPU)	Speedup (GPU)
2^{20}	0.06409	0.46228	0.00394	0.06	0.009
2^{21}	0.04999	0.44782	0.00696	0.14	0.016
2^{22}	0.04313	0.43739	0.01250	0.29	0.029
2^{23}	0.03571	0.43324	0.02129	0.60	0.049
2^{24}	0.02968	0.43077	0.03194	1.08	0.074
2^{25}	0.02599	0.42555	0.04439	1.71	0.104

5.2.3 Performance Variation

Figure 5.4 shows the performance of 32 cores of the Tilera (63 cores are almost the same speed) compared to one core of a CPU and compared with a GPU as the problem size varies. Note that all the processors loose efficiency with increasing the problem size. This occurs because all the processors get some small advantage by using the L1 and L2 caches for small problem sizes.



(a) (b)

Figure 5.4. (a) GUPS vs. problem size (b) Speedup for GUPS benchmark compared to a single core of an AMD quad-core Phenom II X4 (red line) and compared to a Tesla C2070 GPU (blue line) using 32 tiles on the Tilera Pro64

5.2.4 Power Consumption

Table 5.3 and figure 5.5 show the results for power consumption for the GUPS benchmark. The Tilera Pro64 uses 9 times less energy compared to a single core of the AMD quad-core Phenom II X4. But compared to the GPU, the energy consumption is roughly equal. The energy efficiency is given by,

$$EnergyEfficiency = \frac{Power_{(CPUorGPU)} \times Time_{(CPUorGPU)}}{Power_{(Tilera)} \times Time_{(Tilera)}} \tag{5.1}$$

A low efficiency is better as it implies less energy is used to complete a task. An efficiency less than one implies the CPU or GPU is more energy efficient than the Tilera.

Table 5.3. Strong scaling results for GUPS benchmark for Tilera Pro64 comparing the power consumption to the single core of an AMD quad-core Phenom II X4 and Tesla C2070 GPU

CPU Power (W)	GPU Power (W)	# of Tiles	Power (W)	Efficiency (CPU)	Efficiency (GPU)
69	120	1	2	3.1	0.3
		2	3	4.2	0.4
		4	4	6.2	0.6
		8	5	9.2	0.9
		16	10	7.9	0.8
		24	13	7.6	0.8
		32	15	8.1	0.8
		48	20	6.7	0.7

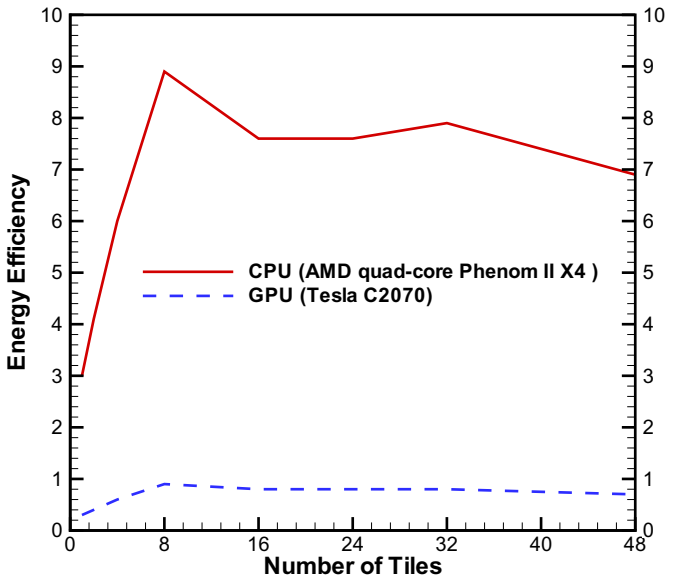


Figure 5.5. Power efficiency for GUPS benchmark compared to a single core of an AMD quad-core Phenom II X4 (red line) and compared to a Tesla C2070 GPU (blue line).

5.3 Sparse Vector-Matrix Multiplication

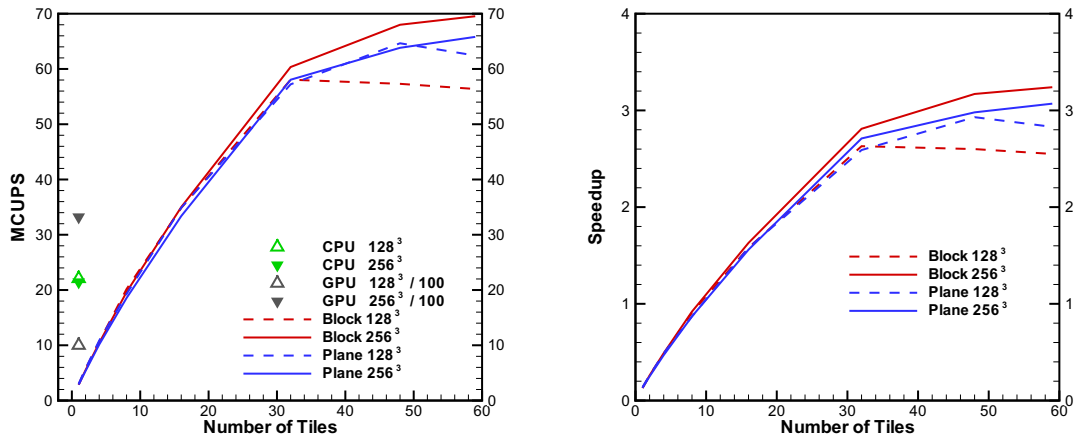
Sparse vector-matrix multiplication is a common random memory operation found in many high performance computing codes. The equation below shows the core formula for the 7-point-stencil sparse vector-matrix multiplication that represents the classic discrete 3D Laplacian operation.

$$D[i][j][k] = B[i][j][k] \times \left(\begin{array}{l} (A[i+1][j][k] + A[i-1][j][k]) \times dxic[i] \times dxiv[i] + \\ (A[i][j+1][k] + A[i][j-1][k]) \times dyic[j] \times dyiv[j] + \\ (A[i][j][k+1] + A[i][j][k-1]) \times dzic[k] \times dziv[k]) \end{array} \right) + A[i][j][k] \times C[i][j][k] \quad (5.2)$$

Two different algorithmic approaches on the Tilera were tested: plane marching and block marching. In the plane marching, every tile is responsible for a XY plane from the 3D domain. In the block marching approach, the 3D domain is divided into hexahedra subdomains of size $8 \times 8 \times NZ$, and every tile is responsible for computing a subdomain. This is similar to the GPU layout (which uses $16 \times 16 \times NZ$ subdomain for each GPU multiprocessor). Plane and block marching have different advantages. In the plane marching approach hardware can load the XY plane to its own cache (small problem size). In the block marching approach, hardware can load $z-1$ and z data items into its cache. We tested different block sizes and 8×8 is the most efficient block size for this approach on the Tilera Pro64 card.

5.3.1 Performance Results

Figure 5.6 shows the MCUPS (millions of cell updates) and speedup for $128 \times 128 \times 128$ and $256 \times 256 \times 256$ total problem sizes.



(a) (b)

Figure 5.6. (a) MCUPS for 128^3 and 256^3 problem sizes using a Tilera Pro64 (with different numbers of tiles), a single core of an AMD quad-core Phenom II X4, and a Tesla C2070 GPU. (b) Speedup of the Tilera versus one core of the AMD CPU

In order to show GPU result on the same graph, the MCUPS for the GPU is divided by 100. MCUPS are millions of cell updates (or results, D_{ijk}) computed per second. Each result formally requires reading 7 data values (A) and two constants (B and C) and performing a number of additions and multiplications.

For both algorithm approaches on the Tilera the performance is very similar. For small sizes, plane marching has slightly better performance than block marching. On the other hand, block marching has better performance than plane marching for large problems. A speedup of around 3× is obtained for sparse vector-matrix multiplication on very large matrices 256^3 when the whole Tilera is compared to 1 core of the AMD CPU.

Table 5.4 and 5.5 show the results for CPU and Tilera for $128 \times 128 \times 128$ and $256 \times 256 \times 256$ problem sizes respectively.

Table 5.4. MCUPS and speedup for $128 \times 128 \times 128$ problem size for Tilera Pro64 comparing to the single core of AMD quad-core Phenom II X4

128 x 128 x 128	Tilera Pro64				
CPU (MCUPS)	# of Tiles	Block (MCUPS)	Plane (MCUPS)	Speedup (Block)	Speedup (Plane)
22.07	1	2.91	3.04	0.13	0.14
	2	5.71	5.78	0.26	0.26
	4	10.65	10.85	0.48	0.49
	8	20.17	19.28	0.91	0.87
	16	34.85	34.88	1.58	1.58
	32	58.06	57.22	2.63	2.59
	48	57.31	64.65	2.60	2.93
	59	56.38	62.43	2.55	2.83

Table 5.5. MCUPS and speedup for 256 × 256 × 256 problem size for Tilera Pro64 comparing to the single core of AMD quad-core Phenom II X4

256 x 256 x 256	Tilera Pro64				
CPU (MCUPS)	# of Tiles	Block (MCUPS)	Plane (MCUPS)	Speedup (Block)	Speedup (Plane)
21.44	1	2.86	2.88	0.13	0.13
	2	5.50	5.35	0.26	0.25
	4	10.42	10.13	0.49	0.47
	8	19.62	18.55	0.92	0.87
	16	35.01	33.38	1.63	1.56
	32	60.35	58.05	2.81	2.71
	48	68.00	63.83	3.17	2.98
	59	69.54	65.81	3.24	3.07

5.3.2 Power Consumption

Table 5.6 shows the power consumption results for Sparse Vector-Matrix multiplication benchmark. The Tilera Pro64 uses up to 21 times less power than a single core of AMD quad-core Phenom II X4. However, it is perhaps better to look at the energy efficiency of the Tilera, which is the total energy used vs. the energy used by the CPU or the GPU.

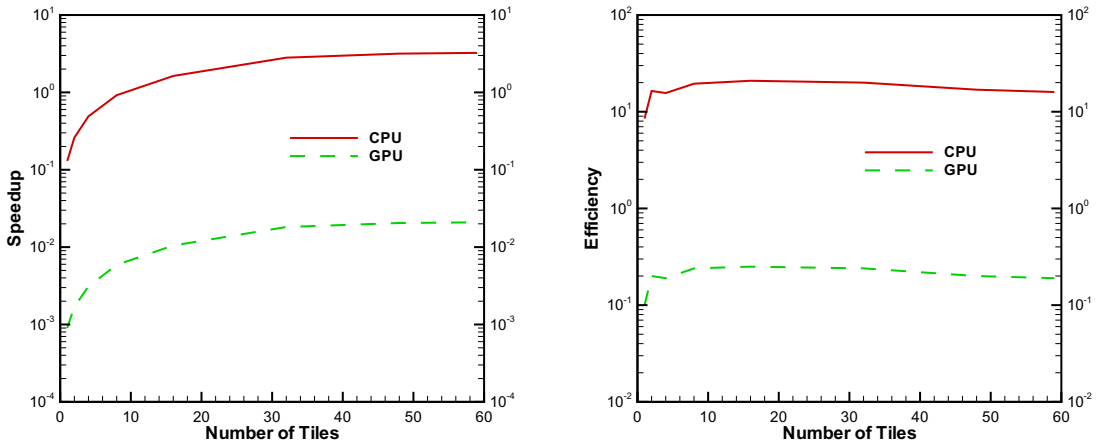
$$EnergyEfficiency = \frac{Power_{(CPUorGPU)} \times Time_{(CPUorGPU)}}{Power_{(Tilera)} \times Time_{(Tilera)}} \tag{5.3}$$

The table 5.6 shows that the Tilera is 16× more energy efficient than the CPU and 5× less efficient than the GPU. However, in practice, the Tilera requires a host (which draws 200-300 W of base power). If we were to account for the base power and use 4 (rather than 1 core) of the CPU, the Tilera would have the same efficiency as the CPU.

Table 5.6. Power consumption for sparse vector-matrix multiplication ($256 \times 256 \times 256$) for Tilera Pro64 comparing to the single core of AMD quad-core Phenom II X4 and Tesla C2070

CPU	GPU	Tilera Pro64			
Power (W)	Power (W)	# of Tiles	Power (W)	Energy Efficiency (CPU)	Energy Efficiency (GPU)
64	120	1	1	8.5	0.10
		2	1	16.4	0.20
		4	2	15.6	0.19
		8	3	19.5	0.24
		16	5	20.9	0.25
		32	9	20.0	0.24
		48	12	16.9	0.20
		59	13	16.0	0.19

Figure 5.7 shows the speedup and energy efficiency of the Tilera Pro64 compared to a CPU, GPU. The results show that up to 8 tiles the energy efficiency increases, and after that with increasing the number of tiles the energy efficiency is the same or decreasing. This shows that the power consumption on the Tilera is directly proportional to the number of memory accesses being performed.



(a) (b)

Figure 5.7. Speedup and energy efficiency for the 256^3 problem size for Tilera Pro64 compared to the single core of an AMD quad-core Phenom II X4 and compared to a Tesla C2070 GPU.

5.4 Smith-Waterman Algorithm

5.4.1 Anti-Diagonal Algorithm

Two different approaches to solving the Smith-Waterman algorithm were developed for the Tilera Pro64. The first one is the anti-diagonal approach. Because every number in the Smith-Waterman table anti-diagonal is independent, it is possible to do the calculation in parallel. Table 5.7 shows the results for this approach. The same approach was used on the CPU to get a fair comparison.

Table 5.7. Results for anti-diagonal Smith-Waterman algorithm with 59 tiles compared to a single core of an AMD quad-core Phenom II X4

Size	CPU		Tilera Pro64 (59 Tiles)		
	Time (s)	MCUPS	Time (s)	MCUPS	Speedup
1024 x 1024	0.261	4.025	2.236	0.469	0.117
2048 x 2048	1.250	3.356	4.319	0.971	0.289
4096 x 4096	5.155	3.255	9.454	1.775	0.545
8192 x 8192	23.478	2.858	19.518	3.438	1.203

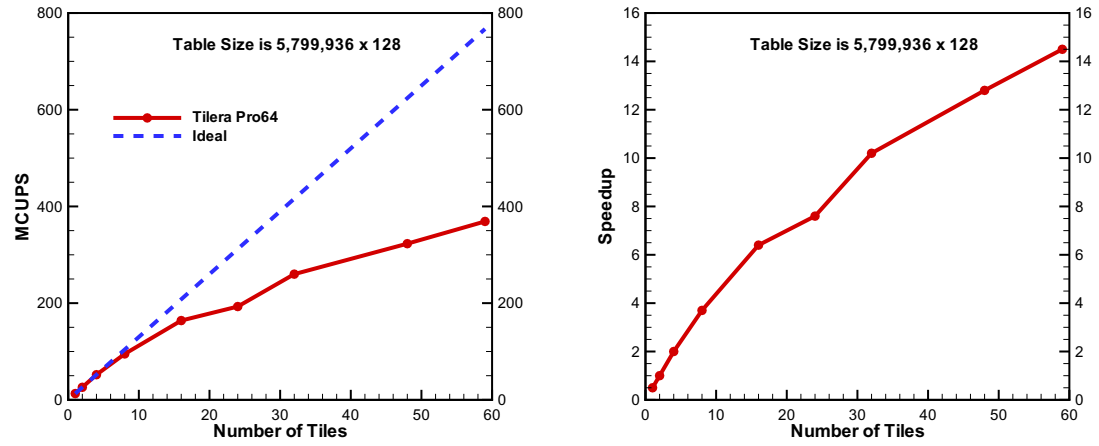
With this approach the MCUPS (millions of cell updates per second) is less than or equal to a single core of the AMD quad-core Phenom II X4. The problem with this approach is, when marching along the diagonal, the memory accesses miss the cache and the Smith-Waterman algorithm is then limited by the random memory access speed.

5.4.2 Row Approach

In the second approach (row approach) the database is divided between the tiles and each subsection of the Smith-Waterman table is calculated in parallel [49, 79]. In this approach some table overlap is necessary to overcome any dependencies between the subsections of the table. For large Smith-Waterman problems the overlap length is small compared to the length of each subsection and is negligible.

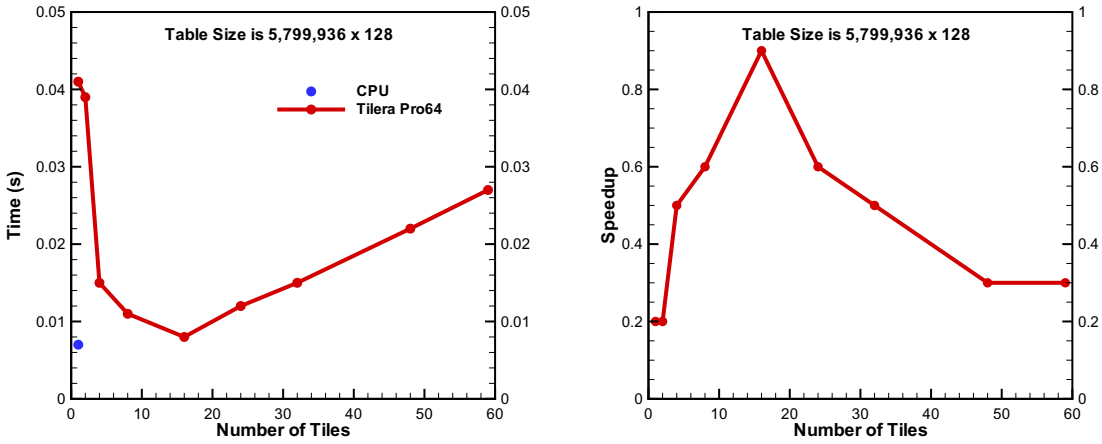
5.4.2.1 Strong Scaling

Figures 5.8 and 5.9 show the result for strong scaling (problem size is constant) for kernel 1 and kernel 2 of the SSCA#1 benchmark respectively. Kernel 2 is very fast, so its performance is not as important.



(a) (b)

Figure 5.8. Strong scaling for kernel 1 (a) MCUPS and (b) speedup for row-access Smith-Waterman algorithm with Tilera Pro64 compared with a single core of an AMD quad-core Phenom II X4.



(a) (b)

Figure 5.9. Strong scaling for kernel 2 (a) time (seconds) and (b) speedup for Smith-Waterman algorithm with Tilera Pro64 compared with a single core of an AMD quad-core Phenom II X4.

Almost 15× speedup is obtained for the table evaluation (kernel 1) compared to a single core of an AMD quad-core Phenom II X4. Kernel 2 is very fast (ms second) and independent of problem size. Table 5.8 (kernel 1) and Table 5.9 (kernel 2) show the MCUPS and speedup for strong the scaling case.

Table 5.8. Strong scaling results for row-access Smith-Waterman algorithm for kernel 1 with single core of AMD quad-core Phenom II X4 and Tilera

Kernel 1 (Problem Size = 5799936 x 128)						
CPU		Tilera Pro64				
Time (s)	MCUPS	# of Tiles	Time (s)	MCUPS	MCUPS/Tile	Speedup
29.21	25	1	58.68	13	13	0.5
		2	28.65	26	13	1.0
		4	14.34	52	13	2.0
		8	7.81	95	12	3.7
		16	4.53	164	10	6.4
		24	3.83	193	8	7.6
		32	2.86	260	8	10.2
		48	2.29	323	7	12.8
		59	2.01	369	6	14.5

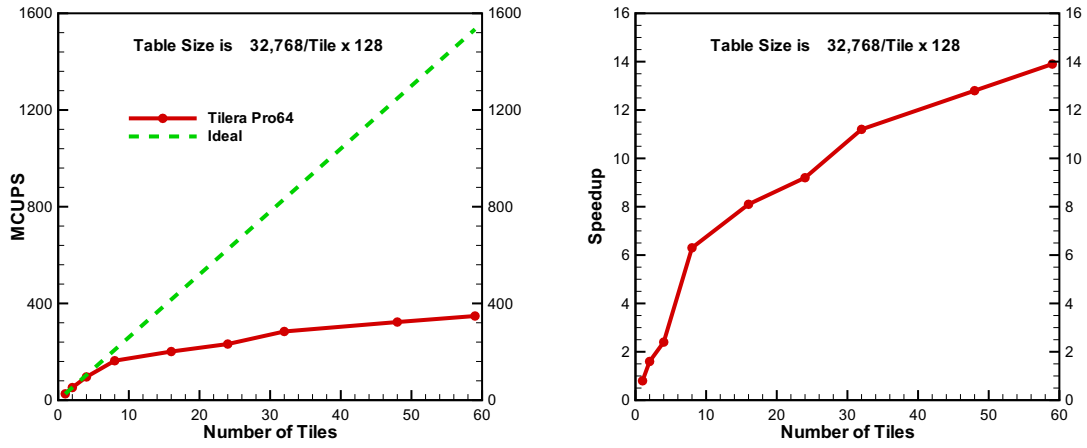
Table 5.9. Strong scaling results for row-access Smith-Waterman algorithm for kernel 2 with single core of AMD quad-core Phenom II X4 and Tilera

Kernel 2 (Problem Size = 5799936 x 128)			
CPU	Tilera Pro64		
Time (s)	# of Tiles	Time (s)	Speedup
0.007	1	0.041	0.2
	2	0.039	0.2
	4	0.015	0.5
	8	0.011	0.6
	16	0.008	0.9
	24	0.012	0.6
	32	0.015	0.5
	48	0.022	0.3
	59	0.027	0.3

Note that the MCUPS for the row-access method on the Tilera is almost 100× faster than anti-diagonal approach.

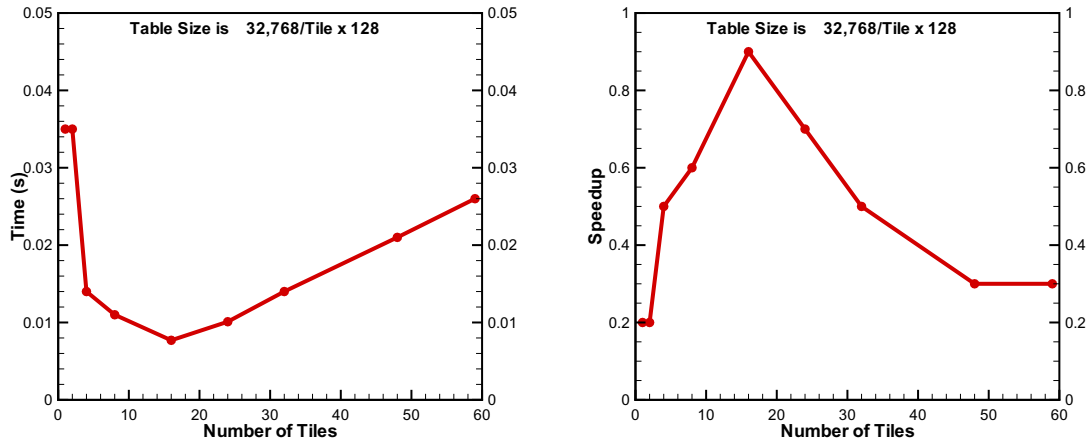
5.4.2.2 Weak Scaling

Figures 5.10 and 5.11 show the results for weak scaling (problem size per tile is constant) for kernel 1 and kernel 2 respectively. The Tilera gets almost 14× speedup, compared to a single core of an AMD quad-core Phenom II X4. Again, because kernel 2 is very fast it is not worth optimizing this kernel.



(a) (b)

Figure 5.10. Weak scaling for kernel 1 (a) MCUPS and (b) speedup for row-access Smith-Waterman algorithm with Tilera Pro64 compared with a single core of an AMD quad-core Phenom II X4.



(a) (b)

Figure 5.11. Weak scaling for kernel 2 (a) time (s) and (b) speedup for row-access Smith-Waterman algorithm with Tilera Pro64 compared with a single core of an AMD quad-core Phenom II X4.

Table 5.10 and 5.11 show the MCUPS and speedup for weak scaling with row approach for Smith-Waterman algorithm. This uses 32,768 table entries per tile.

Table 5.10. Strong scaling results for row approach Smith-Waterman algorithm for kernel 1 with single core of AMD quad-core Phenom II X4 and Tilera

Size	CPU		Tilera Pro64		
	Time (s)	MCUPS	Time (s)	MCUPS	Speedup
32768	0.13	32	0.16	26	0.8
65536	0.26	32	0.16	52	1.6
131072	0.41	41	0.17	96	2.4
262144	1.33	25	0.21	163	6.3
524288	2.67	25	0.33	201	8.1
786432	3.97	25	0.43	232	9.2
1048576	5.28	25	0.47	284	11.2
1572864	7.92	25	0.62	323	12.8
1933312	9.86	25	0.71	348	13.9

Table 5.11. Weak scaling results for row approach Smith-Waterman algorithm for kernel 2 with single core of AMD quad-core Phenom II X4 and Tilera

Size	CPU	Tilera Pro64	
	Time (s)	Time (s)	Speedup
32768	0.0067	0.035	0.2
65536	0.0068	0.035	0.2
131072	0.0069	0.014	0.5
262144	0.0067	0.011	0.6
524288	0.0067	0.0077	0.9
786432	0.0069	0.0101	0.7
1048576	0.0068	0.014	0.5
1572864	0.0068	0.021	0.3
1933312	0.0068	0.026	0.3

5.4.3 Power Consumption

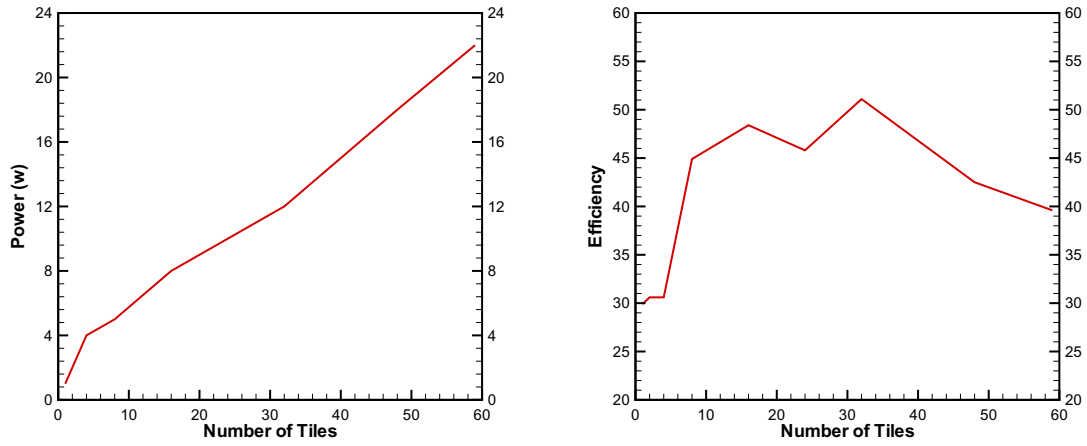
Table 5.12 shows the power and energy consumption results for the first kernel of the sequence matching benchmark (SSCA#1). The Tilera Pro64 uses from 30-50 times less energy compared to the single core of an AMD quad-core Phenom II X4. It was determined that the problem size doesn't have any effect on power consumption.

Table 5.12. Results for first kernel of SSCA#1 for Tilera Pro64 comparing to the single core of AMD quad-core Phenom II X4

5799936 x 128	CPU		Tilera Pro64			
# of Tiles	Time (s)	Power (W)	Time (s)	Speedup	Power (W)	Energy Efficiency
1	29.21	60	58.68	0.5	1	29.9
2			28.65	1.0	2	30.6
4			14.34	2.0	4	30.6
8			7.81	3.7	5	44.9
16			4.53	6.4	8	48.4
24			3.83	7.6	10	45.8
32			2.86	10.2	12	51.1
48			2.29	12.8	18	42.5
59			2.01	14.5	22	39.6

Figure 5.12 shows the power and energy efficiency of the Tilera Pro64 compared to a single core of an AMD quad-core Phenom II X4. The energy efficiency is given by,

$$EnergyEfficiency = \frac{Power_{(CPUorGPU)} \times Time_{(CPUorGPU)}}{Power_{(Tilera)} \times Time_{(Tilera)}} \tag{5.4}$$



(a) (b)

Figure 5.12. Smith-Waterman benchmark (a) power (W) and (b) energy efficiency compared to a single core of an AMD quad-core Phenom II X4

Note that after 8 tiles the energy consumption is constant, so the power draw is directly proportional to the work being performed.

It should also be noted that these energy efficiency calculations do not include the base power. In practice the Tilera and the AMD quad core require a host that draws 200-300 W when idle. This base power (of the host) will dominate the energy consumption. For this reason, the additional power the Tilera or the CPU draws when operating is essentially negligible in practice.

5.5 Fast Fourier Transform (FFT)

5.5.1 1D FFT

Because the CPU and TILERA architecture are the same in many aspects, the first step was to implementing different 1-D FFT algorithms on the CPU in order to find optimal algorithms for the architecture. In general, all FFT algorithms need bit reversal somewhere in the algorithm. It is possible to do bit reversal first or at the end of the FFT calculation.

Figure 5.13 shows the timing in milliseconds for different methods that were implemented in C++ and run on a single core of an AMD quad-core Phenom II X4. The main differences between bit reversed 1 and 2 is how to calculate the right indexes and twiddle factors. As expected, Radix-4 is always faster than Radix-2. But in order to run radix-4, the length of FFT should be multiple of 4.

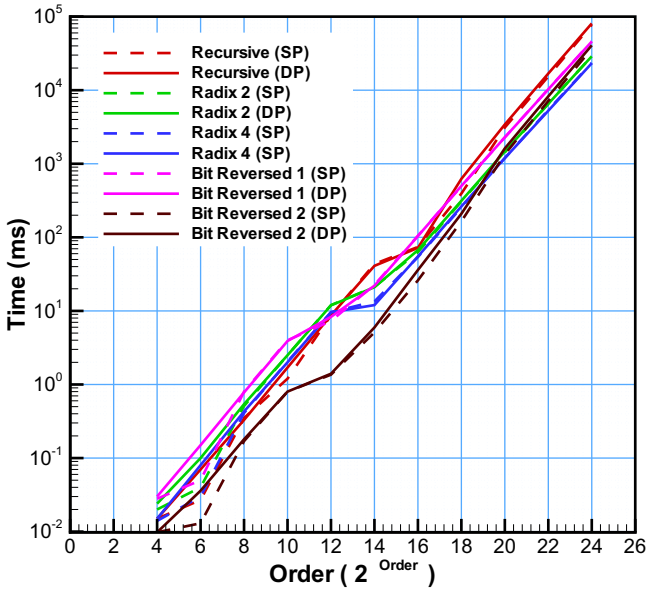


Figure 5.13. Timing for single and double precision for 1-D FFT running on a single core of the AMD quad-core Phenom II X4.

Table 5.13 highlights (in blue) the fastest algorithms. It was found that up to size of 262144 (218), bit reversed 2 has the best performance. 3D FFTs require many small length FFTs (of roughly size 256) so performance at this size is of interest. Radix-4 has the best performance for very large sizes. If one is interested running large FFTs on a single core, Radix-4 is suitable. But our goal was to run large FFT on many cores (TILERA). So bit reversed 2 is selected to be applied on the Tilera. It was determined that for parallel approaches, it is better to do bit reversal first and

then calculate the FFT. Because in this case, each tile or core can put data in its own cache and then do the rest of the calculation in the fast cache memory.

Table 5.13. Timing for single and double precision for 1-D FFT running on a single core of the AMD quad-core Phenom II X4.

Length	Recursive		Radix-2		Radix-4		Bit Reversed 1		Bit Reversed 2	
-----	Single	Double	Single	Double	Single	Double	Single	Double	Single	Double
16	0.015	0.015	0.02	0.024	0.014	0.015	0.028	0.03	0.0098	0.0098
64	0.027	0.07	0.04	0.1	0.03	0.08	0.05	0.15	0.013	0.036
256	0.35	0.33	0.51	0.54	0.41	0.43	0.8	0.78	0.17	0.18
1024	1.2	1.7	2.5	2.5	2.05	2	3.99	3.9	0.8	0.8
4096	8.7	8.7	11.98	11.9	9.6	9.6	7.3	8.3	1.36	1.4
16384	43.46	41	21.28	21	13.44	12	22.61	22	5.1	5.95
65536	74	72	63	67	54	55	105	104	26	36
262144	393	624	308	314	255	263	497	499	165	209
1048576	3112	3423	1428	1471	1196	1192	2290	2284	1345	1619
4194304	15389	16861	6544	6547	5398	5263	10343	10294	7153	8245
16777216	77633	80842	28978	28888	23476	23253	46145	45831	36759	40744

5.5.2 3D FFT

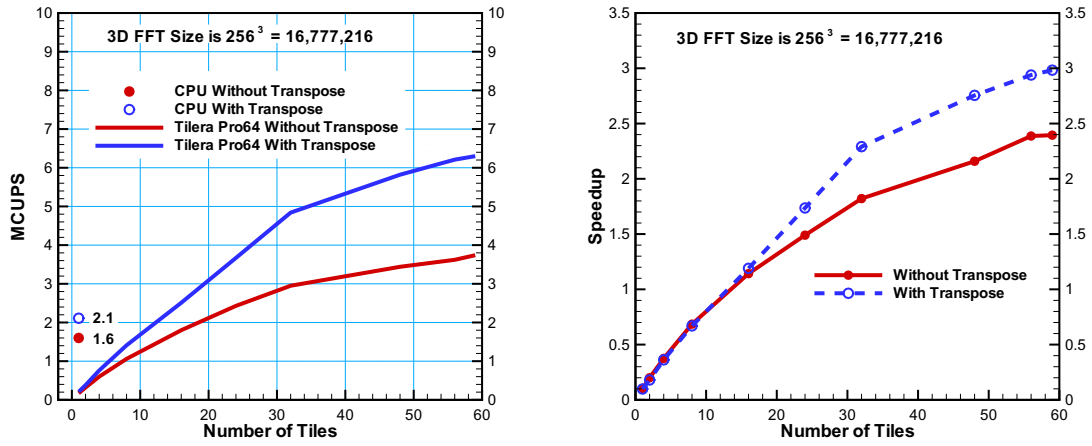
The 1D FFT algorithm was extended to the 3D FFT. The data layout in memory is important to 3D FFTs. Usually the layout means that one direction of FFT is efficient and others are not (because those directions have a long stride through memory). In our layout, the FFT in the X-direction is more efficient than those in the Y- and Z- directions. The common solution for this problem is changing the data layout in memory before the FFT in the next direction. This means that the data is transposed in such way that memory layout is linear (and stride 1) for the new FFT direction. In order to obtain properly ordered results, two matrix transpose are needed (one forward, and one in reverse). But there is a smart way to reduce or even hide the first transpose by doing the transpose and bit reversal at the same time. Table 5.14 shows the time and percentage of each FFT direction for 256^3 with 59 tiles.

Table 5.14. Results for 3D FFT with a single core of AMD quad-core Phenom II X4 and Tilera Pro64

256^3	CPU (s)				TILERA (s)					Speedup
59 Tiles	Time	X FFT	Y FFT	Z FFT	Time	X FFT	Y FFT	Z FFT	MCUPS	
Without Transpose	20.891	3.362	5.574	11.955	8.967	0.770	1.453	6.742	3.74	2.323
		16 %	27 %	57 %		9 %	16 %	75 %		
With Transpose	15.89	3.401	5.315	7.173	5.464	0.923	1.556	2.974	6.14	2.908
		21 %	33 %	45 %		17 %	28 %	54 %		

The matrix transpose was only applied in the Z-direction for both CPU and Tilera. It is also possible to do a matrix transpose before the Y direction FFT. But this saves only 5-10%. Based on this approach the Tilera is almost 3× faster than a single core of the AMD quad-core Phenom II X4.

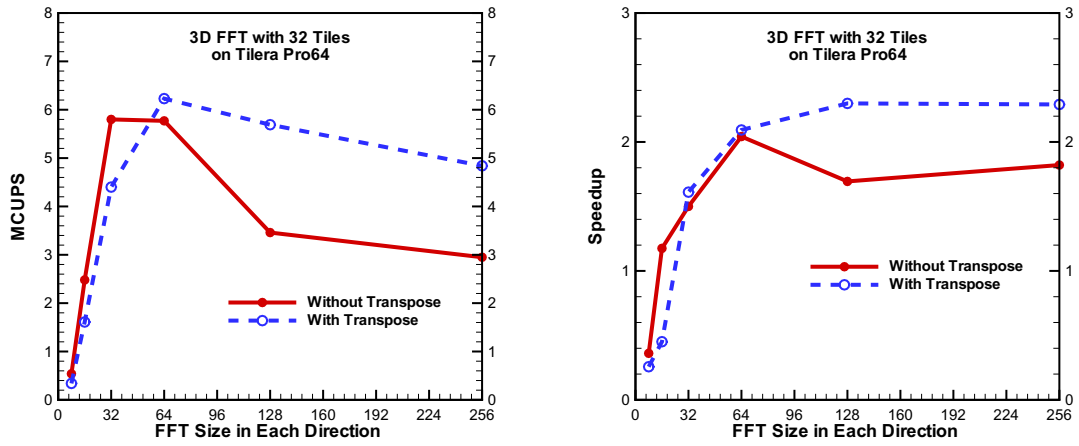
Figure 5.14 shows the MCUPS (millions of cell updates per second) and speedup for a 256^3 3D FFT with different numbers of tiles compared with one core of the AMD Phenom II for two different approaches. Results are presented based on the average of 10 iterations. For CPU timings, we used an in-house 3D FFT code, so that the algorithm is identical. The tiles speed up nearly linearly up to 4 tiles, after that performance decreases with additional tiles, and becomes very small with more than 32 tiles.



(a) (b)

Figure 5.14. (a) MCUPS and (b) speedup for 3D FFT with 2563 for different number of tiles

Figure 5.15 shows the 3D FFT with 32 tiles for different problem sizes. With a 64^3 FFT the performance of the Tilera is almost 2.5 times that of a single core of the CPU. Also results with matrix transpose show the effect of the cache on the Tilera performance. With the matrix transpose, after 64^3, the code shows constant speedup.



(a) (b)

Figure 5.15. (a) MCUPS and (b) speedup for 3D FFT with 32 tiles for different problem sizes with and without transpose

5.5.3 Power Consumption

Table 5.15 shows the power consumption and energy efficiency for the 3D FFT benchmark. The Tilera Pro64 uses up to 8 times less power compared to the single core of an AMD quad-core Phenom II X4. But we should mention here that, we just used a single core of the CPU. The cost of using additional cores is far less (12 W per additional core). In addition, the base power (200-300 W) to host the processors far exceeds the incremental power costs associated with actually running a computation on the processors.

Table 5.15. 3D FFT results for Tilera Pro64 for $256 \times 256 \times 256$ compared to the single core of an AMD quad-core Phenom II X4

CPU		Tilera Pro64				
Time (s)	Power (W)	# of Tiles	Time (s)	Speedup	Power (W)	Energy Efficiency
15.89	58	1	162.220	0.098	2	2.8
		2	88.258	0.180	2	5.2
		4	43.950	0.362	3	7.0
		8	23.736	0.669	5	7.8
		16	13.348	1.190	10	6.9
		32	6.935	2.291	16	8.3
		48	5.767	2.755	20	8.0
		56	5.406	2.939	22	7.7
		59	5.326	2.983	23	7.5

5.6 Conclusions

We have implemented four important high performance computing benchmarks on the Tilera Pro64. Like every multi and many core architectures, the Tilera has some advantages and disadvantages. We have divided these features into seven different design metrics; performance, bandwidth, price, software, hardware power and scalability.

Based purely on speed, the Tilera is roughly competitive with a CPU and significantly slower than a GPU. The reason for the relatively low speeds is primarily due to the design of the memory subsystem.

Memory bandwidth is the most critical hardware performance measure for all the benchmarks tested in this work. The Tilera Pro64 has less bandwidth to main memory than a typical CPU or GPU. For example the Tilera Pro64 has maximum of 14 GB/s and modern CPUs are around 40 GB/s and GPUs are roughly 180 GB/s. Newer GPUs have an L1 and an L2 cache (like the CPU and Tilera). The newest Tilera (100-core, not yet released) uses DDR3 instead of DDR2 so its maximum theoretical

bandwidth is close to a modern CPU (around 65 GB/s). GPUs currently use DDR5 memory. The Tilera has essentially 4 memory channels to main memory. After 8 tiles start accessing memory, the performance with more tiles is relatively small. It has a fast inter-tile communication network, but we found it very difficult to use this hardware characteristic to any substantial advantage for the 4 benchmarks presented here.

One can buy the latest GPU for around $500-$3000, an Intel CPU (10/20 core/thread E7 Xenon series) for around $4000, and an AMD 12-core for around $1600. The new Tilera with 100 tiles (cores) costs around $11000. It seems that Tilera hardware is expensive compared to CPUs and GPUs. This is probably due to the economies of scale and the high demand for CPU and GPU products outside the area of high performance computing.

The CPU and the GPU (CUDA and OpenCL) programming languages are well known and more up to date than the Tilera's Multicore Development Environment (MDE). However, we should mention that changing codes that are already written in C (not C++) to the Tilera language is very straightforward. One just needs to modify the algorithm in order to take advantage of the Tilera architecture.

The Tilera appears to be compatible with only a few system configurations. For example, the available MDE compilers are compatible with only three specific versions of Linux and three hardware systems (CPU and motherboards).

The Tilera uses 75% less power (above the base/host power) Roughly 24 W extra for the whole TilePro64. Whereas a GPU takes roughly 115 W per GPU and a 4-core CPU takes about 110 W for all four cores. More power means more heat and more cooling. So saving 1 W in power also means saving another watt in cooling costs. Note however, that the Tilera requires a host computer a typical host takes 200-300 W to run so that the power savings of the Tilera are actually highly marginalized by the power costs of maintaining the Tilera's host. In addition, the fast speed of the

GPU means that it typically consumes less total energy for a task than the Tilera even though its power consumption is much higher.

Only one Tilera card was tested in this work. However, the Tilera has a number of interconnect options on the card itself. So unlike a GPU, it may be possible to bypass the CPU host and directly connect the Tilera cards together. This might alleviate the MPI bottleneck currently experienced when using many GPUs in a cluster environment.

CHAPTER 6

CFD AND GPUS

6.1 Introduction

The in-house CFD code (Stag3D) is used to simulate the incompressible Navier-Stokes equations. This code was originally written in FORTRAN 95. A second-order accurate staggered mesh scheme for space descretization along with a three-step second-order convectively stable Runge-Kutta time marching scheme are employed. A classic projection method is used for the pressure solver. A non-uniform spacing in vertical direction, ability to set a constant pressure drop across the computational domain in both the streamwise and spanwise direction and also parallelization for use on supercomputer clusters were added by Michael Martell [80]. Stag3D was used for hybrid RANS/LES model development [81, 82] and for the direct numerical simulation on turbulent 3D channel flows with superhydrophobic walls [83, 84].

Stag3D is validated for laminar and turbulent flow cases. For the laminar regime, Poiseuille flow and Couette flow were compared with analytical solutions [80]. The 2D Taylor vortex decay was performed to validate the code's transient solution capabilities. The code was validated against previous widely known numerical results for turbulent channel flow [85, 86].

In order to apply Stag3D to the GPU, Stag3D was converted to C++ and the CUDA environments by Ali Khajeh-Saeed and Timothy McGuiness. The C++ version of Stag3D is named Stag++. The code is also modified and became object-oriented and optimized for the CPU by Prof. Blair Perot. Also, non-uniform meshes in all direction are implemented in Stag++. The code is now fully parallel, using

MPI libraries and optimized for execution on CPU and GPU based supercomputers clusters. The important optimization techniques applied in Stag++ will be discussed in detail in the next sections. The code is capable of domain decomposition in every direction, and each partition is then handled on individual CPU cores or GPU units. Stag++ now supports only a Cartesian mesh, but it is designed in such way that applying an unstructured mesh in the future is straightforward.

6.2 Literature Review

Graphics processing Units (GPUs) present an energy and time efficient architecture for High Performance Computing (HPC) in flow simulations [87, 88, 89, 90, 11, 91, 92, 93, 94]. Several researchers have shown that GPU simulations obtain one or two orders of magnitude speedup over optimized CPU implementations [95, 96, 97, 98]. Some cases that have been implemented on the GPU are Lattice Boltzmann [99, 100, 101], 2D [102] and 3D [103] Euler solvers, atmospheric dispersion simulations [66], an incompressible Navier-Stokes solver with Boussinesq approximation [67], multi-GPU 3D incompressible Navier-Stokes solver with a Pthreads-CUDA [104], and finite elements [105]. Some of the important and useful papers for CFD computations on GPUs are reviewed in detail below.

Three dimensional Euler equations have been solved on the GPU with OpenGL by Hagen et al. [87]. On their implementation, each pixel (fragment) in an off-screen frame buffer is assigned to a grid cell. The data stream (cell-averages, fluxes, etc.) is saved in the textures memory and is invoked by rendering the geometry to a frame buffer. They run two test cases; 2D shock-bubble interaction and 3D Rayleigh-Taylor instability and achieved a maximum speedup 25 and 14 respectively, with GTX 7800 comparing with AMD Athlon X2 4400+.

Harris [88] proposed a flow solver based on Stam's "stable fluids algorithm". He simulated a periodic volume of fluid on a two-dimensional rectangular domain using a Cartesian power-of-two mesh.

Elsen et al. [89] presented a simulation of a hypersonic vehicle configuration on the GPU using the compressible Euler equations. They used the Navier-Stokes Stanford University Solver (NSSUS). The NSSUS program solves the three-dimensional Unsteady Reynolds-Averaged Navier-Stokes (URANS) equations on multi-block meshes using a vertex-centered solution with first to sixth order finite difference and artificial dissipation operators. A geometric multi-grid scheme with support for irregular coarsening was also used to accelerate the solution of the system. They ported their code to the GPU using Brook. Brook is a source to source translator which converts Brook code into C++ code and a high-level shader language like Cg or HLSL. They achieved speedups of over 20 based on a comparison of the Intel Core 2 Duo and NVIDIA GTX 8800.

Corrigan et al. [90] solved the three-dimensional Euler equations for inviscid, compressible flow on an unstructured grid. The computationally intensive part of the solver consists of a loop in the host (CPU) which repeatedly computes the time derivatives of the conserved variables. For the time-stepping scheme they implemented an explicit Runge-Kutta algorithm. The most expensive computation in their application consists of collecting flux contributions and artificial viscosity across each face when calculating the time derivatives. One thread per element is assigned for the time derivative computation. First, each thread loads the element's volume and conserved variables (from global memory), from which derived quantities such as the velocity, pressure, the speed of sound, and the flux contribution are calculated. After that the GPU kernel loops over each of the four faces of the tetrahedral element, in order to compute fluxes and artificial viscosity. All required derived quantities are calculated in the GPU and then the artificial viscosity and flux are added to the

element's residual. But the main issue with this approach is redundant computation of the flux contributions and other quantities derived from the conserved variables. They also tried another possible approach in which they precomputed each element's flux contribution, in order to avoid redundant computation. However, they figured out this approach is slower even though they avoided redundant computation. This is because the computational grid domain is unstructured, and the global memory access required for loading the conserved variables of neighboring elements is highly non-coalesced, which results in lower effective memory bandwidth. They avoided some non-coalesced accesses by renumbering the grid points. If neighboring elements are nearby in memory to each other then the possibility of coalesced memory access will be increased.

Micikevicius [11] applied 3D finite difference computation using CUDA on the GPU. He introduced a method that attempts to hide GPU communication time with computation time. In this work, data is partitioned by assigning each GPU half the data set plus (k/2) slices of ghost nodes (see figure 6.1). Each GPU computes its half of the output, receiving the updated ghost nodes from the neighboring GPU.

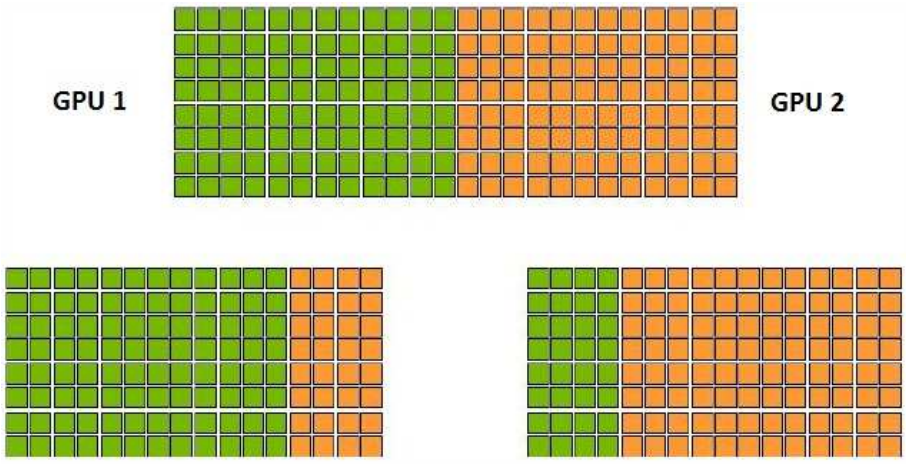


Figure 6.1. Data distribution between two GPUs [from [11]]

He divided the data along the slowest varying dimension so he guaranties that contiguous memory regions are copied during ghost node exchanges. In order to maximize scaling, this approach overlaps the exchange of ghost nodes with kernel execution by defining different CUDA streams. Based on the GPU architecture, different streams can be executed in the same time. In most GPU architectures it is possible to copy data from the CPU to the GPU or vice versa and execute a program at the same time.

Each time step is executed in two phases. In the first phase, a GPU computes the slices of ghost nodes in the neighboring GPU. In the second phase, a GPU executes the compute kernel on its remaining data (large data set), at the mean time exchanging the ghost nodes with its neighbor with *cudamemcopy* and MPI (see figure 6.2). For each CPU process controlling a GPU, the exchange involves three memory copies: GPU to CPU (*cudamemcopy DevicetoHost*), CPU to CPU (*MPI_Isend* and *MPI_Irecv*), and CPU to GPU (*cudamemcopy HosttoDevice*). After copying the data and executing the kernel, it is necessary to synchronize the copy procedure before sending or receiving data with MPI. Based on this approach he achieved linear speedup for running up to four GPUs.

Rossinelli et al. [92] described a GPU solver for simulations of bluff body flows in 2D using a remeshed vortex particle method and the vorticity formulation of the Brinkman penalization technique to enforce boundary conditions. They used OpenGL to perform efficient particle-grid operations. For solving the Poisson equation they applied the CUFFT-based (CUDA FFT) solver that is in the CUDA BLAST library. They also used regularly spaced grid nodes for the computational domain. They employed an RGB texture to represent a set of particles where each texture element (texel) contains the state of one particle. The red and green channel of a given texel represent the particle position in two dimensional flows. The B channel saves the

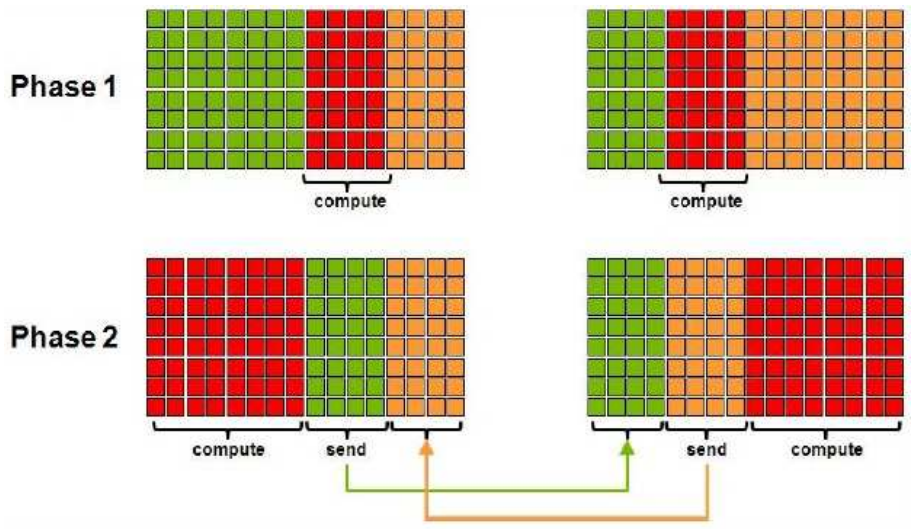


Figure 6.2. Two phases of a time step for a 2-GPU [from [11]]

transported vorticity [91]. For simplicity they assigned single texel to a grid node in computational domain. They achieved a speedup of 25 for their implementation.

Antoniou et al. [93] implemented the finite-difference weighted essentially non-oscillatory (WENO) scheme on the programmable GPU with CUDA. On the GPU they assigned a single thread per element in 2D plan (x and y axis) and marching along the z axis. They implemented this strategy of 2D domain decomposition on the single GPU and multi-GPU mode and figured out that this approach adds additional capabilities for large scale DNS or LES. Furthermore, they used multiple of 32 points in each block to maximize the throughput with appropriate expression of parallelism. They anticipated that better handling of data transfer among GPUs can further increase processing speed. In their results they didn't get linear speedup for multiple GPUs. One possible reason for this less than optimal performance is that data communication between the CPU and the GPU was not overlapped. Due to low bandwidth of PCI-e bus, it is necessary to implement a method that decreases the number of communications or hides data transfers.

Jacobsen et al. [94] discretized the Navier-Stokes equations on a uniform Cartesian staggered grid with second-order central difference scheme for spatial derivatives and a second order accurate Adams-Bashforth scheme for time derivatives. The projection method is applied to solve the Navier-Stokes equation for incompressible fluid flows. For pressure, the Poisson equation is solved using a Jacobi iterative solver. They applied three different data communication approaches; non-blocking MPI with no overlapping of computation, overlapping computation with MPI and finally overlapping computation with MPI communications and GPU transfers. Their results show that overlapping computation with MPI and overlapping computation with MPI communications and GPU transfers have same effects. Even in some cases overlapping computation with MPI is faster than other approaches for a number of GPUs more than 16. They achieved 11× speedup compared with an 8-core CPU (using OpenMP) for a single GPU and 130× speedup for 128 GPUs for the strong scaling case.

6.3 Optimization Techniques

6.3.1 Removing Ghost Cells and Maximizing Coalesced Access

As mentioned in chapter 2, an important optimization for the GPU is coalesced access to the global memory. Stag++ is designed in such way that there are no ghost cells around each sub-domain. The solver loads one XY plane at a time and solves that plane before marching along to the next plane in the z direction. For obtaining maximum bandwidth of the GPU, it is necessary that the number of cells in every plane should be multiple of 16. For example, if there are 16 threads executing with the same kernel, 16 sequential positions in global memory (1 position per thread) can be accessed in the same time that it would take 1 thread to read 1 position in memory (coalesced access). If all memory accesses are performed this way, performance can speed up by a factor of 16 (in the memory access code). Also the address of the start

point for reading the data from global memory should be a multiple of 64 bytes $\big($16 (data)$\times$4 (bytes for single precision)$\big)$ to get maximum coalesced access.

If the code was to use ghost cells at every sub-domain, coalesced access is impossible. If one assumes that sub-domain size is a multiple of 16 (including the ghost cells), there is still a problem. The problem is thread divergence. So instead of using 16 threads (half-warp) 14 threads are used ($16 - 2$ (ghost cells at the beginning and end of the plane)). Because we are marching in Z direction we keep two extra planes (the just before and just after planes) in order to perform the z-direction fluxes. One major change from Stag3D to Stag++ was removing the ghost cells in the XY planes.

6.3.2 Conjugate Gradient and Matrix Multiplication

The code solves the pressure Poisson equation using a polynomial preconditioned Conjugate Gradient (CG) iterative method. The conjugate gradient method is an efficient iterative method and is guaranteed to converge for a symmetric, positive-definite matrix. The CG method is composed of one matrix multiply, one preconditioner matrix multiply, 2 scalar (dot) products, and 3 AXPY (Alpha X Plus Y) operations. The three AXPY parts are easily mapped to the GPU architecture. However, the most computationally intensive part of the solution procedure is the matrix multiply which computes the Cartesian-mesh discrete Laplacian. In the current implementation the preconditioner has the same sparsity pattern as the Laplacian matrix and is therefore implemented in exactly the same way as the Laplacian. To compute the Laplacian matrix for a particular cell, all neighboring cells and the central cell are needed (7 cells in 3D). This is more difficult than the simple AXPY to map to the GPU.

When performed naively, the 7 point matrix stencil reads each data item 7 times from the main GPU memory. Only 3 of the points are linear, stride one, and therefore fast, the others are large stride memory accesses and in terms of speed are essentially

random memory operations. The code is made more efficient by using a modified version of Micikevicius [11] implementation. This involves reading the data once into the shared memory on each GPU multiprocessor, and then accessing it from the fast memory location 7 times. To do this each multiprocessor keeps three XY planes of data (from the data chunk) in its memory. The middle XY plane contains 5 of the stencil points (in the X and Y directions) saved in shared memory, and the upper and lower XY planes contain the 6^{th} and 7^{th} stencil values (just above and below the middle XY plane) saved in a register. After the discrete Laplacian is computed for the middle plane, the middle (shared memory) and upper (register memory) planes are copied to the lower (register memory) and middle (shared memory) planes respectively. The top plane reads in the new data from the main (global) GPU memory to the register memory (See figure 6.3).

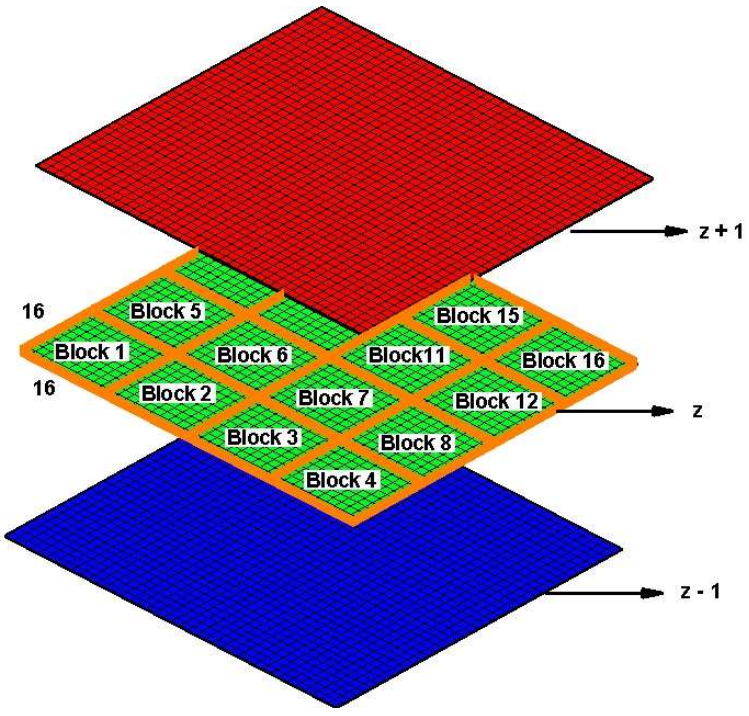


Figure 6.3. Thread and block distribution for a XY plane

Note, however, that in order to compute a 16×16 Laplacian stencil result, an 18×18 data input is actually required (minus the 4 corners). This is read in as a 16×18 block (with stride 1 fast access), and two 1×16 strips for the two sides. These last two strips have a stride equal to the subdomains size in the x-direction, and therefore are much slower to read. It therefore takes roughly the same amount of time to read the two 1×16 strips as it does the rest of the data (18×16). This is the first example of where the internal data is processed so efficiently that the unusual operations (2 boundary strips in this case) take just as much time as the far more numerous (but much more efficient) common operations.

The other option would be to read 16×16 blocks efficiently (no side strips) but only compute a 14×14 region of the stencil. Because the SIMD cores process 16 items at a time, this means that the code would have an instruction divergence. This is where some cores do an operation, and some others do something else. On these SIMD cores this results in slower execution (by about a factor of 2). In addition, this approach means the multiprocessors are reading blocks of 16 that overlap at the edges. This makes the 16×16 read slower. Therefore, the advantage of reading no boundary strips is actually lost. One way or the other, there is a price to be paid for the fact that the chunks of data being computed on each multiprocessor must use data from a different chunk. We have structured the algorithm so that the data transfer between chunks is an absolute minimum, (it is about 1/8th of the internal data on each chunk). Nevertheless, the slower times for boundary data between chunks means that this smaller amount of data still has a significant impact on the total solution time.

The second major optimization in the CG algorithm is to perform the two dot products at the same time as the matrix multiply and the preconditioner matrix multiply (one dot-product for each). Both arrays for the dot-product are already in fast shared memory when performing the matrix multiply, so this saves reading the two

arrays for each dot-product (4 array reads in total). The dot-products are therefore essentially free of any time impact on the code, except that their final result must be summed among all the GPUs. This requires an MPI all-to-all communication that cannot be hidden by any useful computations (but the amount of data communicated is very small, one word per GPU). We recognize that restructuring of the CG algorithm can be performed in order to overlap dot-product summations with computation, however this also leads to a CG algorithm with more storage and more memory read/writes. So the speed improvement of a modified CG algorithm is not expected to be significant.

With 512^3 meshes, naive summation (for turbulence averages) can lead to round-off errors that are on the order of 10^9 times the machine precision (for single precision this would mean order 1 error). Even though we use double precision in all the computations, summation is still performed in stages to reduce the round-off error. The 3D array is first collapsed into a 2D array using the GPU, by summing along the Z-direction. Further reduction is then performed by reducing in the Y-direction, and then the X direction on the CPU, and then by summing the results from all the GPUs using MPI all-to-all communication (4 stages in total). This procedure only looses roughly two decimal places of accuracy during the summation, and allows the expensive portion of the computation (the first reduction to XY planes) to be performed on the fast GPUs.

6.3.3 Reduction and Maximum

For reductions (dot products) and finding the maximum value in a vector the same algorithm is used. Each block is responsible for a single XY subset of the plane. 256 Threads in each block start to find the summation or maximum values in specified XY plane using shared memory. After this step every block reduces its results to single number. Based on a block index, every block writes the result to the

global memory. The output of these kernels is nz numbers (nz is number of cells in z direction).

Page-locked memory with the mapped memory flag is used here for the output of these kernels. So there is no need to explicitly copy data between the device and host. After executing the kernel, the final sum or maximum (of the nz values) is calculated by the CPU. It is possible to find the single summation or maximum value in a single GPU kernel. But there is a very small amount of work to do for a GPU at the final summation or maximum value and this is not efficient ($nz < 256$).

A second optimization is important when many GPUs (or CPU cores) are operating. Because every summation or maximum value in single node (CPU or GPU) should be sent to all nodes, it is necessary to copy a single data from the GPU to the CPU and send it with MPI. When the copy amount is small, the initial cost of copying is dominant. Based on these two reasons, page-locked memory is used in these two kernels with finalizing the results on the host side.

6.3.4 Multi-GPU Optimization Algorithm

The main goal is here to hide sending and receiving boundaries time. The basic structure of a subroutine is therefore:

(A) On the GPU, load the 6 boundary planes of the subdomain data (which resides in the GPU main memory) into 6 smaller (and stride 1) arrays

(B) Copy the small boundary arrays from step (A) to the CPU

(C) Send the data planes using MPI.

(D) On the GPU, start the internal calculation. This step is the primary action of the subroutine.

(E) Receive the data planes using MPI.

(F) Copy the received data from the CPU back to the GPU.

(G) Apply the boundary data to the calculation.

The flow chart for the Laplace, Gradient, Divergent, Convection and Laplace Inverse is shown in figure 6.4.

Figure 6.4. General flow chart for Laplace, Gradient, Divergent, Convection and Laplace Inverse operators

Because all data is on the GPU, it is necessary to copy data to the CPU in order to send it with MPI. When the mesh is partitioned, partion boundary data needs to be sent via MPI because there are no ghost cells. When sending the boundary data planes, the boundary values must first be copied from the GPU to the CPU. Copying data between the GPU and CPU (and vice versa) is explained here.

In order to hide the copying time, it is necessary to use different streams for extracting kernels with copying command and interior execution. In Stag++ we used stream number 2 for extracting and copying code and stream number 1 for executing the interior kernel. Because these are different streams, the GPU can execute these kernels at the same time. The modified structure of a subroutine is therefore:

(A) On the GPU, load the 6 boundary planes of the subdomain data (which resides in the GPU main memory) into 6 smaller (and stride 1) arrays. This requires the GPU and usually cannot be well overlapped with GPU computations.

(B) On the GPU, start the internal calculation. This step is the primary action of the subroutine.

(C) Copy the small boundary arrays from step (A) to the CPU. This can overlap with part (B).

(D) When part (C) is finished Send/Receive the data planes using MPI. The CPU handles all MPI operations and is otherwise idle, so this can also overlap with part (B).

(E) Copy the received data from the CPU back to the GPU. Again, this still can overlap with part (B).

(F) When both part (B) and part (E) are finished, apply the boundary data to the calculation.

Alternatively, if we use *zerocopy* memory types there is no need to explicitly copy data from the GPU to the CPU or vice versa. We just need to synchronize the kernel to make sure the data is on the CPU or the GPU. In order to get high performance, we chose regular pinned memory for the MPI sending buffer to load data from the GPU to the CPU. To receive data we also used regular memory. There is also a alternative way to hide the copying and MPI by using mapped and write-combined memories. In this approach, mapped is used for send buffer and write-combined memory is used to receive buffers. As mentioned before, it is more efficient to use write-combined memory for receive buffers. Because this kind of memory is fast and efficient when CPU just writes to memory and GPU reads from that memory. With these approaches, the algorithms shown in figure 6.5 are :

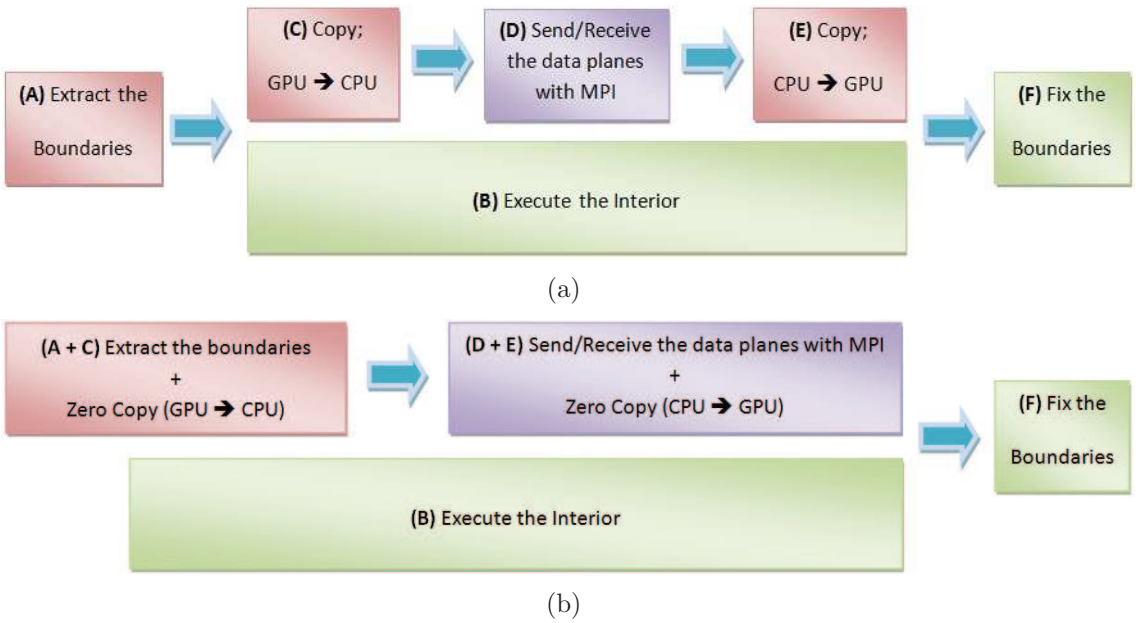


Figure 6.5. Efficient flow chart for Laplace, Gradient, Divergent, Convection and Laplace Inverse operators for (a) regular pinned memory (b) mapped memory; red, green and purple boxes are using stream 2, stream 1 and CPU respectively to execute the box

In the latest GPU architecture (Fermi), it is possible to run up to 16 kernels at the same time. So in theory part A and B can be executed at the same time if there are available resources in the GPU. In practice, the code rarely goes faster when doing this.

Since pinned memory is used in this algorithm, there is no need to explicitly copy data from the GPU to the CPU, but synchronization is necessary to make sure that data is available for the CPU to send with MPI (third box in figure 6.5). When the boundaries are available to send, the CPU sends the data (via MPI) and in the mean time the GPU is executing the interior kernel (second box in figure 6.5). Also there is no need to copy data from the CPU to the GPU, because the pinned memory with the write-combined flag is used for the MPI receive buffers. Finally, the kernel that fixes the result of the interior calculation to account for the boundary data is on the same stream as the interior kernel, so there is no need to synchronize the GPU before

launching the fix-boundaries kernel (because it must happen after the interior kernel is finished).

6.3.5 Avoid *cudaThreadSynchronize* and unnecessary *cudaStreamSynchronize*

cudaThreadSynchronize is very expensive and makes the CPU and GPU stall and wait for all the GPU kernels to complete before continuing the execution. *cudaThreadSynchronize* sometimes takes $200\mu s$ and also stops the CPU to load the code to the GPU. *cudaStreamSynchronize* like *cudaThreadSynchronize* makes the CPU and GPU stall if the specified stream is still running. But if the stream has already finished, *cudaStreamSynchronize* doesn't take too much time ($10\mu s$).

6.3.6 Maximize The GPU Occupancy

By passing the "–ptxas-options=-v" flag to the NVIDIA CUDA Compiler (NVCC) and specifying the CUDA Compute capability, the number of registers, the amount of static shared memory and local memory used by a specific kernel can be specified before executing the code. Based on these numbers, the number of threads per block and using the CUDA Occupancy Calculator Spreadsheet, the number of active threads per multiprocessor (occupancy) is identified. If occupancy is less than 25% and restricted by the amount of registers per thread, it is possible to decrease the number of registers and increase the occupancy. If occupancy is restricted by the amount of shared memory, constant memory should be used whenever possible. Each kernel in Stag++ is optimized for double precision and for the GT200 and Fermi series. In CUDA version 3.0 and higher, it is possible to control the number of registers for individual code kernels by using __ *launch_bounds*__ command in the declaration of the kernel.

6.3.7 Minimize Shared Memory and Maximize Constant Memory Usage

Grid information such as $\Delta x, \Delta y$, Δz and grid points are copied to constant memory. Because Stag++ is capable of mesh motion in every time step, two copies of $\Delta x, \Delta y$ and Δz are in the code, one copy in global memory and one copy in constant memory. Every time mesh motion is needed, data in global memory is updated and then data are copied from global memory to the constant memory (device to device copy). Device to device copy is fast and close to the device bandwidth (177 GB/s for Fermi architecture GPUs) and these copies are running on different streams. So it is straightforward to hide the copying time with kernel execution and this approach is efficient enough.

6.3.8 Memory Allocation and Deallocation

Memory allocation and deallocation are expensive. Also when there is an allocation call in the code, the GPU is synchronized implicitly and the CPU is stopped in order to load the code to the GPU. For this reason, temporary space is allocated only once and used wherever necessary. Similarly, these temporary arrays are all deallocated only once at the end of the execution.

6.3.9 Minimize The Data Transfer Between Host and Device

Data transfers between host and device and vice versa are slow and expensive (5-10 GB/s). Data transfers are minimized in Stag++, and only when data is needed (to be sent with MPI) to other nodes, is data copied to the host. When data is needed to copy to the host, pinned memory is used in a kernel. Pinned memory has fast transfer rate comparing to other types of memories in CUDA.

CHAPTER 7

STAG++ PERFORMANCE RESULTS

7.1 Introduction

As mentioned earlier, pressure solvers are expensive in incompressible flow solvers. In Stag++ almost 90% of the time is spent in the pressure solver. The most time consuming part in the pressure solver in the Stag++ code is the Laplace kernel (Extract the boundaries (A) + Interior calculation (B) + Fix the boundaries (F)). Almost 50% of GPU time is spent in these kernels (for 128^3 mesh). Based on these timing results, these kernels have the most influence on the performance of the solver and should be optimized the most.

7.2 Optimization Techniques Using NVIDIA Parallel Nsight

NVIDIA Parallel Nsight [106] software is an integrated development environment (IDE) for General Purpose GPU (GPGPU) accelerated applications that works with Microsoft Visual Studio. Parallel Nsight is a powerful plug-in that allows programmers to debug and analyze both GPUs and CPUs applications within Microsoft Visual Studio.

Here again the efficient structures of a Laplace subroutine are shown in figure 7.1 for comparison with actual results from NVIDIA Parallel Nsight.

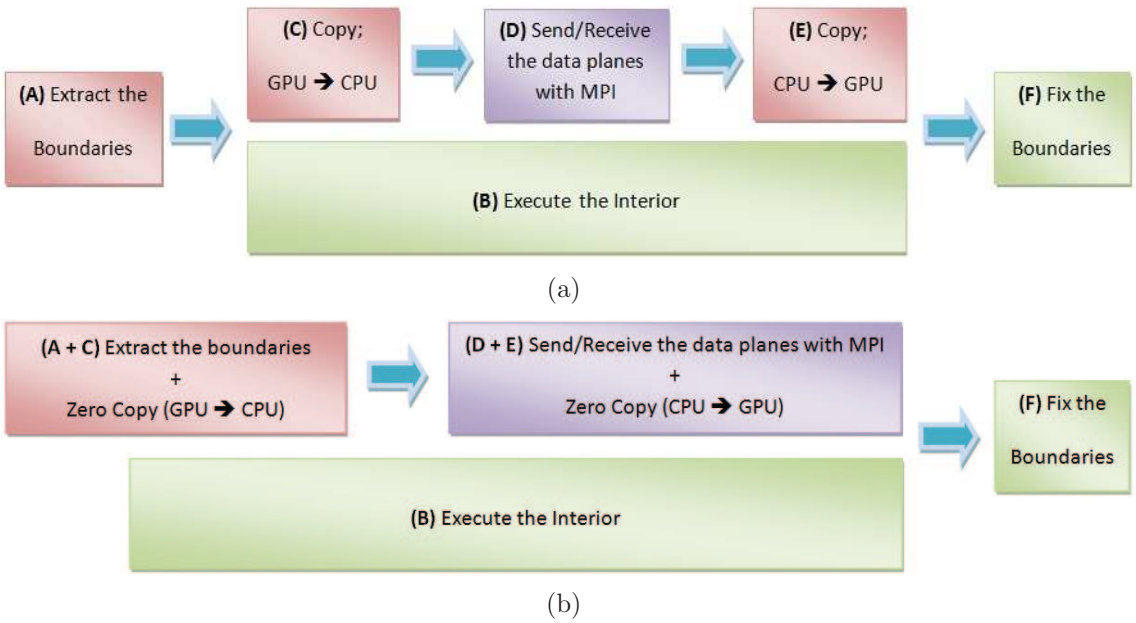


Figure 7.1. Efficient flow chart for Laplace, Gradient, Divergent, Convection and Laplace Inverse operators for (a) regular pinned memory (b) mapped memory; red, green and purple boxes are using stream 2, stream 1 and CPU respectively to execute the box

Figure 7.2 shows the timeline from NVIDIA Parallel Nsight software with Tesla C2070 GPU for the 64^3 for Laplace kernel with regular and mapped write-combined memories for send and receive buffers. The part (D) in figure 7.2(a) is the MPI send and receive time. This time is completely hidden with part (B) execution. Also the copying time ($71\mu s$) from GPU to the CPU (C) is completely hidden with internal calculation (B). But copying time ($74\mu s$) from GPU to CPU (E) is done after internal calculation and in theory this copy can be hidden by kernel execution. The major problem with 64^3 is the large (light green) GPU idle time ($360\mu s$) that is even larger than internal calculation time. The major reason for such a large gap between internal calculation (B) and fixing the boundaries (F) comes from the second *cuStreamSynchornize* ($210\mu s$). As mentioned before, *cudaThreadSynchornize* and *cudaStreamSynchornize* are very expensive. Every time the CPU hits a *cuStream-Synchornize* command, it stops there and waits for the GPU to finish an assigned

stream. The large gap between the last two *cuStreamSynchornize* calls is the time for loading the fix boundary code to the GPU and initializing this kernel. Because mapped memory is used here, there is no way to avoid using *cuStreamSynchornize* command. The only way to decrease the GPU idle time is hide both the copying time from CPU to the GPU and second *cuStreamSynchornize* call with kernel execution.

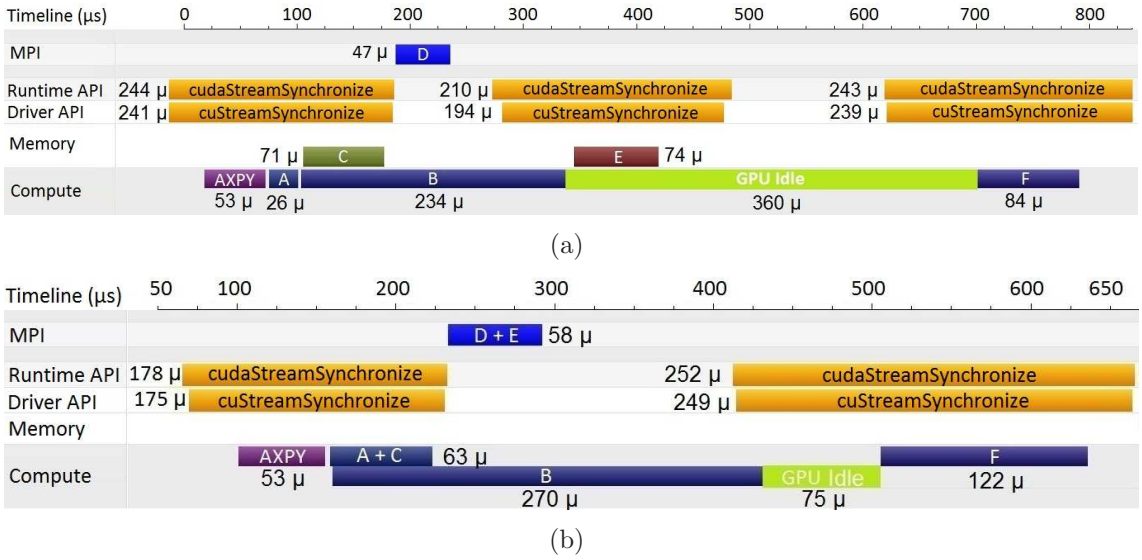


Figure 7.2. Timeline for Laplace kernel for 64^3 with (a) regular pinned memory and (b) mapped and write-combined memories for send and receive buffers

With mapped and write-combined memories for send and receive buffers, as shown in figure 7.2(b), the parts (A) and (C), and parts (D) and (E) are combined to one single transaction. mapped and write-combined memories are used for send and receive buffers, the time for (A) plus (C) is smaller than when regular pinned memory is used for these parts. Also the part (F) is faster in regular memory than write-combined memory. It is obvious that reading form regular pinned memory is faster than write-combined memory. Also figure 7.2 shows that part (B) in regular memory is faster than pinned memory case. The main reason for this deference is in mapped memory parts (A), (C) and (B) run concurrently on the GPU. So some of the resources (multiprocessors) are used to execute parts (A) and (C). We also should mention

that when write-combined memory is used for receive buffers, there is no need for *cuStreamSynchornize* after the MPI. Because the CPU runs synchronically and the GPU idle time is much more less than regular memory.

The total run time for 64^3 when pinned memory is used is $704\mu s$ and for the mapped memory is $467\mu s$. In this case mapped memory is almost 50% faster than pinned memory. The main reason for this difference, as mentioned before, is the second *cuStreamSynchornize* that stops GPU to load the part (F) to the GPU.

Figure 7.3 shows the timeline for the 128^3 for the Laplace kernel with regular and mapped memories for send and receive buffers.

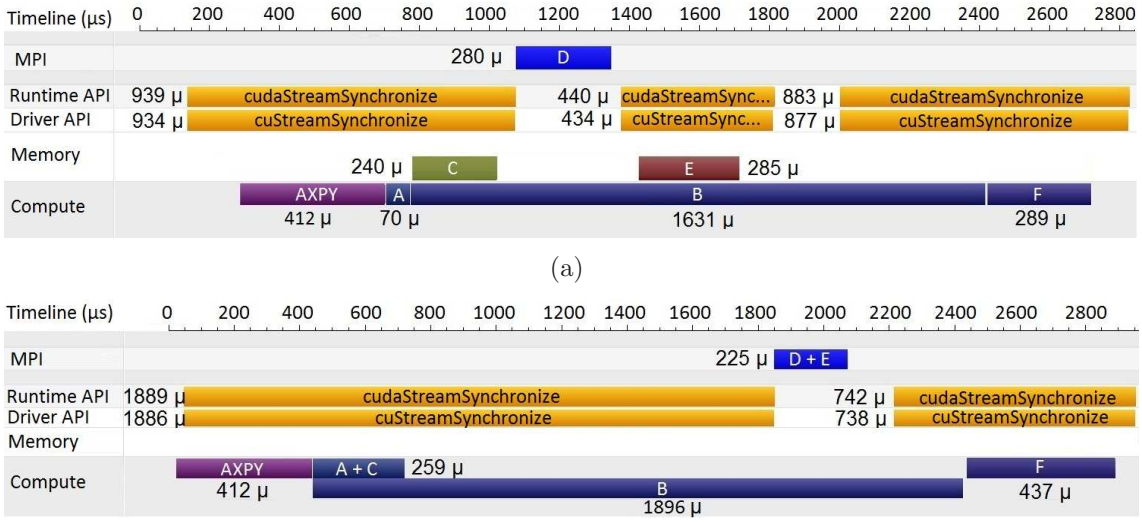


Figure 7.3. Timeline for Laplace kernel for 128^3 with (a) regular pinned memory and (b) mapped and write-combined memories for send and receive buffers

In this case, MPI and copying times are 4 times and interior calculation time is 8 times larger than 64^3. In communication the data between nodes, six boundary planes are sent and received $((128 \times 128)/(64 \times 64) = 4)$. But interior nodes in 128^3 is 8 times more than 64^3 $((128 \times 128 \times 128)/(64 \times 64 \times 64) = 8)$. Although this result is just for a single GPU and not many-GPUs, Stag++ uses periodic boundary

condition and MPI and copying data are always running even for a single GPU. But the network connection isn't probably used in this single GPU simulation. But there is almost a $700\mu s$ safety margin (the gap between the last two *cuStreamSynchornize* in figure 7.3(a)) for MPI when many nodes are used for the simulation.

Figure 7.3(b) shows that parts (A) and (C) are running fully concurrently with part (B). This feature is available only for the Fermi architecture. Also the total run time for 128^3 when pinned memory is used is $1990\mu s$ and for the mapped memory is $2332\mu s$. In this case pinned memory is almost 17% faster than mapped memory. The main reason for this difference, as mentioned before, writing and reading from mapped memory is expensive than regular pinned memory. So parts (A+C) and (F) in mapped and write-combined memories cases took much more time than part (A) and (F) in regular pinned memory (the part (C) in pinned memory is completely hidden with the part (B)). We should also mention that when mapped memory is used, there is small time margin available in order to hide MPI communications for large number of GPUs.

Figure 7.4 shows the timeline for the 256^3 for the Laplace kernel with regular and mapped memories for send and receive buffers. In this case, interior calculation is 4 times larger than MPI plus copying times. Also there is almost an $8500\mu s$ safety margin (the gap between the last two *cuStreamSynchornize* in figure 7.4(a)) for MPI when many nodes are used for the simulation.

Figure 7.4(b) shows that parts (A) and (C) are running partly concurrently with part (B) for the 256^3. The main reason for this is that parts (A) and (C) are using resources (blocks or multiprocessors) more than available in the device. So only some section of parts (A) and (C) are running concurrently on the GPU.

The total run time for 256^3 when pinned memory is used is $12403\mu s$ and for the mapped and write-combined memories is $14043\mu s$. In this case pinned memory is

almost 13% faster than mapped and write-combined memories. The same reason as 128^3 is true here.

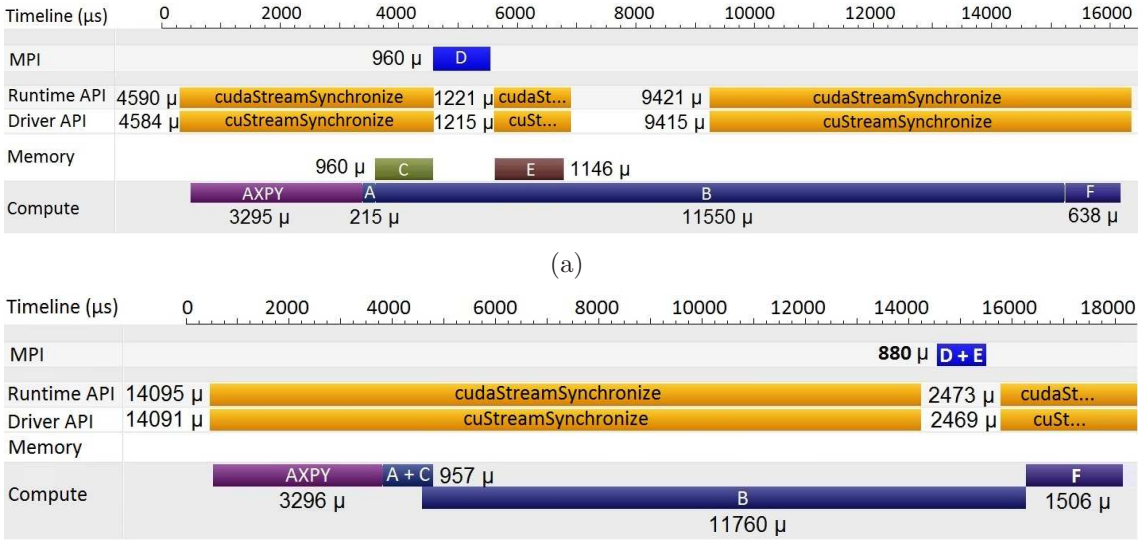


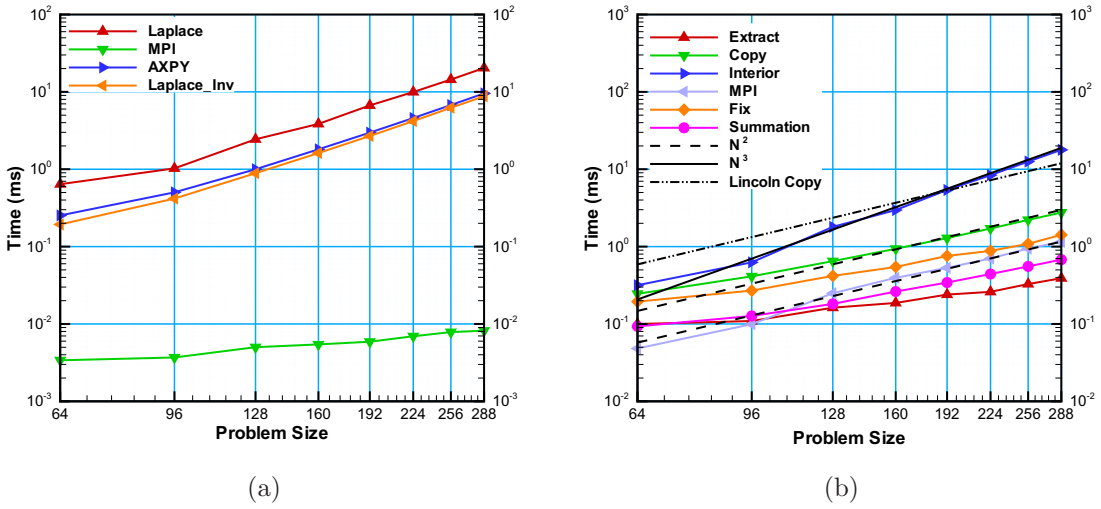
Figure 7.4. Timeline for Laplace kernel for 256^3 with (a) regular pinned memory and (b) mapped and write-combined memories for send and receive buffers

In general, if there is the possibility of hiding the copying time with kernel execution, it is efficient to use regular pinned memory instead of mapped memory (Mapped memory is useful only, one couldn't hide the copying time with kernel execution, like finding the summation or maximum value between all nodes).

As mentioned before, unlike the cache-based hardware (CPU), the GPU has much better performance on large data sets. Using this fact, and doubling the grid points in every direction, the GPU execution time for part (B) is increased 2 times more than the sending, receiving and copying time without any penalty. So if the MPI time becomes dominant in many-GPU simulations, it is possible to increase the number of grid points to 256^3 per GPU and hide the communication time with the kernel execution.

7.3 CG and Laplace Results for Single Processor on Orion

The performance of the code on a single GPU on Orion (our in-house GPU cluster) is more closely analyzed. Timings indicate that 87% of the code execution time is spent in the Conjugate Gradient (CG) solver (which solves for the pressure and implicit diffusion terms) and fully 50% of the time is spent in the sparse matrix multiply subroutines. A breakdown of the time spent in the CG and Laplace algorithms is shown in Figure 7.5.



(a) (b)

Figure 7.5. Time for (a) CG and (b) Laplace subroutines for different problem sizes

The summation item in the figure 7.5(b) includes copying the dot product results from GPU to CPU plus the last steps of summation on the CPU. This figure shows that interior (Part B) is the most time consuming part in the Laplace solver. Copying time plus MPI for largest cases is 4 times smaller than interior time. So on Orion, subdomain problem sizes of 128^3 per GPU and larger are sufficient to hide the MPI and copying time. On Lincoln, this is not actually true because the copy time is $4x$ slower (dashed black line),it always is as large as the useful computation time. Current GPUs do not have enough memory to handle problem sizes greater than

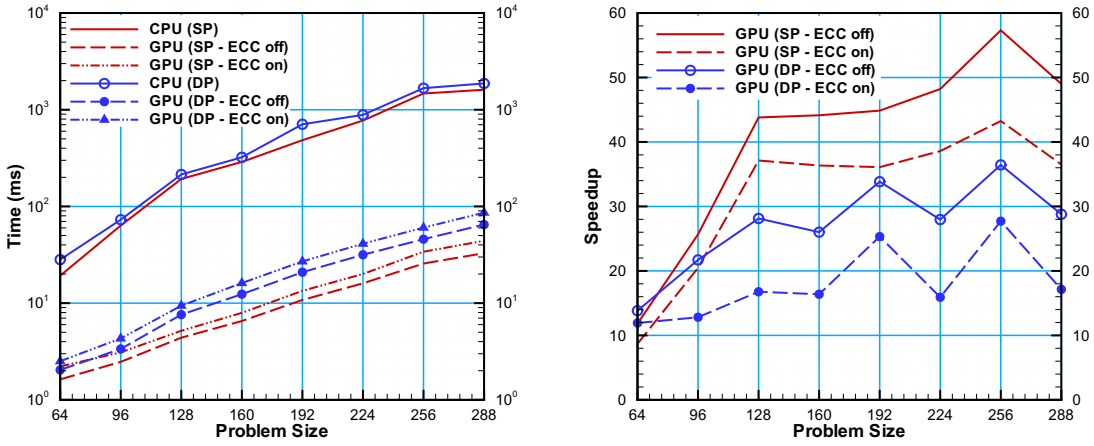
288^3 per GPU. Because every node has to send data on its boundaries to other nodes, it needs to copy six boundary surfaces to CPU. So parts (A), (C), (D) and (E) scale like N^2. But part (B), solving the interior points, grows like N^3. As mentioned before, Lincoln has 4 times slower bandwidth between the CPU and GPU. Figure 7.5(b) shows the extrapolated time for copying between the CPU and GPU on Lincoln without MPI time. As you can see even for 256^3 copying times are barely overlapped with domain computation.

7.4 Single CPU and GPU Results for Different Computers

In the single processor case, although there is no MPI communication via network card, there are four copies, two on GPU side and one on CPU, because of periodic boundary conditions. These copying times should be hidden for single processor. Two copies on GPU use PCI-e bandwidth speed. The maximum theoretical PCI-e speed for $\times 4$, $\times 8$ and $\times 16$ are 2, 4 and 8 GB/s. Single copy on CPU is dependent on each CPU memory bandwidth that is different for each CPU architecture.

7.4.1 Orion Single Processor Results

Figure 7.6 shows time per iteration for a single CPU and GPU and the speedup for single GPU on Orion for single and double precision with ECC on and off options.



(a) (b)

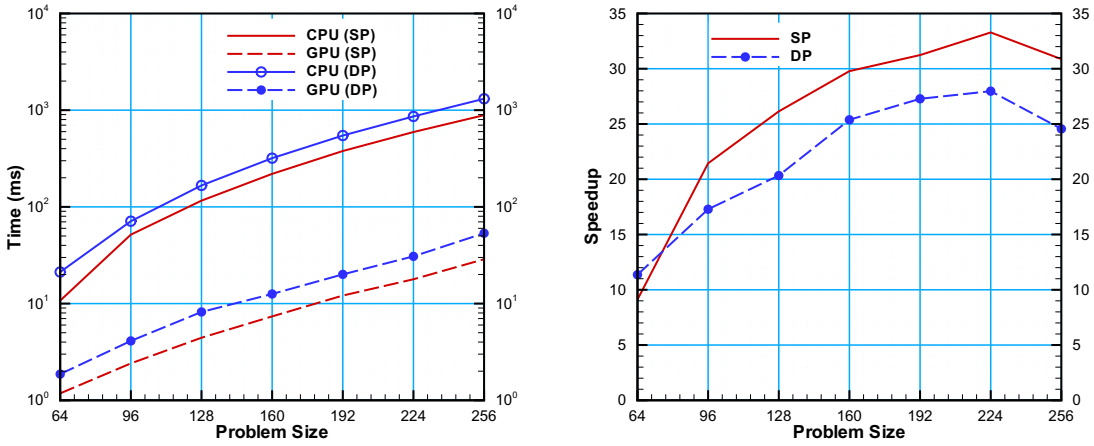
Figure 7.6. Single CPU and GPU results, (a) Time (ms) per iteration and (b) Speedup for different problem sizes on Orion with single (SP) and double (DP) precision

The highest speedup (57×) and (36×) occurred for single and double precision when the ECC is off, respectively. It seems that there is small difference in timing results for CPU for single and double precision. The one possible reason for this small difference is 64-bit operating system (OS). Orion has 64-bit CPU and OS. It seems on 64-bit architecture there is small difference between single and double precision. But in GPU case, even for the latest generation of GPUs, single precision is 2 times faster than double precision. As mentioned earlier, Stag++ is mostly throughput (bandwidth) limit. So reading a double precision data from GPU memory takes 2 times longer than a single precision.

Figure 7.6 shows that when the ECC is on, performance is dropped by 30%. We also examined the effect of ECC on the GPU's performance (it is not shown here). When the ECC in on, the raw bandwidth is reduced by 12.5% for Tesla C2070.

7.4.2 Lincoln Single Processor Results

Figure 7.7 shows time per iteration for a single CPU and GPU and the speedup for single GPU on Lincoln supercomputer for single and double precision.



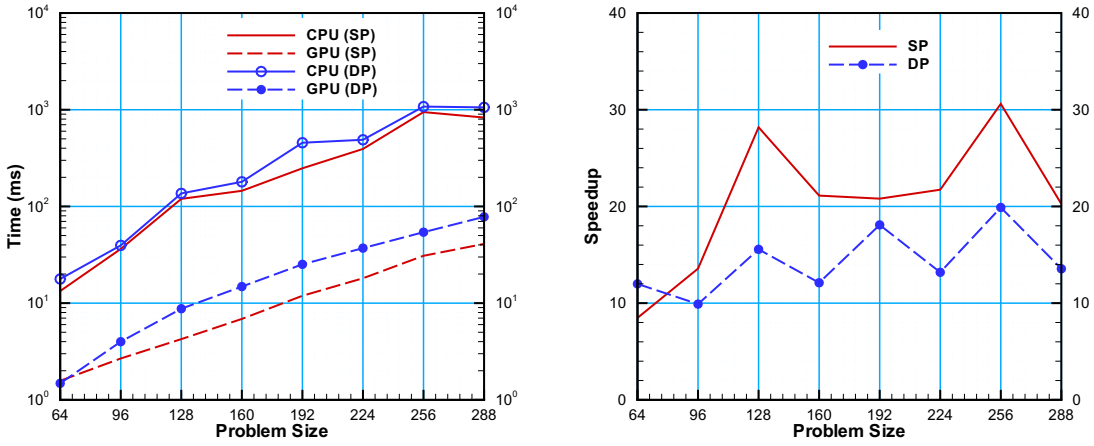
(a) (b)

Figure 7.7. Single CPU and GPU results, (a) Time (ms) per iteration and (b) Speedup for different problem sizes on Lincoln supercomputer with single (SP) and double (DP) precision

The highest speedup (33×) and (28×) occurred for single and double precision respectively. Lincoln uses Tesla 10 series GPUs. In this case single precision is 2 times faster than double precision like the Tesla 20 series. As mentioned before it seems that there is small difference in timing results for CPU for single and double precision.

7.4.3 Forge Single Processor Results

Figure 7.8 shows time per iteration for a single CPU and GPU and the speedup for single GPU on Forge supercomputer for single and double precision.



(a) (b)

Figure 7.8. Single CPU and GPU results, (a) Time (ms) per iteration and (b) Speedup for different problem sizes on Forge supercomputer with single (SP) and double (DP) precision

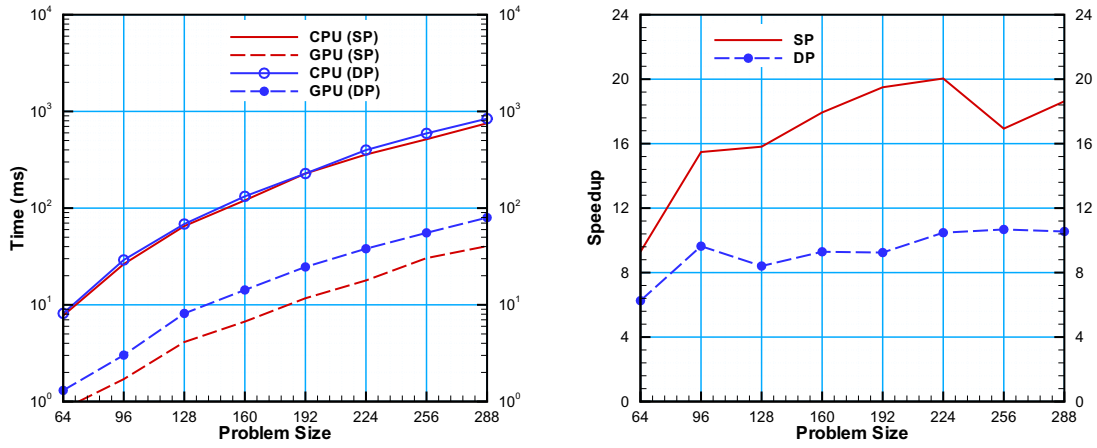
Forge has the same GPU as Orion, but uses a newer CPU core architecture. Forge hold two AMD Opteron Magny-Cours 6,136 with 2.4 GHz dual-socket eight-core and Orion hold AMD quad-core Phenom II X4 CPU, operating at 3.2 GHz.

Figure 7.6(a) and 7.8(a) show same behavior for CPU because of the AMD works in same way. The results for SP and DP for CPUs on Forge are roughly same order. Figure 7.8(b) shows speedup for single GPU on Forge supercomputer for single and double precision. The highest speedup (31×) and (20×) times for Forge occurred for single and double precision, respectively.

7.4.4 Keeneland Single Processor Results

Figure 7.9 shows time per iteration for a single CPU and GPU and the speedup for single GPU on Keeneland supercomputer for single and double precision. Keeneland and Lincoln, both have Intel CPUs, but Keeneland uses a new CPU core architecture.

Keeneland hold two hex-core Intel Xeon (Westmere-EP) 2.93 GHz and Lincoln hold two Intel 64 (Harpertown) 2.33 GHz dual socket quad-core processors.



(a) (b)

Figure 7.9. Single CPU and GPU results, (a) Time (ms) per iteration and (b) Speedup for different problem sizes on Keeneland supercomputer with single (SP) and double (DP) precision

Figures 7.7(a) and 7.9(a) show the same behavior for CPUs because they are both Intel. Keeneland also has the same GPU as Orion and Forge, but use different CPUs (Intel instead of AMD). The results for SP and DP for CPUs are roughly same order. Figure 7.9(b) shows speedup for single GPU on Keeneland supercomputer for single and double precision. The highest speedup (20×) and (11×) times for Keeneland occurred for single and double precision respectively.

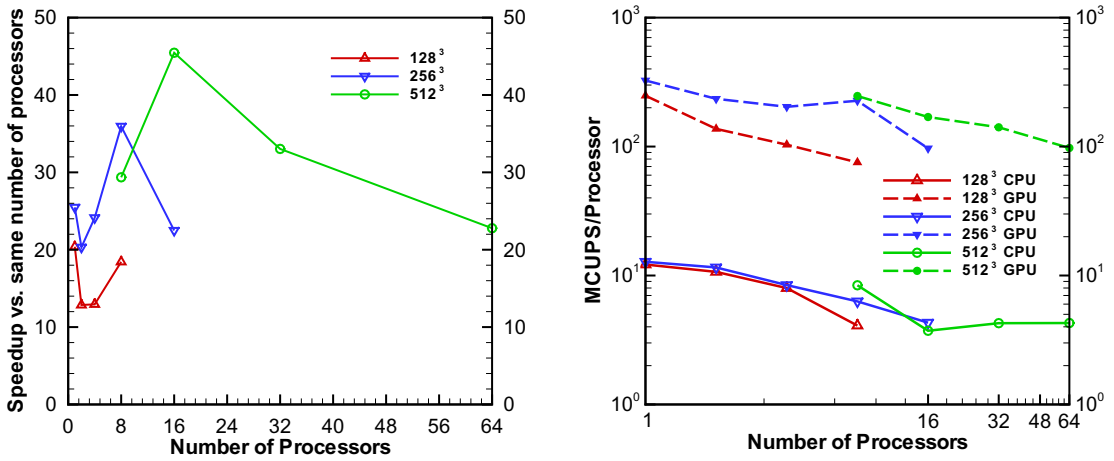
7.5 Strong Scaling Results

In the strong scaling situation, the problem size is constant, and as the number of processors in increased, the problem size per processor gets smaller and smaller. Below the results for three different supercomputers. The Lincoln, Forge and Keeneland have ×4, ×8 and ×16 PCI-e bandwidth speed when all GPUs per node are used

respectively. The PCI-e bandwidth is a very important key in large high performance computing codes.

7.5.1 Lincoln Strong Scaling Results

Figure 7.10 shows the speedup (versus the same number of CPU cores) and millions of cell updates per second per GPU or CPU core (MCUPS/Processor) for strong scaling results.



(a) (b)

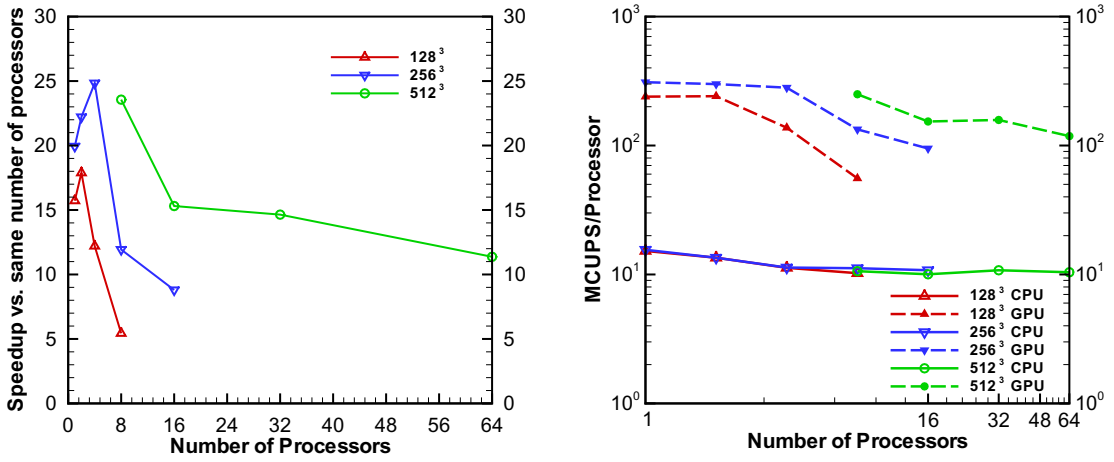
Figure 7.10. (a) Speedup and (b) Performance per processor for strong scaling of the 128^3, 256^3 and 512^3 CFD problem on Lincoln supercomputer using GPUs and CPUs

A MCUPS represents how many millions of finite-volume cells can be updated (one CG iteration) during a second of wall-clock time. For strong scaling, the highest speedup (45×) occurred for 16 GPUs compared to 16 cores (on 4 CPUs). In theory, the MCUPS/Processor (is directly related to the hardware efficiency) and should look like a horizontal line (a constant). Figure 7.10 shows that from 2 GPUs to 64 GPUs the performance loss is roughly 50%. (One GPU and one core can each access more memory bandwidth and therefore perform better than when the memory

system is loaded to its typical state). We should mention that using 64 GPUs in Lincoln means using 32 nodes and 4 times more network traffic than 64 CPU cores (which only requires 8 nodes). Also Figure 7.10(b) shows that with increasing the number of processors, the performance is increased and decreased for CPU and GPU. As mentioned before GPUs are more efficient for large problem sizes and CPUs are small problem sizes.

7.5.2 Forge Strong Scaling Results

Figure 7.11 shows the speedup (versus the same number of CPU cores) and MCUPS/Processor for strong scaling results.



(a) (b)

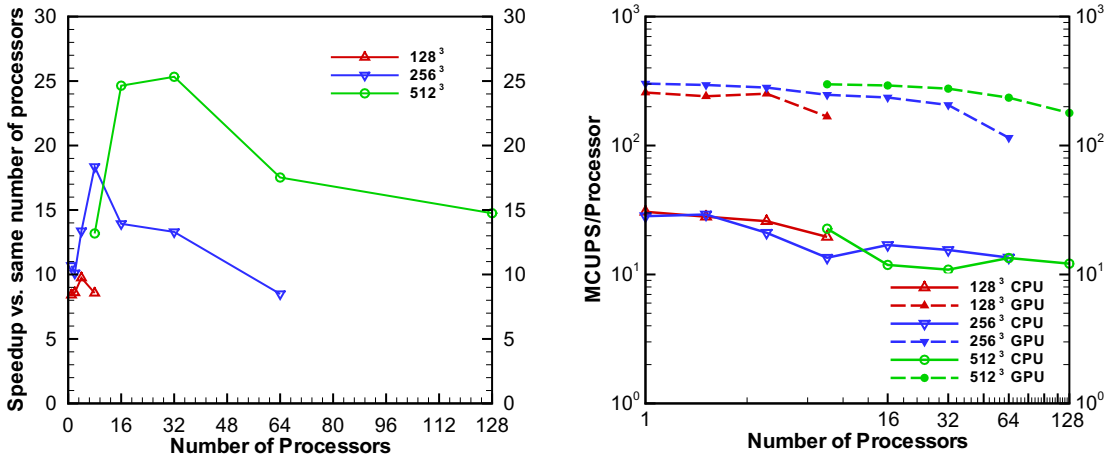
Figure 7.11. (a) Speedup and (b) Performance per processor for strong scaling of the 128^3, 256^3 and 512^3 CFD problem on Forge supercomputer using GPUs and CPUs

For strong scaling, the highest speedup (24×) occurred for 4 GPUs compared to 4 cores. Figure 7.11 shows that for 256^3 from single GPU to 4 GPUs the performance is perfect. From 4 to 8 the performance loss is roughly 50%. The main reason for this loss is again PCI-e bandwidth. When 4 GPUs per node are used, the PCI-e bandwidth is ×16 for each GPU. But When 8 GPUs per node are used, the PCI-e bandwidth

is ×8 that is two times slower than ×16. Because there are 8 GPUs per node on Forge, for 512^3, form 8 GPUs to 16 GPUs, MPI times increases and there is almost 50% performance loss in this case. But CPU has almost steady performance for all cases. Figure 7.11 also shows that for CPU cases almost all MPI communications are hidden by useful computation.

7.5.3 Keeneland Strong Scaling Results

Figure 7.12 shows the speedup and MCUPS/Processor for strong scaling results. For strong scaling, the highest speedup (25×) occurred for 32 GPUs compared to 32 cores.



(a) (b)

Figure 7.12. (a) Speedup and (b) Performance per processor for strong scaling of the 128^3, 256^3 and 512^3 CFD problem on Keeneland supercomputer using GPUs and CPUs

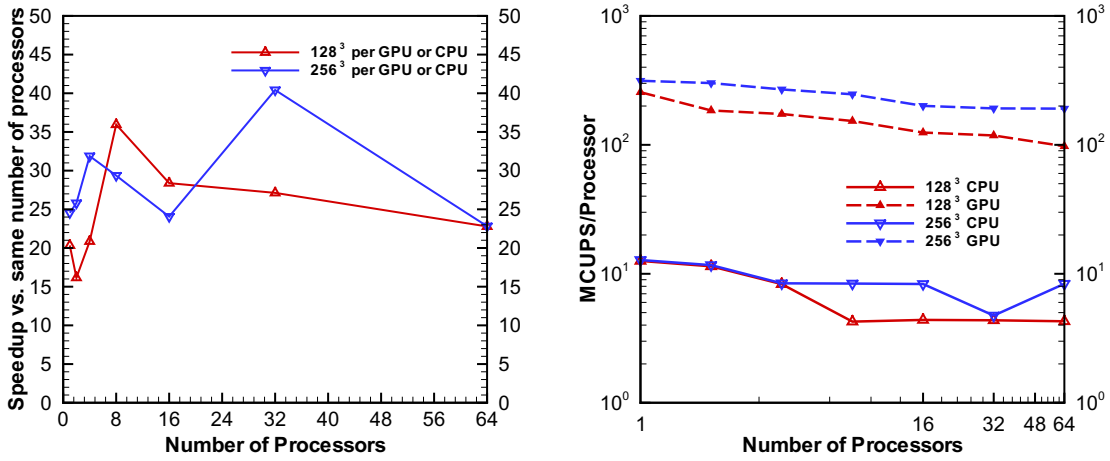
Figure 7.12 shows that for 256^3 from single GPU to 4 GPUs the performance is perfect. From 4 to 16 the performance loss is roughly 25%. The main reason for this loss is problem is getting smaller with increasing the GPUs number.

7.6 Weak Scaling Results

In the weak scaling situation, the problem size is constant per processor as the number of processors in increased, the communication time (MPI time) gets larger and larger. Below the results for three different supercomputers, Lincoln, Forge and Keeneland.

7.6.1 Lincoln Weak Scaling Results

Figure 7.13 shows the speedup (versus the same number of CPU cores) and MCUPS per GPU or CPU core (MCUPS/Processor) for weak scaling results.



(a) (b)

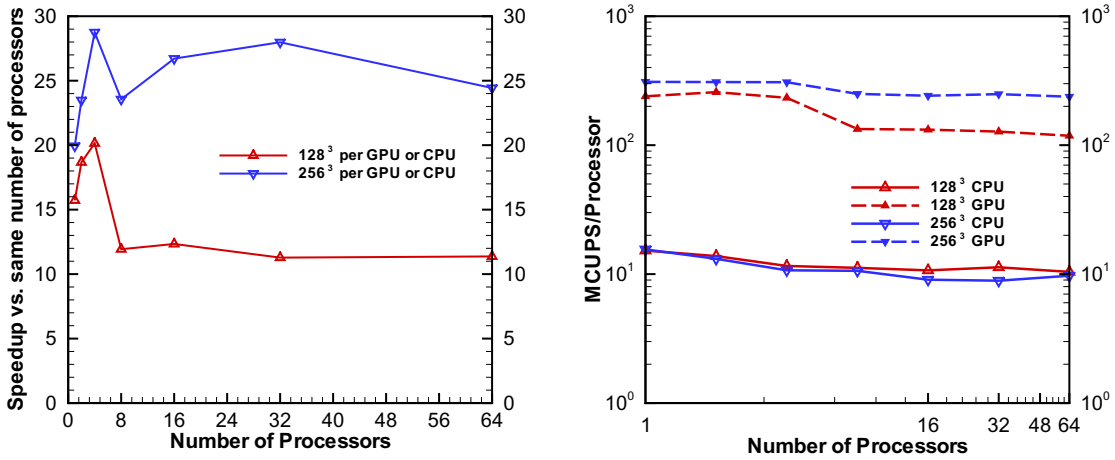
Figure 7.13. (a) Speedup and (b) Performance per processor for weak scaling of the 128^3 and 256^3 CFD problem on Lincoln supercomputer using GPUs and CPUs

For weak scaling, the highest speedup (40×) occurred for 32 GPUs compared to 32 cores (on 8 CPUs). In theory, the MCUPS/Processor should look like a horizontal line (a constant). Figure 7.13 shows that from 2 GPUs to 64 GPUs the performance loss is roughly 50%. The main reason for this loss in efficiency is low bandwidth between CPU and GPU (PCI-e ×4).

The main reason for this loss in efficiency is low bandwidth between CPU and GPU (PCI-e ×4). As mentioned earlier, when the copying times increased, there is not enough time to hide MPI communication time with computation.

7.6.2 Forge Weak Scaling Results

Figure 7.14 shows the speedup (versus the same number of CPU cores) and MCUPS per GPU or CPU core (MCUPS/Processor) for weak scaling results.

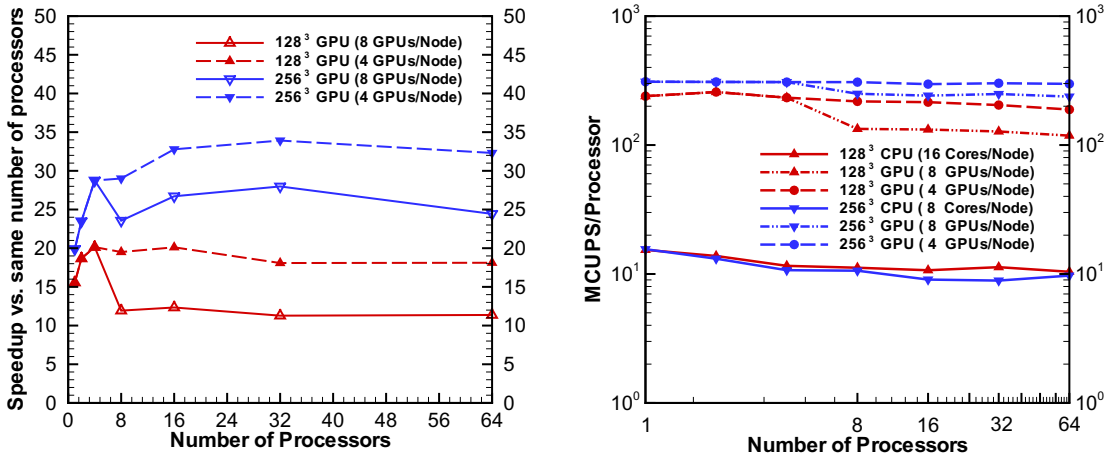


(a) (b)

Figure 7.14. (a) Speedup and (b) Performance per processor for weak scaling of the 128^3 and 256^3 CFD problem on Forge supercomputer using GPUs and CPUs

For weak scaling, the highest speedup (29×) occurred for 4 GPUs compared to 4 CPU cores. In theory, the MCUPS/Processor should look like a horizontal line (a constant). As mentioned earlier, up to 4 GPUs, the MCUPS/Processor is a horizontal line. But after 4 GPUs per node, with increasing the GPUs, the PCI-e bandwidth is reduced to ×8 and the performance loss is roughly 50%. Like strong scaling, CPU shows stable performance and MPI communication are completely hidden with computation.

Figure 7.15 shows the comparison for speedup (versus the same number of CPU cores) and MCUPS/Processor for weak scaling results with maximum of 4 and 8 GPUs per node. For weak scaling, the highest speedup (34×) versus (29×) occurred when 4 GPUs per node used instead of 8 GPUs with total 32 GPUs compared to 32 CPU cores. As mentioned earlier,with 4 GPUs per node, the PCI-e bandwidth is ×16 and there is no performance loss for 256^3 case. There is a performance loss for 128^3 per GPU for 64 GPUs. It seems that performance loss is related to MPI not PCI-e bandwidth.

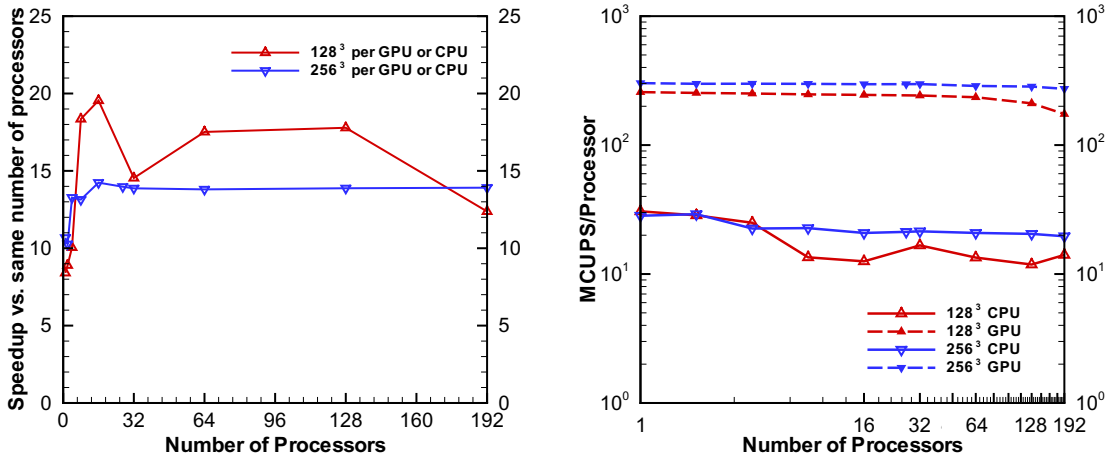


(a) (b)

Figure 7.15. (a) Speedup and (b) Performance per processor for weak scaling of the 128^3 and 256^3 CFD problem on Forge supercomputer using 4 GPUs per node and 8 and 16 CPU cores per node for 256^3 and 128^3 respectively

7.6.3 Keeneland Weak Scaling Results

Figure 7.16 shows the speedup (versus the same number of CPU cores) and MCUPS/Processor for weak scaling results. For weak scaling, the highest speedup (21×) occurred for 8 GPUs compared to 8 CPU cores. Figure 7.16(b) shows perfect performance for the GPU up to 192 GPUs. It means that all MPI communications are completely hidden with kernel executions. For the CPU case, 8 and 4 GPUs are used for 128^3 and 256^3 cases, respectively. So for 128^3 the performance is decreased up to 8 CPU cores, and after that the performance is almost constant. But for 256^3, the performance is decreased up to 4 CPU cores, and after that the performance is almost constant. Also there is performance lost in figure 7.16(b) for 128^3 per GPU. The main reason for this lost that we used 3 GPUs per node with slow bandwidth. And also because the problem size is small, the MPI and copying are barely hidden with kernel executions.

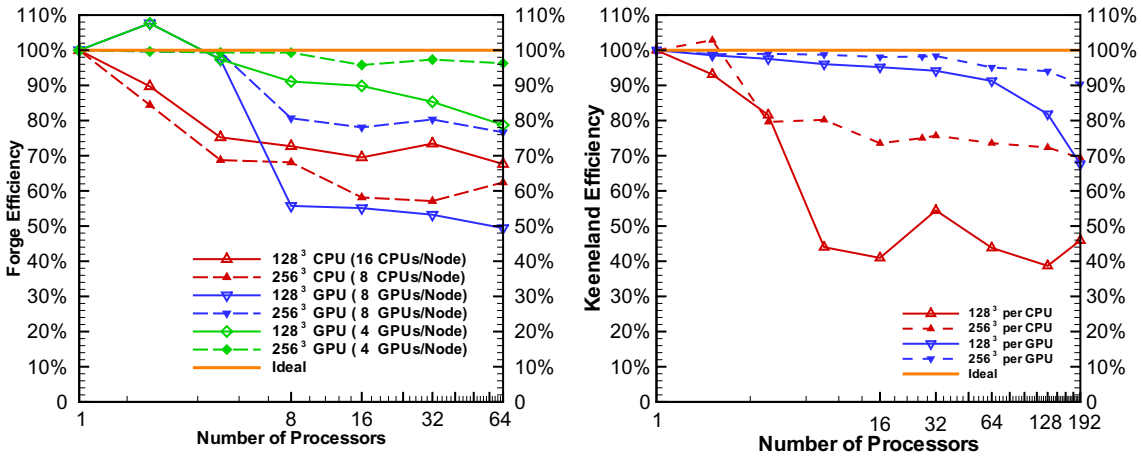


(a) (b)

Figure 7.16. (a) Speedup and (b) Performance per processor for weak scaling of the 128^3 and 256^3 CFD problem on Keeneland supercomputer using GPUs and CPUs

7.7 Forge and Keeneland Supercomputers Efficiency Results

Figure 7.17 shows the MCUPS/preocessor versus single GPU for weak scaling results for the Forge and Keeneland supercomputers. This figure represents a efficiency of the code and supercomputer. Figure 7.17(a) and 7.17(b) show that for 256^3 per GPU up to 64 and 192 the efficiency reduced by 4% and 10% for the Forge and Keeneland supercomputers respectively.



(a) (b)

Figure 7.17. (a) Forge and (b) Keeneland efficiency results for weak scaling of the 128^3 and 256^3 CFD problem using GPUs and CPUs

Figure 7.17(a) shows that number of GPUs per node has a significant effect on the efficiency for the Forge supercomputer. As mentioned before, when 8 GPUs are used per node, the PCI-e speed is reduced to ×8 instead of ×16. The results also show that PCI-e ×16 is good enough to hide copying and MPI communications in large CFD codes, if suitable approaches are applied.

There is different philosophy behind the CPU results. All CPUs work fine up to number of memory pipelines. After that the performances is decreased up to number of cores per CPU are used. Because like the GPU, all MPI communications are hidden, CPU results show roughly stable performance with increasing the number of

processors. But we should also mention that all of these conclusions based on the memory bandwidth limited codes. All CFD and large portion of scientific codes are memory bound.

CHAPTER 8

DIRECT NUMERICAL SIMULATION OF TURBULENCE

8.1 Introduction

The direct numerical simulation (DNS) of turbulence is a computationally intensive scientific problem that can benefit significantly from improvements in computational hardware performance. The memory streams in a direct numerical solution of turbulence, such as a pressure or velocity field, are on the order of 1 billion bytes each.

We would like to efficiently stream this data in, do a few computations and return the same field but at the next time level. Because processor speeds have been increasing over the last decade but memory speeds have not, all CFD simulations (and in fact almost all PDE solution techniques) are entirely memory bound. This means, that on modern computers the computations are not important to the performance of the algorithm. The critical factor is the ability to read data in, and write results out. GPU memory subsystems do this an order of magnitude more quickly than CPU memory subsystems based on caches. Perhaps not too surprisingly, the GPU memory subsystem functions (and is optimized) remarkably similarly to the memory subsystem of the original Cray supercomputer vector processors.

8.2 Software

The solution method uses a three-step, low-storage Runge-Kutta scheme [107] for time advancement that is second-order accurate in time. This scheme is stable for eigenvalues on the imaginary axis less than 2, which implies $CFL < 2$ for advective

stability. The simulations always use a maximum $CFL < 1$. The diffusive terms are advanced with the trapezoidal method for each Runge-Kutta substep, and the pressure is solved using a classical fully-discrete fractional step method [108], although an exact fractional step method [109] is also possible.

For the spatial discretization, a second order Cartesian staggered-mesh scheme is used. This not only conserves mass and momentum to machine precision, but because it is a type of Discrete Calculus method [110] it also conserves vorticity (or circulation) and kinetic energy in the absence of viscosity. As a result, there is no artificial viscosity/diffusion in this method except that induced by the time-stepping scheme [111]. In addition, the staggered mesh discretization is free from spurious pressure modes and the need for pressure stabilization terms. This discretization method also treats the wall boundary condition well because the wall normal velocity unknown lies exactly on the wall, so no interpolation is required to enforce the kinematic no penetration condition. Higher order versions of this method exist but are more complicated to parallelize [112].

Many of the simulations presented below were performed on 512^3 meshes with fully periodic boundary conditions on the exterior of the computational domain, and wall boundary conditions on interior embedded objects. The highest Reynolds numbers simulated in this work are comparable to the Reynolds numbers found in laboratory wind tunnel turbulence experiments (such as Comte-Bellot and Corrsin [113, 114]). In addition, the highest Reynolds numbers tested in this work are sufficient to show decay rates that are very consistent with high Reynolds number decay theories [115]. The simulations of turbulence are initialized by driving fluid past stationary "mixing boxes". This initialization procedure has the advantage of allowing the initial turbulence spectrum to develop naturally rather than being imposed as an initial condition. Further details of the simulations can be found in Perot [115].

Physical units can be helpful for the reader to put the simulations in perspective. If the simulated fluid is water at standard temperature and pressure (with $\nu = 10^{-6}$) then the domain size is a cube that is $48cm$ on a side. The small cubes that initialize the turbulence are $1.4cm$ on a side. In the 512^3 simulations there are 768 initialization cubes randomly placed in the domain (Figure 8.1). The total volume of all the stirring elements is therefore 1.92% of the total simulation volume. The mesh size itself is $0.9375mm$ (which is $1/15^{th}$ of the stirring cube size). At early times in the simulation, the timestep can be as small as $1/1000^{th}$ of a second. In all the simulations, the timestep is never larger than a $1/10^{th}$ of a second.

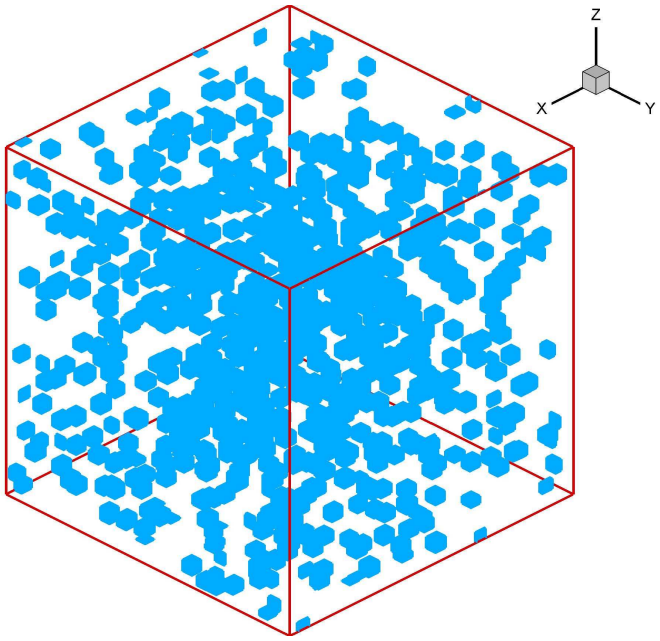
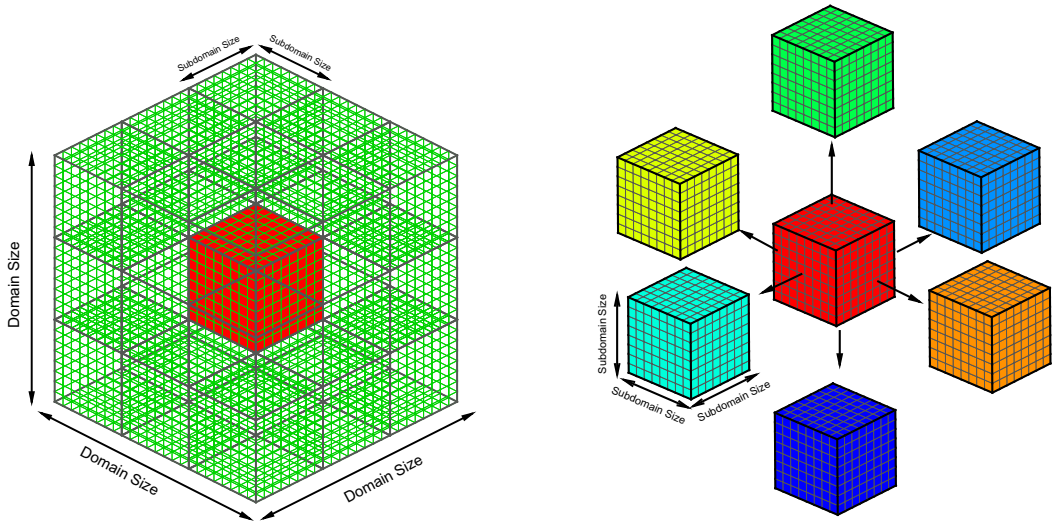


Figure 8.1. Simulation domain with 768 randomly distributed cubes

8.3 Partitioning

All PDE solution methods ultimately involve placing a large number of unknowns (which approximate the solution) into a 3D domain and evolving those unknowns in time. Parallel solution algorithms explicitly or implicitly partition these unknowns among the available processing units. Because of the local nature of many PDEs it is

often advantageous if the partitioning is performed so that the unknowns are grouped into physical clusters that are spatially coherent. These are called subdomains (See figure 8.2). For a Cartesian mesh, this type of partitioning is not a difficult task, and results in the large cuboid computational domain being divided into smaller cuboid subdomains. Each subdomain is allocated to one GPU, and the each GPU communicates via MPI with its 6 local neighbors. A few calculations, such as a sum(), require very small amounts of data to be globally communicated between all the GPUs. For example, for a 512^3 simulation running on 64 GPUs, each GPU solves a 128^3 subdomain problem (2 Million mesh points), but it communicates only the data at the surface of that subdomain with its six neighbors. So $6 \times 128^2 = 0.1$ Million data items (or about 5% of the data) is communicated from/to each GPU. As mentioned before, most of the communication instructions (MPI) can be overlapped with regular computations, so communication is hidden and does not take any extra time to perform.



(a) (b)

Figure 8.2. (a) Domain and (b) Subdomains with boundary planes for MPI communication

Although the boundary data is typically small ($< 10\%$) compared to the internal data of the partitions, the MPI communication operations are also typically very slow ($> 10\times$ slower) than the internal operations. It is therefore quite possible for the boundary operations, and not the large number of bulk internal operations, to dominate the solution time and dictate the scaling behavior of the code. When the boundary operations dominate, the code speeds up by roughly a factor of $2^{2/3} = 1.59$ when the number of processors is doubled (rather than producing the expected doubling of the speed).

8.4 Isotropic Turbulence Decay

In order to generate turbulence flow, we initially moved fluid from randomly distributed small no-slip cube boxes (Figure 8.1). Because small boxes are distributed randomly, it is possible that two or more cubes intersect with each other. This is allowed and is computed correctly. The small cubes remain fixed in the domain and an external (constant in space) acceleration is applied to the fluid to force it past the cubes for almost 5s. The direction of this acceleration is random, but its magnitude is chosen by the user. Two values of the acceleration are applied, 80 and 100. In these simulations the direction of the acceleration is changed to a new random direction (with the same magnitude) every 0.3 seconds [115].

The primary acceleration is then turned off from 5s to 7s. This is the final motion of the domain back to its rest position. After 2 seconds this restoring acceleration causes the mean flow to be extremely close to zero. A mean flow of zero is not necessary for the code, but it helps the solver to take slightly larger timesteps (by minimizing the CFL stability criteria), and it seems to lead to better statistical accuracy at very long times. During this 2 second time period the turbulence changes from being accelerated to being in isotropic decay. At the end of this period (when the mean flow is zero), the boxes instantaneously turn into (zero velocity) fluid [115].

Because all boxes are removed from the domain at 7 seconds of the simulation, we leave the flow up to 12 seconds to be relaxed. Now after 12 seconds the turbulent flow is in isotropic decay.

Figure 8.3 shows the validation for the turbulent kinetic energy (TKE) with the de Bruyn Kops and Riley result [12].

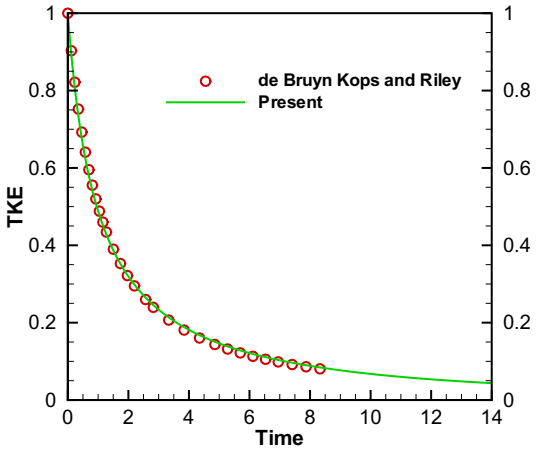
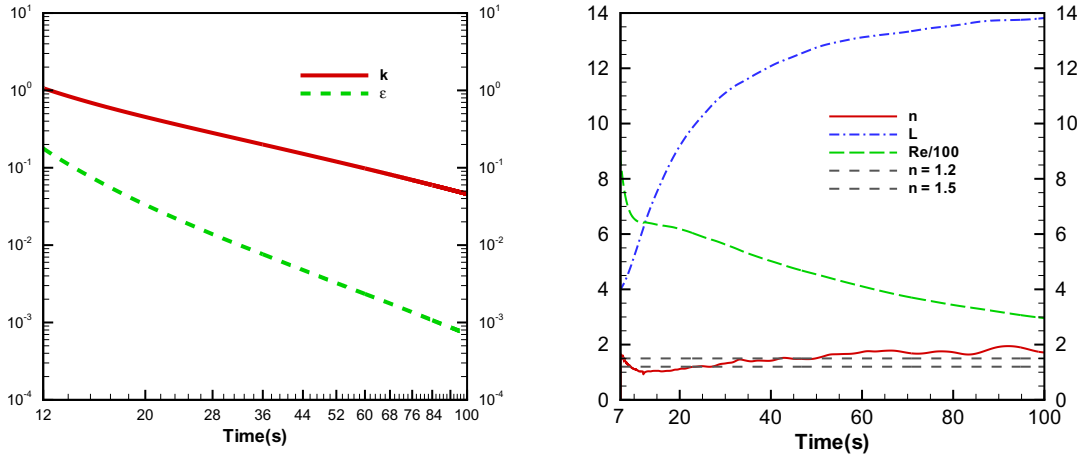


Figure 8.3. Validation of TKE with de Bruyn Kops and Riley result [12]

Figure 8.4 shows the TKE, ε, Re number, large-eddy length scale and decay exponent for isotropic decay.

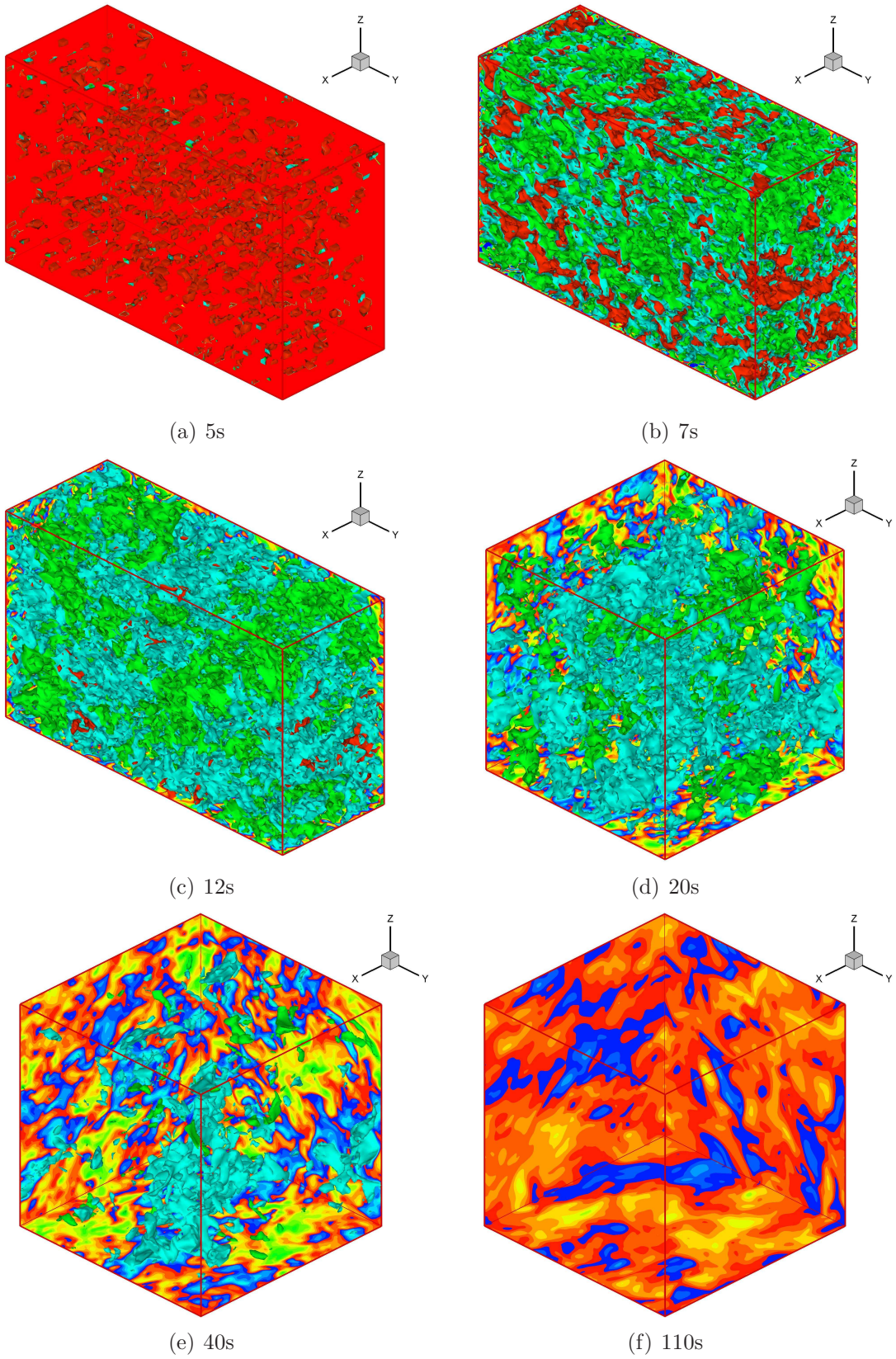


(a) (b)

Figure 8.4. (a) TKE and ε and (b) Re number, large-eddy length scale and decay exponent for isotropic decay

Four different plain strain cases were also executed. In this case Initial domain is a rectangular cube. The same procedure is applied up to 12 seconds. From 12 seconds up to specified time by user plane strains are applied in X and Y directions. In this case, four different strain rates are used; 0.025 (ST = 0.4), 0.0625 (ST = 1), 0.15625 (ST = 2.5) and 0.625 (ST = 10). The amount of strain time is chosen in a way that at the end of strain cases, the rectangular cube became a square cube. So plane strain was applied up to 32, 20, 15.2 and 12.8 seconds for ST = 0.4, 1, 2.5 and 10, respectively.

Figure 8.5 shows the turbulence decay form 5s to 110s for plain strain 1 case.



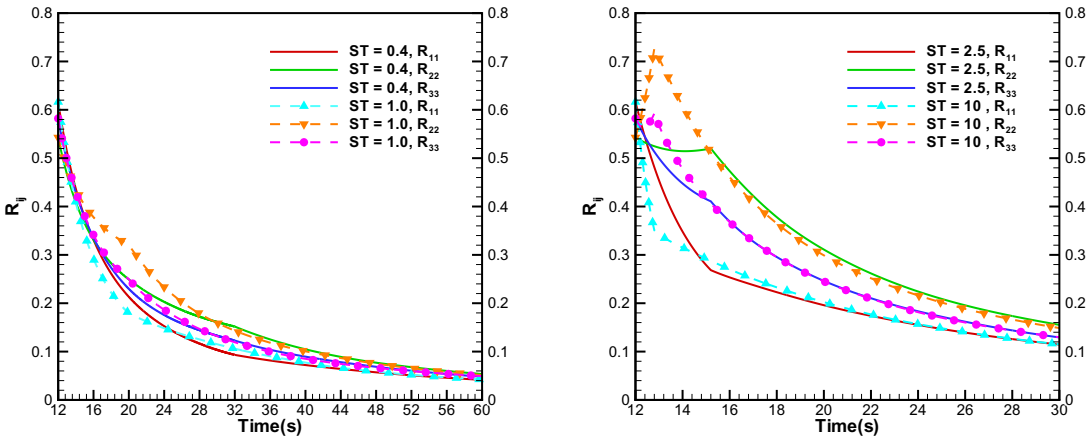
(a) 5s (b) 7s

(c) 12s (d) 20s

(e) 40s (f) 110s

Figure 8.5. u velocity contours for isotropic turbulence decay in (a) 5s (b) 7s (c) 12s (d) 20s (e) 40s and (f) 110s for plane strain case 1

Figure 8.6 shows the diagonal Reynolds stress component for all four cases.



(a) (b)

Figure 8.6. Diagonal Reynolds stress component for (a) ST = 0.4 and 1 (b) ST = 2.5 and 10 cases

Figure 8.7 shows the TKE and ε for all four plain strain cases.

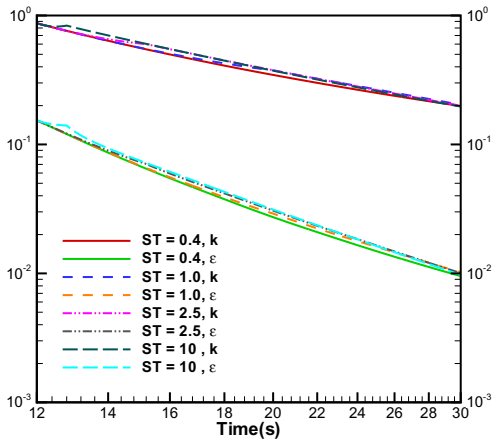


Figure 8.7. TKE and ε for different plain strain cases

CHAPTER 9

CONCLUSION

The research presented in this document demonstrates several notable advances in the area of high performance computing with many-core GPUs, especially in the area of Bioinformatics and Computational Fluid Dynamic (CFD). In particular, the aspects of this dissertation can be summarized in three different areas; Bioinformatics, CFD and GPUs.

9.1 Bioinformatics (Sequence Matching)

Finding regions of similarity between two very long data streams is a computationally intensive problem referred to as sequence alignment. The well-known Smith-Waterman algorithm is modified and used in his research. In particular, the advances in Bioinformatics research can be summarized as follows;

- In this work it was shown that effective use of the GPU requires a novel reformulation of the SmithWaterman algorithm. In order to accomplish sequential memory accesses a novel row (or column) parallel version of the SmithWaterman algorithm was formulated. The performance of this new version of the algorithm was demonstrated using the SSCA#1 (Bioinformatics) benchmark running on one GPU and on up to 120 GPUs executing in parallel. The results indicate that for large problems a single GPU is up to 105 times faster than one core of a CPU for this application, and the parallel implementation shows almost linear speed up on up to 120 GPUs (Ali Khajeh-Saeed and J. Blair Perot [79]).

- The issue of programming multiple GPUs is interesting because it requires a completely different type of parallelism to be exploited. A single GPU functions well with massive fine grained (at least 30 k threads) nearly SIMT parallelism. With multiple GPUs which are potentially on different computers connected via MPI and Ethernet cards, very course grained MIMD parallelism is desired. For this reason, all our multi-GPU implementations partition the problem coarsely for the GPUs, and then use fined grained parallelism within each GPU (Ali Khajeh-Saeed, Steve Poole and J. Blair Perot [49]).

- Performance increases of an order of magnitude over a conventional high-end CPU are possible on the SmithWaterman algorithm when graphics processors are used as the compute engine. Like most scientific algorithms, the Smith-Waterman algorithm is a memory bound computation when the problem sizes become large. The increased performance of the GPUs in this context is due almost entirely due to the GPUs different memory subsystem. The GPU uses memory banks rather than caches. This fundamental difference in the memory architecture means that these results are not unique to this problem, and superior GPU performance (of roughly an order of magnitude) is to be expected from a great number of scientific computing problems (most of which are memory bound like the SmithWaterman algorithm).

9.2 Computational Fluid Dynamics (CFD)

Direct numerical simulations of turbulence are optimized for up to 192 graphics processors. The results from three large GPU clusters (Lincoln, Forge and Keeneland) are compared to the performance of fairly new CPUs. A number of important algorithm changes are necessary to access the full computational power of graphics processors and these adaptations are discussed. In particular, the aspects of CFD research can be summarized as follows;

- The GPU has a fairly narrow operating range in terms of the number of unknowns per subdomain that the GPU should process. With less than 1M mesh points CFD calculations do not have enough internal work to hide communication times. And with more than 4M mesh points, standard GPUs run out of memory Tesla GPUs can go another 4× larger (up to 16M mesh points) (Ali Khajeh-Saeed and J. Blair Perot [116]).

- Detailed timings are preformed and the best approaches are highlighted in order to get more efficiency from different GPU memories (Ali Khajeh-Saeed and J. Blair Perot [117]).

- An algorithm was developed to overlap the bottleneck of copying data between the GPU and CPU in order to optimize MPI communication for large DNS simulation (Ali Khajeh-Saeed and J. Blair Perot [118]).

- Perfect scale up was presented with the GPU and the bottleneck for the CPU scale up are highlighted for large memory bound scientific codes (Ali Khajeh-Saeed and J. Blair Perot [119]).

9.3 GPU as High Performance Computational Resource

Different benchmarks were implemented on single and many GPUs. Advantages and disadvantages of GPUs as high performance computing hardware were explained for different scientific benchmarks. In particular, the aspects of many-core GPU research can be summarized as follows;

- The GPU is well-suited for many memory bound problems (Ali Khajeh-Saeed and J. Blair Perot [120]).

- Power/performance of new GPUs is low compared to traditional multi-core CPUs (Ali Khajeh-Saeed, Stephen Poole and J. Blair Perot [121]).

- Different GPU memory types were used in order to find the best suitable memory for various applications.
- Different algorithms were explored to circumvent the low speed PCI-e bandwidth between the GPU and CPU.
- The most important features were determined for building large GPU based supercomputers.
- Two different optimization approaches (using CUDA Visual Profiler and Parallel Nsight) were explored to fully optimize the application.

9.4 Publication List

Publications directly resulting form this work are:

1. Ali Khajeh-Saeed, Stephen Poole, and J. Blair Perot. Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229:42474258, 2010.
2. Ali Khajeh-Saeed and J. Blair Perot. GPU-supercomputer acceleration of pattern matching. In Wen-Mei W. Hwu, editor, *GPU Computing Gems*, chapter 13, pages 185198. Morgan Kaufmann, emerald edition, 2011.
3. Ali Khajeh-Saeed and J. Blair Perot. Computational fluid dynamics simulations using many graphics processors. *Submitted to the Computing in Science and Engineering*, August 2011.
4. Ali Khajeh-Saeed and J. Blair Perot. Direct numerical simulation of turbulence using GPU accelerated supercomputers. *Submitted to the Journal of Computational Physics*, November 2011.

5. Ali Khajeh-Saeed and J. Blair Perot. High performance computing on the 64-core Tilera processor. Submitted to *the Journal of Parallel and Distributed Computing*, November 2011.

6. Ali Khajeh-Saeed and J. Blair Perot. Turbulence simulation using many graphics processors. In *The* 64^{th} *Annual Meeting of the APS Division of Fluid Dynamics*, Baltimore, MD, November 20-22 2011.

7. Ali Khajeh-Saeed and J. Blair Perot. Efficient implementation of CFD algorithms on GPU accelerated supercomputers. Submitted to *the GPU Technology Conference (GTC)*, San Jose, California, May 14-17 2012. NVIDIA.

8. Ali Khajeh-Saeed, Stephen Poole and J. Blair Perot. A comparison of Multi-Core Processors on Scientific Computing Tasks. Submitted to *the Innovative Parallel Computing: Foundations and Applications of GPU, Manycore, and Heterogeneous Systems (InPar2012)*, San Jose, California, May 13-14 2012.

APPENDIX A

EQUIVALENCE OF THE ROW PARALLEL ALGORITHM

The row parallel algorithm has a mathematical form very close to the original Smith-Waterman algorithm. However, the equivalence of these two forms is not trivial. Using equation 4.1 and 4.5 one can write

$$H_n = \max(\widetilde{H}_n, E_n) \tag{A.1}$$

From the definition of E_n in equation 4.1 this becomes,

$$H_n = \max\left(\widetilde{H}_n, \max\begin{Bmatrix} H_{n-1}- & G_s- & G_e \\ H_{n-2}- & G_s- & 2G_e \\ \vdots & \vdots & \vdots \\ H_0- & G_s- & nG_e \end{Bmatrix}\right) \tag{A.2}$$

Expanding H_{n-1} in a similar fashion gives,

$$H_{n-1} = \max\left(\widetilde{H}_{n-1}, \max\begin{Bmatrix} H_{n-2}- & G_s- & G_e \\ H_{n-3}- & G_s- & 2G_e \\ \vdots & \vdots & \vdots \\ H_0- & G_s- & nG_e \end{Bmatrix}\right) \tag{A.3}$$

After placing equation A.3 into equation A.2 we have

$$H_n = \max\left(\widetilde{H}_n, \max\begin{Bmatrix} \left(\left(\widetilde{H}_{n-1} & , H_{n-3} & -G_s - 2G_e, \cdots, \right. \\ H_0 - & G_s - & \left.(n-1)G_e\right) - G_s - G_e\right) \\ H_{n-2} - & G_s - & 2G_e \\ \vdots & \vdots & \vdots \\ H_0 - & G_s - & nG_e \end{Bmatrix}\right) \tag{A.4}$$

Comparing the top row items with the column of items, it is clear that all the items in the top row (except the first one) are smaller than those in the column if $G_s \geq 0$. So we can write this as,

$$H_n = \max\left(\widetilde{H}_n, \max\begin{Bmatrix} \widetilde{H}_{n-1} - & G_s - & G_e \\ H_{n-2} - & G_s - & 2G_e \\ \vdots & \vdots & \vdots \\ H_0 - & G_s - & nG_e \end{Bmatrix}\right) \tag{A.5}$$

This process can be repeated. So the next item is,

$$H_{n-2} = \max\left(\widetilde{H}_{n-2}, \max\begin{Bmatrix} H_{n-3} - & G_s - & G_e \\ H_{n-4} - & G_s - & 2G_e \\ \vdots & \vdots & \vdots \\ H_0 - & G_s - & (n-2)G_e \end{Bmatrix}\right) \tag{A.6}$$

And inserting this into A.5 gives

$$H_n = \max\left(\widetilde{H}_n, \max\left\{\begin{array}{lll} \widetilde{H}_{n-1}- & G_s- & G_e \\ \Big(\big(\widetilde{H}_{n-2}, & H_{n-3}- & G_s - G_e, \cdots, \\ H_0- & G_s- & (n-2)G_e\big) - G_s - 2G_e\Big) \\ H_{n-3}- & G_s- & 3G_e \\ \vdots & \vdots & \vdots \\ H_0- & G_s- & nG_e \end{array}\right.\right) \tag{A.7}$$

Again the row items are smaller except the first, so

$$H_n = \max\left(\widetilde{H}_n, \max\left\{\begin{array}{lll} \widetilde{H}_{n-1}- & G_s- & G_e \\ \widetilde{H}_{n-2}- & G_s- & 2G_e \\ H_{n-3}- & G_s- & 3G_e \\ \vdots & \vdots & \vdots \\ H_0- & G_s- & nG_e \end{array}\right.\right) \tag{A.8}$$

After repeating this substitution for all the items, one obtains,

$$H_n = \max\left(\widetilde{H}_n, \max\left\{\begin{array}{lll} \widetilde{H}_{n-1}- & G_s- & G_e \\ \widetilde{H}_{n-2}- & G_s- & 2G_e \\ \vdots & \vdots & \vdots \\ \widetilde{H}_0- & G_s- & nG_e \end{array}\right.\right) \tag{A.9}$$

And with definition of $\widetilde{E}$ (equation 4.4), this shows that,

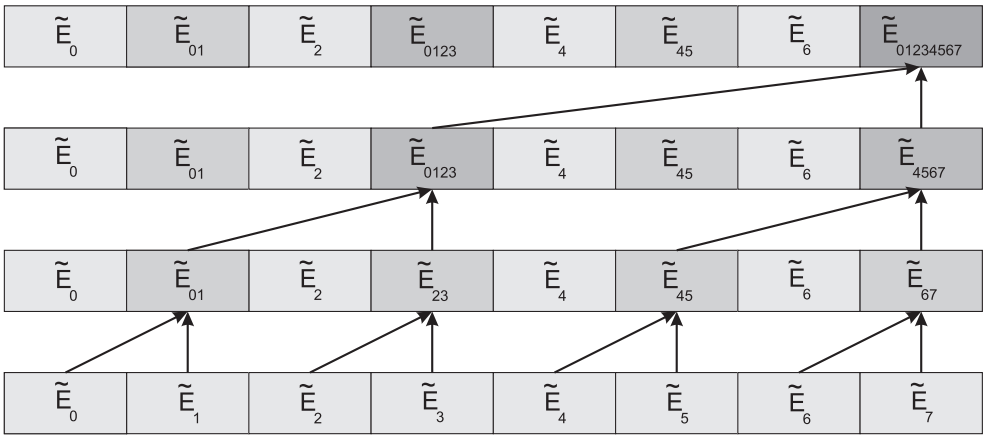
$$H_n = \max(\widetilde{H}_n, \widetilde{E}_n - G_s) \tag{A.10}$$

Which is also equation 4.5.

APPENDIX B

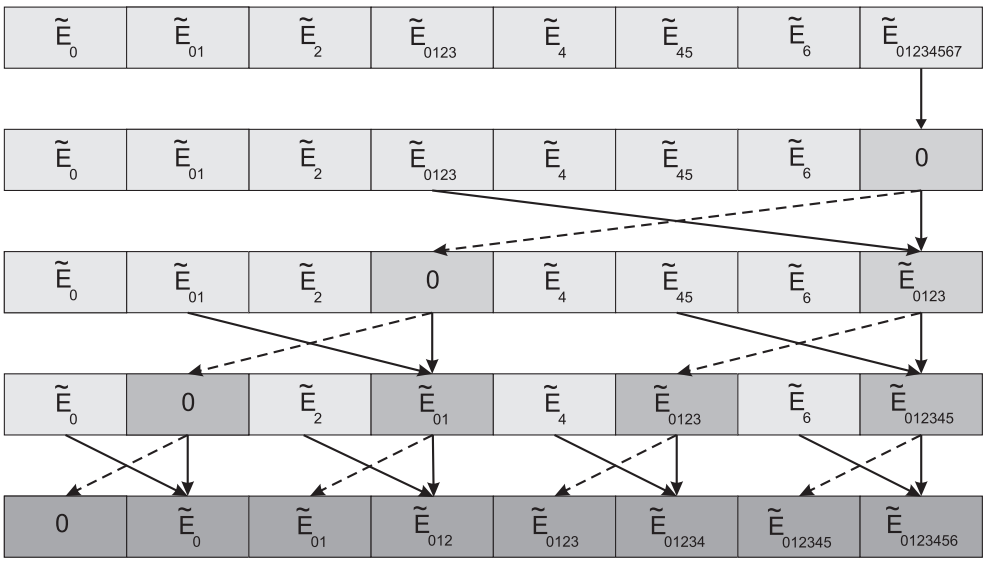
MODIFIED PARALLEL SCAN

The modified parallel maximum scan algorithm for calculating $\widetilde{E}$ in the row parallel Smith-Waterman Algorithm is described. The implementation basically uses the work-efficient formulation of Blelloch [122] and GPU implementation of Sengupta et al. [123]. This efficient scan needs two steps to scan the array, called up-sweep and down-sweep. The algorithms for these two steps is shown in figures B.1 and B.2, respectively. Each of these two steps requires $\log n$ parallel steps. Since the amount of work becomes half at each step. The overall work is $O(n)$.



$for\ \ d = 0\ \ to\ \ \log_2 n - 1$
$in\ \ parallel\ \ for\ \ k = 0\ \ to\ \ n - 1\ \ by\ \ 2^{d+1}$
$\widetilde{E}[k + 2^{d+1} - 1] = \max(\widetilde{E}[k + 2^d - 1], \widetilde{E}[k + 2^{d+1} - 1] + 2^d \times G_e)$

Figure B.1. Up-sweep for modified parallel scan for calculating the $\widetilde{E}$ for Smith-Waterman Algorithm



$$\widetilde{E}[n-1] = 0$$
$$for \;\; d = \log_2 n - 1 \;\; to \;\; 0$$
$$in \;\; parallel \;\; for \;\; k = 0 \;\; to \;\; n-1 \;\; by \;\; 2^{d+1}$$
$$Temp = \widetilde{E}[k + 2^d - 1]$$
$$\widetilde{E}[k + 2^d - 1] = \widetilde{E}[k + 2^{d+1} - 1]$$
$$\widetilde{E}[k + 2^{d+1} - 1] = \max(Temp, \widetilde{E}[k + 2^{d+1} - 1]) - 2^d \times G_e$$

Figure B.2. Down-sweep for modified parallel scan for calculating the $\widetilde{E}$ for Smith-Waterman Algorithm.

Each thread processes two elements and if the number of elements is more than the size of a single block, the array is divided into many blocks and the partial modified scan results are used as an input to the next level of recursive scan. The dark gray cells in figures B.1 and B.2 are the values of $\widetilde{E}$ in these cells that are updated in each step. The dash lines in figure B.2 means the data is copied to that cell.

APPENDIX C

CUDA SOURCE CODE FOR A 1-POINT STENCIL USED IN LAPLACE OPERATOR

```
__global__ void Interior_Lap_Kernel(real* mult, real* diag, real* pp, real* ww, real* acc)
{
      __shared__ real pp_Sh[BDIMY+2][BDIMX+2];
      real upper, lower, ww_Loc;

      int ix = blockIdx.x*blockDim.x + threadIdx.x;
      int iy = blockIdx.y*blockDim.y + threadIdx.y;
      int idx = iy*NX + ix;

      int i = threadIdx.x + 1;      // thread's x-index into corresponding shared memory tile
      int j = threadIdx.y + 1;      // thread's y-index into corresponding shared memory tile

      int loc = idx;
      // fill the "in-front" and "behind" data
      pp_Sh[j][i] = pp[idx];  //z=0
      upper = pp[idx+NXY];    //z=1
      real sum = 0.0;
      for(int iz=0; iz<NZ; iz++) {
            ////////////////////////////////////////
            // advance the slice (move the thread-front)
            idx  += NXY; //z=iz+1
            lower = pp_Sh[j][i];
            pp_Sh[j][i] = upper;
            upper = pp[idx+NXY];

            ////////////////////////////////////////
            // update the edges of the block
            if(threadIdx.y == 0) {// halo above/below
                  pp_Sh[0][i] = pp[idx-NX];
                  pp_Sh[BDIMY+1][i] = pp[idx+BDIMY*NX];  //wrong if size not= *16
            }
            if(threadIdx.x == 0) {// halo left/right  (all the same warp - so OK)
                  pp_Sh[j][0] = pp[idx-1];
                  pp_Sh[j][BDIMX+1] = pp[idx+BDIMX];   // also can be wrong
            }
            __syncthreads();   //all shared memory loaded and ready to go
            ////////////////////////////////////////
            // compute the output value
            ww_Loc = diag[idx]*   pp_Sh[j][i]  +
                     mult[idx]*( (pp_Sh[j][i+1]*DXIV[ix+1] + pp_Sh[j][i-1]*DXIV[ix])*DXIC[ix] +
                                 (pp_Sh[j+1][i]*DYIV[iy+1] + pp_Sh[j-1][i]*DYIV[iy])*DYIC[iy] +
                                 (upper        *DZIV[iz+1] + lower        *DZIV[iz])*DZIC[iz] );

            ww[idx] = ww_Loc;
            sum += ww_Loc*pp_Sh[j][i];
      }
      acc[loc] = sum;
}
```

BIBLIOGRAPHY

[1] NVIDIA. *NVIDIA Next Generation CUDA Compute Architecture: Fermi*, 2010.

[2] NVIDIA. *NVIDIA CUDA C Programming Guide*, 2010.

[3] `http://developer.nvidia.com/gpudirect`.

[4] NVIDIA. *Whitepaper, NVIDIA Next Generation CUDA Compute Architecture: Fermi*, 2010.

[5] `http://www.nvidia.com/object/tesla_computing_solutions.html`.

[6] `http://tiny.cc/MicrowayTeslaS2050`.

[7] `http://queue.acm.org/detail.cfm?id=1629155`.

[8] `http://gladiator.ncsa.illinois.edu/Images/forge/IMG_3006.jpg`.

[9] `http://blogs.nvidia.com/wp-content/uploads/2011/04/Keeneland.jpg`.

[10] TILERA Corporation. *TILEPro64 Processor Data Sheet*, 2010.

[11] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of* 2^{nd} *Workshop on General Purpose Processing on Graphics Processing Units*, 2009.

[12] S. M. de Bruyn Kops and J. J. Riley. Direct numerical simulation of laboratory experiments in isotropic turbulence. *Physics of Fluids*, 10(9):2125–2127, 1998.

[13] D. B. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan-Kaufmann Publishers, 2009.

[14] NVIDIA. *Tuning CUDA Applications for Fermi*, 2010.

[15] `http://en.wikipedia.org/wiki/PCI_Express#PCI_Express_3.0`.

[16] `http://www.nvidia.com/object/fermi_architecture.html`.

[17] `http://www.nvidia.com/object/cuda_home_new.html`.

[18] T. J. Chung. *Computational Fluid Dynamics*. Cambridge University Press, 2002.

[19] `http://en.wikipedia.org/wiki/Supercomputer`.

[20] http://www.top500.org/list/2010/11/100.

[21] http://www.nccs.gov/jaguar.

[22] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Tureka. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing*, 33:685 – 699, 2007.

[23] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. Hwu. Qp: A heterogeneous multi-accelerator cluster. In *In proceeeding of 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[24] Intel 64 tesla linux cluster lincoln webpage. http://tiny.cc/Lincoln_NCSA, 2008.

[25] Accelerator cluster webpage. http://iacat.illinois.edu/resources/cluster/, 2009.

[26] J. Enos V. Kindratenko, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. Hwu. Gpu clusters for high-performance computing. In *In proceeeding of Workshop on Parallel Programming on Accelerator Clusters, IEEE Cluster*, 2009.

[27] http://en.wikipedia.org/wiki/Tianhe-I.

[28] http://www.top500.org/lists/2010/06/press-release.

[29] http://tiny.cc/Froge_NCSA.

[30] http://keeneland.gatech.edu/node/7.

[31] NVIDIA. *CUDA C Best Practices Guide*, 2010.

[32] NVIDIA. *NVIDIA CUDA Reference Manual*, 2010.

[33] http://www.nvidia.com/object/product_geforce_gtx_480_us.html.

[34] http://www.nvidia.com/object/preconfigured-clusters.html.

[35] http://www.nvidia.com/object/product_geforce_gtx_295_us.html.

[36] http://www.nvidia.com/object/product_tesla_s1070_us.html.

[37] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *IEEE International Parallel and Distributed Processing Symposium*, 2007.

[38] Y. Liu, B. Schmidt, and D. L. Maskell. Cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. *BMC Research Notes*, 3:73 – 85, 2010.

[39] L. Ligowski and W. Rudnicki. An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databasesm. In *Proceeding of the* 23^{th} *IEEE International Parallel and Distributed Processing Symposium, Aurelia Convention Centre and Expo Rome, Italy*, 2009.

[40] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Jounal of Molecular Biology*, 147:195 – 197, 1981.

[41] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipmanl. Basic local alignment search tool. *Jounal of Molecular Biology*, 215:403 – 410, 1990.

[42] E. G. Shpaer, M. Robinson, D. Yee, J. D. Candlin, R. Mines, and T. Hunkapiller. Sensitivity and selectivity in protein similarity searches: a comparison of smith-waterman in hardware to blast and fasta. *Genomics,*, 38(2):179 – 191, 1996.

[43] `http://www.clcbio.com/index.php?id=1046`.

[44] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Jounal of Molecular Biology*, 48(3):443 – 453, 1970.

[45] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity search. *Science*, 227:1435 – 1441, 1985.

[46] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Science of the United States of America*, 85:2444 – 2448, 1988.

[47] O. Gotoh. An improved algorithm for matching biological sequences. *Jounal of Molecular Biology*, 162:705 – 708, 1982.

[48] O. O. Storaasli and D. Strenski. Accelerating science applications up to 100× with fpgas. In *Proc. of* 9^{th} *International Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway*, 2008.

[49] A. Khajeh-Saeed, S. Poole, and J. B. Perot. Acceleration of the smith-waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229:4247 – 4258, 2010.

[50] `http://www.ecs.umass.edu/mie/tcfd/Programs.htm`.

[51] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145 – 150, 1997.

[52] T. Rognes and E. Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699 – 706, 2000.

[53] M. Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156 – 161, 2007.

[54] A. Wirawan, C. K. Kwoh, N. T. Hieu, and B. Schmidt. Cbesw: Sequence alignment on the playstation 3. *BMC Bioinformatics*, 9:377 – 387, 2008.

[55] B. Alpern, L. Carter, and K.S. Gatlin. Microparallelism and high-performance protein matching. In *ACM/IEEE Supercomputing Conference, San Diego, California*, 1995.

[56] W. R. Rudnicki, A. Jankowski, A. Modzelewski, A. Piotrowski, and A. Zadrozny. The new simd implementation of the smith-waterman algorithm on cell microprocessor. *Fundamenta Informaticae*, 96(1):181 – 194, 2009.

[57] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on gpu. In *Proceeding of the 20th IEEE International Parallel and Distributed Processing Symposium, Rhodes Island, Greece*, 2006.

[58] Y. Liu, W. Huang, J. Johnson, and Sh. Vaidya. Gpu accelerated smith-waterman. In *International Conference on Computational Science, University of Reading, UK*, 2006.

[59] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming algorithms for biological sequence alignment on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 18(9):1270 – 1281, 2007.

[60] S. A. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerator for smith-waterman sequence alignment. *BMC Bioinformatics*, 9:10 – 19, 2008.

[61] Y. Liu, D. L. Maskell, and B. Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2:73 – 83, 2009.

[62] G. M. Striemer and A. Akoglu. Sequence alignment with gpu: Performance and design challenges. In *Proceeding of the* 23^{th} *IEEE International Parallel and Distributed Processing Symposium, Aurelia Convention Centre and Expo Rome, Italy*, 2009.

[63] A. Akoglu and G. M. Striemer. Scalable and highly parallel implementation of smith-waterman on graphics processing unit using cuda. *Cluster Computing*, 12:341 – 352, 2009.

[64] `http://www.highproductivity.org/SSCABmks.htm`.

[65] TILERA Corporation. *Multicore Development Environment System Programmers Guide*, June 2010.

[66] D. G. Waddington, C. Tian, and KC Sivaramakrishnan. Scalable lightweight task management for mimd processor. In *Systems for Future Multicore Architectures , EuroSys workshop*, pages 1–6, Salzburg, Austria, April 2011.

[67] D. Abts, N. D. E. Jerger, J. Kim, D. Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *the* 36^{th} *annual international symposium on Computer architecture, ISCA 09*, pages 451 – 461, New York, NY, USA, 2009.

[68] L.J. Karam, I. AlKamal, A. Gatherer, G.A. Frantz, D.V. Anderson, and B.L. Evans. Trends in multicore dsp platforms. *Signal Processing Magazine, IEEE*, 26(6):38–49, November 2009.

[69] I. Choi, M. Zhao, X. Yang, and D. Yeung. Experience with improving distributed shared cache performance on tilera's tile processor. *IEEE Computer Architecture Letters*, 10(2):45–48, July 2011.

[70] C. Hernndez, A. Roca, J. Flich, F. Silla, and J. Duato. Characterizing the impact of process variation on 45 nm noc-based cmps. *J. Parallel Distrib. Comput*, 71:651–663, 2011.

[71] C. Chen, J. B. Manzano, G. Gan, G. R. Gao, and Vivek Sarkar. A study of a software cache implementation of the openmp memory model for multicore and manycore architectures. In *the* 16^{th} *international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 341 – 352, Berlin, Heidelberg, 2010. Springer-Verlag.

[72] B. Bornstein, T. Estlin, B. Clement, and P. Springer. Using a multicore processor for rover autonomous science. In *Aerospace Conference, 2011 IEEE*, pages 1–9, March 2011.

[73] J. Ha and S. P. Crago. Opportunities for concurrent dynamic analysis with explicit inter-core communication. In *the* 9^{th} *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE 10*, pages 17 – 20, 2010.

[74] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[75] C. Yan, F. Dai, and Y. Zhang. Parallel deblocking filter for h.264/avc on the tilera many-core systems. *In Advances in Multimedia Modeling*, 6523:51–61, 2011.

[76] J. Richardson, C. Massie, H. Lam, K. Gosrani, and A. George. Space applications on tilera. In *Workshop for Multicore Processors For Space - Opportunities and hallenges, IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 19–23, Pasadena, CA, July 2009.

[77] C. Ulmer, M. Gokhale, B. Gallagher, P. Top, and T. Eliassi-Rad. Massively parallel acceleration of a document-similarity classifier to detect web attacks. *J. Parallel Distrib. Comput*, 71:225–235, 2011.

[78] `http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/`.

[79] A. Khajeh-Saeed and J. Blair Perot. Gpu-supercomputer acceleration of pattern matching. In Wen-Mei W. Hwu, editor, *GPU Computing Gems*, chapter 13, pages 185–198. Morgan Kaufmann, emerald edition, 2011.

[80] Michael B Martell JR. Simulation of turbulence over superhydrophobic surface. Master's thesis, University of Massachusetts, Amherst, February 2009.

[81] J. Gadebusch and J. B. Perot. Self-adapting turbulence model for hybrid rans/les. In *Meeting of the Canadian CFD Society*, Toronto, CA, June 2007.

[82] J. B. Perot and J. Gadebusch. A self-adapting turbulence model for flow simulation at any mesh resolution. *Physics of Fluids*, 19:115105–115116, 2007.

[83] M. B. Martell, J. B. Perot, and J. Rothstein. Direct numerical simulations of turbulent flows over superhydrophobic surfaces. *Journal of Fluid Mechanics*, 620:31 – 41, 2009.

[84] M. B. Martell, J. P. Rothstein, and J. B. Perot. An analysis of superhydrophobic turbulent drag reduction mechanisms using direct numerical simulation. *Physics of Fluids*, 22(6):1 – 13, 2010.

[85] J. Kim, P. Moin, and R. Moser. Turbulence statistics in fully developed channel flow at low reynolds number. *Journal of Fluid Mechanics*, 177:133 – 166, 1987.

[86] R. Moser, J. Kim, and N. Mansour. Direct numerical simulation of turbulent channel flow up to $re_\tau = 590$. *Physics of Fluids*, 11(4):943 – 945, 1998.

[87] T. R. Hagen, K. Lie, and J. R. Natvig. Solving the euler equations on graphics processing units. In *International Conference on Computational Science, University of Reading, UK*, 2006.

[88] M.J. Harris. Fast fluid dynamics simulation on the gpu. In *GPU Gems*, chapter 38, pages 637 – 665. Addison Wesley, 2004.

[89] E. Elsen, P. LeGresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a gpu. *Journal of Computational Physics*, 227:10148 – 10161, 2008.

[90] A. Corrigan, F. Camelli, R. Lohner, and J. Wallin. Running unstructured grid based cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics, San Antonio, Texas*, 2009.

[91] D. Rossinelli and P. Koumoutsakos. Vortex methods for incompressible flow simulations on the gpu. *Visual Computer*, 24:699 – 708, 2008.

[92] D. Rossinelli, M. Bergdorf, G. Cottet, and P. Koumoutsakos. Gpu accelerated simulations of bluff body flows using vortex particle methods. *Journal of Computational Physics*, 229:3316 – 3333, 2010.

[93] A. S. Antoniou, K. I. Karantasis, E. D. Polychronopoulos, and J. A. Ekaterinaris. Acceleration of a finite-difference weno scheme for large-scale simulations on many-core architectures. In *American Institute of Aeronautics and Astronautics Paper*, 2010.

[94] D. A. Jacobsen, J. C. Thibault, and I. Senocak. An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters. In *48th AIAA Aerospace Sciences, Orlando, Florida*, January 2010.

[95] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618 – 2640, 2007.

[96] J.S. Meredith, G. Alvarez, T.A. Maier, T.C. Schulthess, and J.S. Vetter. Accuracy and performance of graphics processors: a quantum monte carlo application case study. *Parallel Computing*, 35(3):151 – 163, 2009.

[97] D. Juba and A. Varshney. Parallel, stochastic measurement of molecular surface area. *Journal of Molecular Graphics and Modelling*, 27(1):82 – 87, 2008.

[98] N.A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227(18):8290 – 8313, 2008.

[99] W. Li, X.M. Wei, and A. Kaufman. Implementing lattice boltzmann computation on graphics hardware. *Visual Computer*, 19(7-8):444 – 456, 2003.

[100] J. Tolke and M. Krafczyk. Teraflop computing on a desktop pc with gpus for 3d cfd. *International Journal of Computational Fluid Dynamics*, 22(7):443 – 456, 2008.

[101] Z. Fan, F. Qiu, A. Kaufman, and Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing, IEEE Computer Society, Washington, DC, USA*, 2004.

[102] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the* 47^{th} *AIAA Aerospace Sciences Meeting*, 2009.

[103] T. Brandvik and G. Pullan. Acceleration of a 3d euler solver using commodity graphics hardware. In 46^{th} *AIAA Aerospace Sciences Meeting and Exhibit*, 2008.

[104] J. C. Thibault and I. Senocak. Cuda implementation of a navier-stokes solver on multi-gpu platforms for incompressible flows. In 47^{th} *AIAA Aerospace Science Meeting*, 2009.

[105] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using gpus to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering*, 4(1):36 – 55, 2008.

[106] `http://developer.nvidia.com/object/nsight.html`.

[107] J. B. Perot and J. Gadebusch. A stress transport equation model for simulating turbulence at any mesh resolution. *Theoretical and Computational Fluid Dynamics*, 23(4):271–286, 2009.

[108] J. B. Perot. An analysis of the fractional step method. *Journal of Computational Physics*, 108(1):183–199, 1993.

[109] W. Chang, F. Giraldo, and J. B. Perot. Analysis of an exact fractional step method. *Journal of Computational Physics*, 179:1–17, 2002.

[110] J. B. Perot and V. Subramanian. Discrete calculus methods for diffusion. *Journal of Computational Physics*, 224(1):59–81, 2007.

[111] J. B. Perot. Conservation properties of unstructured staggered mesh schemes. *Journal of Computational Physics*, 159(1):58–89, 2000.

[112] V. Subramanian and J. B. Perot. Higher-order mimetic methods for unstructured meshes. *Journal of Computational Physics*, 219(1):6–85, 2006.

[113] G. Comte-Bellot and S. Corrsin. The use of a contraction to improve the isotropy of grid-generated turbulence. *Journal of Fluid Mechanics*, 25:657–682, 1966.

[114] G. Comte-Bellot and S. Corrsin. Simple eulerian time correlation of full- and narrow-band velocity signals in grid-generated, isotropic turbulence. *Journal of Fluid Mechanics*, 48:273–337, 1971.

[115] J. Blair Perot. Determination of the decay exponent in mechanically stirred isotropic turbulence. *AIP Advanced*, 1(2):022104–022122, 2011.

[116] A. Khajeh-Saeed and J. Blair Perot. Computational fluid dynamics simulations using many graphics processors. *Submitted to the Computing in Science and Engineering*, August 2011.

[117] A. Khajeh-Saeed and J. Blair Perot. Direct numerical simulation of turbulence using gpu accelerated supercomputers. *Submitted to the Journal of Computational Physics*, November 2011.

[118] A. Khajeh-Saeed and J. Blair Perot. Turbulence simulation using many graphics processors. In *The* 64^{th} *Annual Meeting of the APS Division of Fluid Dynamics*, Baltimore, MD, November 20-22 2011.

[119] A. Khajeh-Saeed and J. Blair Perot. Efficient implementation of cfd algorithms on gpu accelerated supercomputers. In *Submitted to the GPU Technology Conference (GTC)*, San Jose, California, May 14-17 2012. NVIDIA.

[120] A. Khajeh-Saeed and J. Blair Perot. High performance computing on the 64-core tilera processor. *Submitted to the Journal of Parallel and Distributed Computing*, November 2011.

[121] A. Khajeh-Saeed and S. Poole J. Blair Perot. A comparison of multi-core processors on scientific computing tasks. In *Submitted to the Innovative Parallel Computing: Foundations and Applications of GPU, Manycore, and Heterogeneous Systems (InPar2012)*, San Jose, California, May 13-14 2012.

[122] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[123] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Graphics Hardware*, pages 97 – 106, San Diego, CA, August 2007.