Matrix-Matrix Multiplications on GPUs for Accelerating a Parallel Fluid Dynamics Code

by

Kenneth Webster

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Hans De Sterck

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this report. This is a true copy of the report, including any required final revisions, as accepted by my examiners.

I understand that my report may be made electronically available to the public.

Abstract

A few approaches are investigated of matrix-matrix multiplication on graphics processing units (GPUs). Aspects of memory management and GPU saturation are described and discussed. The focus of this paper is to off-load matrix-matrix multiplications to a GPU in an HPC setting for the purpose of accelerating a parallel fluid dynamics code.

Acknowledgements

I would like to thank all the little people who made this possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In many scientific computing areas, a solution to a problem is obtained through iterative updates to a set of values. The process of updating the set of values and the necessary operations to do so can sometimes be posed in such a way that the task of updating is done with the help of matrix-matrix multiplication. In an HPC setting, the task of performing updates can involve thousands of matrix-matrix multiplications. Several studies, [1, 3, 4, 10, 13, 14, 15], have claimed that using a graphics card to assist with vector arithmetic can speed up computations by a factor of 100 and more. In practice, the results of using graphics cards are quite varied, and there is no general answer to the question of whether or not using a graphics card will be beneficial.

The hardware that performs the computation inside a graphics card is called the Graphics Processing Unit (GPU). GPUs, like central processing units (CPUs), can be used for computation. The difference between a CPU and a GPU is that a CPU focuses on executing one thread very quickly over a small amount of data whereas a GPU focuses on executing many concurrent threads over a large amount of data. The GPU sacrifices speed in compensation for the ability to handle many threads running concurrently. The whole idea of the GPU is to flood it with many identical and simple operations on data in parallel. So when a program is known to fit this data parallel pattern, the GPU is well suited to the task.

For the parallel fluid dynamics code described in [6], the task of performing thousands of matrix-matrix multiplications at each iteration was pinpointed as a bottleneck in the code. One approach to reducing the bottleneck is to use graphics cards for the matrix-matrix multiplications. This paper investigates the various techniques needed to offload this computation to a graphics card while other independent computations are performed

on the CPU.

Chapter 2

Background

2.1 Motivation

The paper [6] describes a parallel framework for computing the solution of hyperbolic conservation laws in domains between two concentric spheres. The specific conservation laws that were considered are related to fluid dynamics. The equations were discretized using a Godunov-type finite-volume scheme. The solution is obtained iteratively and within each iteration, solution values are updated. These computations can be done in parallel. One component of the computations is a large amount of matrix-matrix multiplications.

Consider matrix-matrix multiplication $C = A^{-1} * B$, where C is an $m \times n$ matrix, A^{-1} is an $m \times z$ matrix, and B is a $z \times n$ matrix. A^{-1} is a pseudo-inverse of A as explained in [6]. The matrices result from the discretization of the PDEs and from the current solution values of physical aspects such as pressure density, etc. In [6], various discretization stencils were used. In 3D, the stencils considered are shown in Figure 2.1

The stencils in Figure 2.1 determine the sizes of the matrices in the matrix-matrix products, specifically the number of cells in the stencil is the z from $C_{m \times n} = A^{-1}_{m \times z} * B_{z \times n}$. The size of m is the number of polynomial coefficients for the required order of accuracy of the reconstruction. The size of n is the number of unknowns in the system of equations being used: Euler has 5 in 3D, magnetohydrodynamics (MHD) has 8 in 3D. Our standard case has a reconstruction order of 3 so there will be 19 coefficients and 9 variables.

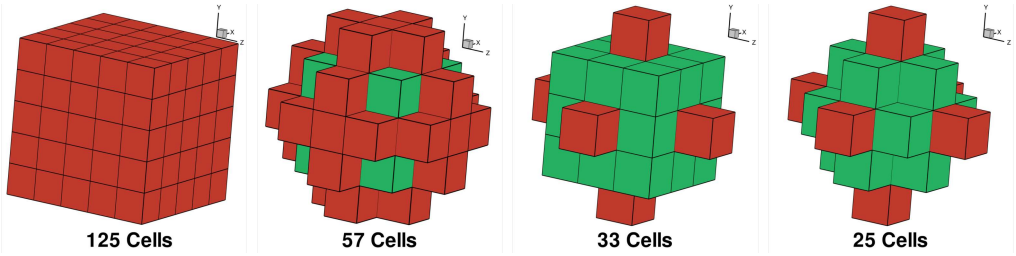

Figure 2.1: Discretization stencils considered in the three-dimensional fluid dynamics code from [6]

Cells in Stencil	size of A	size of B	size of C
125	19 x 124	124 x 9	19 x 9
57	19 x 56	56 x 9	19 x 9
33	19 x 32	32 x 9	19 x 9
25	19 x 24	24 x 9	19 x 9

Table 2.1: Matrix sizes for the various stencils using our standard case of 19 coefficients and 9 unknowns for the fluid dynamics equations

From this point onwards the matrix-matrix multiplication equation $C = A^{-1} * B$ will be defined as $C = A * B$ to make the notation easier.

2.2 SHARCNET

SHARCNET stands for Shared Hierarchical Academic Research Computing Network. It is a consortium of Canadian academic institutions who share a network of high performance computers. "HPC is the use of high-end computing resources (computers, storage, networking and visualization) to help solve highly complex problems, perform business critical analyses, or to run computationally intensive workloads that are, in scale, far beyond the tasks that could be achieved on today's leading desktop systems". [11]

2.3 Monk Cluster

Monk is the newest graphics card cluster in SHARCNET. It is the system that was used to run tests for all results in this paper. Unless otherwise specified, all tests were performed on 15 nodes, using 1 graphics card per node, with the timed calls being repeated in code 10 times. This means that 150 samples were taken for each test and the average times were reported. Table 2.2 gives some information on the Monk cluster, and Table 2.3 gives some specifics of the graphics card itself.

Description	Value
Number of Cores	432
Number of Nodes	54
Interconnect	QDR InfiniBand
Cores per Node	8
CPUs per Node	2
Graphics Cards per Node	2
Memory per Node	48 GB
CPU	Intel E5607 4 cores @ 2.26 GHz

Table 2.2: SHARCNET Monk cluster configuration

2.4 CUDA

In this and the following sections we describe the Compute Unified Device Architecture (CUDA), typical GPU layout, typical code layout, and GPU memory types, summarizing the detailed information available in, for example, [9]. CUDA was developed by Nvidia for the purpose of being a parallel computing architecture for graphics processing. Nvidia GPUs all use CUDA as the computing engine. CUDA is accessible through the programming language C, via an extension. Through the use of CUDA, GPUs can be used for computation as CPUs are. But the architecture of the GPU emphasizes running many threads concurrently whereas the CPU emphasizes running one thread very quickly. Since the design of the CPU concentrates on running one thread it has also been designed to be

Description	Value
CUDA Driver Version / Runtime Version	4.1 / 4.1
CUDA Capability Major/Minor version number:	2.0
Total amount of global memory:	5375 MBytes (5636554752 bytes)
(14) Multiprocessors x (32) CUDA Cores/MP:	448 CUDA Cores
GPU Clock Speed:	1.15 GHz
Memory Clock rate:	1566.00 Mhz
Memory Bus Width:	384-bit
L2 Cache Size:	786432 bytes
Max Texture Dimension Size (x,y,z)	1D=(65536) 2D=(65536,65535) 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers	1D=(16384) x 2048 2D=(16384,16384) x 2048
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	32768
Warp size:	32
Maximum number of threads per block:	1024
Maximum sizes of each dimension of a block:	1024 x 1024 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 65535
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and execution:	Yes with 2 copy engine(s)
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Concurrent kernel execution:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support enabled:	Yes
Device is using TCC driver mode:	No
Device supports Unified Addressing (UVA):	Yes

Table 2.3: Properties of Graphics Card Nvidia Tesla M2070 with Fermi architecture, 448 cores, 1.15 GHz, 6 GB memory, 144 GB/s memory bandwidth and compute capability 2.0

more versatile than the GPU, but when handling many threads its performance suffers.

The GPU processors are far less versatile than the CPU processors and are slower when performing operations that do not fit into the data-parallel mold. When general-purpose problems are approached this way using GPUs, it is known as GPGPU computing. [5]

The CUDA programming framework was selected to be used for our problem instead of the more general OpenCL standard for a few reasons. It exposes double precision accuracy on the graphics cards in the Monk cluster. It allows individual threads to read from arbitrary addresses in the memory. It exposes on-chip memory, which is several times faster than the global memory. It boasts full support for integer and bitwise operations, and is also the proprietary computing engine for the Nvidia graphics cards in Monk.

2.5 GPU layout

The layout of the GPU greatly affects the performance of an algorithm running on it. Knowledge of a GPU layout helps to understand why some algorithms perform better than others. The GPU is separate from the main computer entirely. It has its own memory, processors, and runs at a different clock speed than the CPU. A GPU is a collection of Streaming Multiprocessors (SMs), memory and a few other valuable units. Figure 2.2 shows the layout of the GPU and how the various components are connected.

Each SM is a multicore processor with several cores. The SM differs from a CPU in that the SM usually has more cores than a CPU but the cores cannot operate independently. Each core in an SM must execute the same set of instructions but it can do this on different data.

Like a CPU an SM has on-chip memory. Registers are a very limited form of memory that can only be used for individually named variables with only one value, and they are only accessible by the processor that declared them. Also on-chip is an L1 cache which acts exactly like the L1 cache on a CPU. Another type of on-chip memory is the shared memory. Within the set of instructions running on the SM, memory can be allocated on-chip using the shared memory. For data that is generated by and/or shared among the cores of an SM, it is better to use shared memory instead of global memory because it reduces the waiting time of read/write operations.

The shared memory in a GPU is very small and is unsuitable for long-term storage of

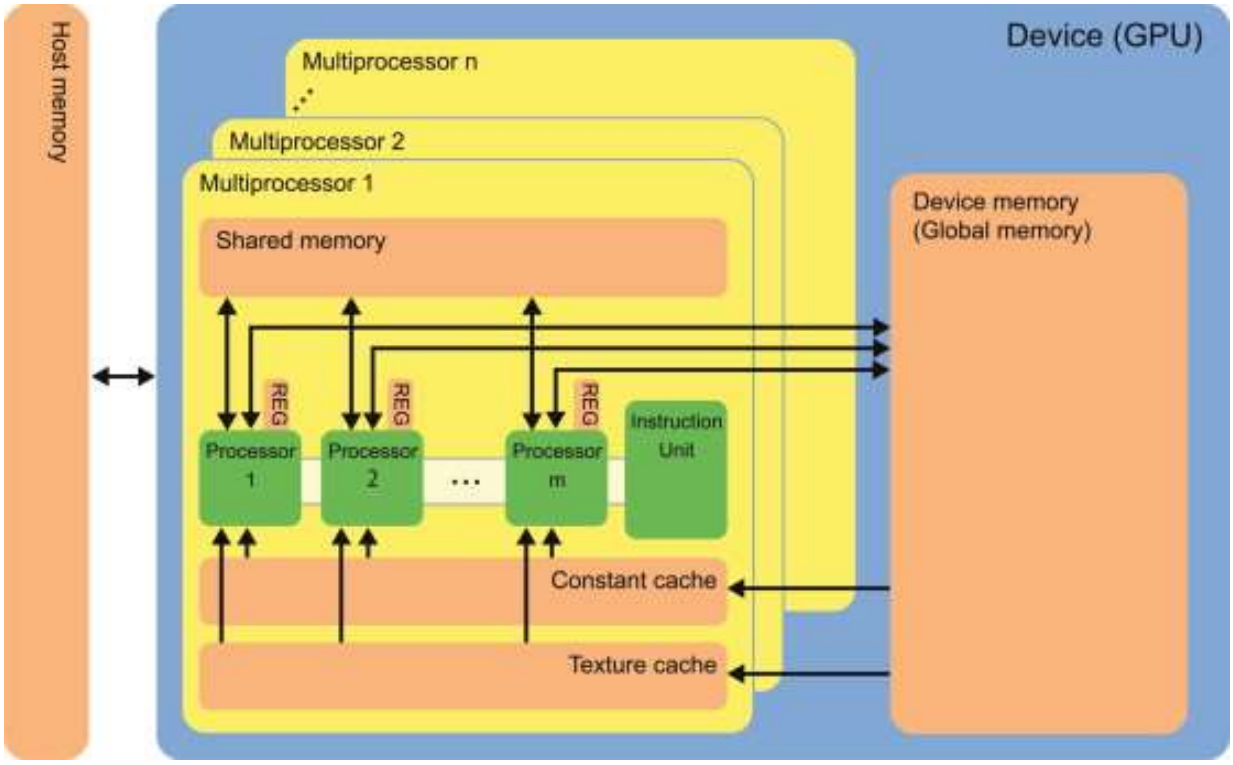

Figure 2.2: An illustration of a graphics card setup [17]

data. Since shared memory is located on-chip, all the cores in the SM can access it, but cores from other SMs cannot access it. There is also the global memory on the device, which is off-chip but is much larger. This space can be accessed by all processors in the GPU. These two aspects make the global memory suitable for long-term storage of data. A negative aspect is that it has much slower access times than on-chip memory, usually about 100 times slower. However, if the code is programmed well some threads will run while other threads wait for memory retrieval.

A GPU also has a copy engine. The copy engine is a dedicated processor that only handles data transfers between the host CPU and the GPU. This unit functions entirely separately from SMs and as such, does not affect computation. This recent developement allows code to be running while data is being transferred from the host to the device. More recently, GPUs are beginning to have multiple copy engines. The additional engines allow a streaming approach to the transfer of data, which is helpful for video processing and continuous transmission.

2.6 Code Layout

Programming for the GPU is quite different from programming for the CPU. The method that executes on the GPU is called the Kernel. Every processor on the GPU will execute this exact same kernel if no other concurrent kernels are running. If other kernels are running, then all the processors in an SM will execute the same kernel although different SMs can execute different kernels. Data has to be mapped out in a certain way so that the GPU can run the kernel. There are two levels to this mapping. The first level is called a grid. A grid is a multidimensional array where each element corresponds to an SM. These elements are the second level and are called blocks. A block can also be multidimensional. Each entry in the block will be a thread that runs on the SM. When a thread requests data from any type of memory the core running that thread will begin another thread. Doing this keeps the cores busy while data is being retrieved. Eventually an entire block finishes and another block is assigned to the SM. Depending on the memory usage of each block, the GPU may assign multiple blocks to run simultaneously on the same SM. Grids and blocks have hardware-defined maximum sizes and a maximum number of dimensions, usually 3. The reason for this two level setup is that it allows multiple kernels to run simultaneously if a kernel is not using all the SMs. It forces the program to be set up in such a way that each block is independent of other blocks.

Figure 2.3 shows what types of memory each part of the two-level system can access. Blocks have their own shared memory which each thread can access. Each thread has its own set of registers, stored on chip, and its own local memory, stored in the global memory. Everything in the grid can access the global memory. Each thread that is executed is given a set of individual parameters, which describe which block it is in and where in each block it is. This is what allows the kernel to actually process different values for different threads.

2.7 Memory Types of a GPU

There are several types of memory that a GPU interacts with throughout a program's lifetime and they all have various speeds and sizes. The amount of memory that is available on the host CPU system is limited by the computer itself. It is the slowest memory to access, because it has to be transferred from the CPU host random access memory (RAM) to the GPU device RAM over a peripheral component interconnect express (PCIe) slot.

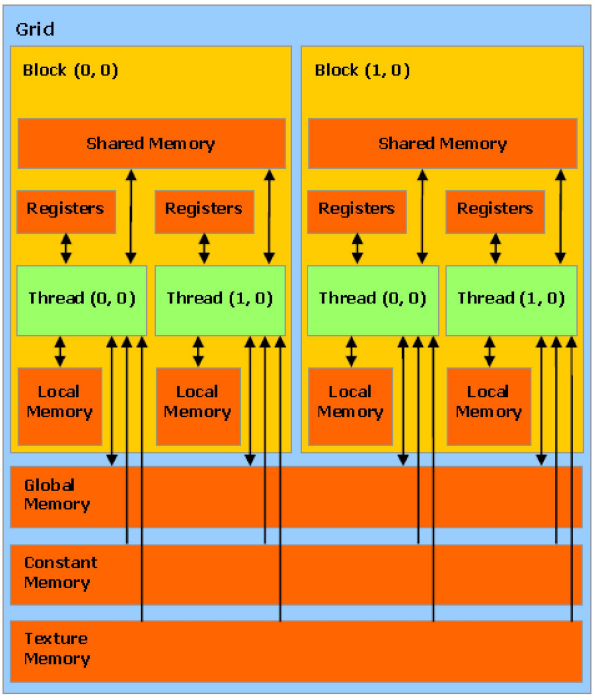

Figure 2.3: Distribution of computation within the code from [16]. Not shown is the copy engine. The copy engine is not accessible by any part of the kernel. Only the host can access it with a separate call by the CPU code.

In addition to being slow, only the CPU can initiate memory transfer. The benefit of the CPU host memory is that its size is usually several times greater than the size of RAM available on the GPU. Global memory is the RAM in the GPU device itself. It is much faster than CPU host memory and the GPU can initiate memory operations. There is also shared memory on the GPU which is again many magnitudes smaller than the global GPU memory, so it has to be managed by the programmer. Table 2.4 illustrates an approximation of the various speeds associated with each memory type.

Name	Location	Size	Cycles	Speed
Host	CPU host	48 GB	N/A	8 GB/s
Global	DRAM on GPU	6 GB	400-800	144 GB/s
Shared (per SM)	on-chip on GPU	48/16 kB	8-20 cycles	2.81-14.1 TB/s
L1 cache (per SM)	on-chip on GPU	16/48 kB	8-20 cycles	2.81-14.1 TB/s

Table 2.4: This shows that if the problem is not approached carefully then it could become IO bound, so we want to reduce all Host memory transactions as much as possible, followed by fewer global memory accesses. This type of memory awareness is essential to build fast code in a GPU setting.

Chapter 3

Matrix-Matrix Multiplication on GPUs

3.1 Approaches To Matrix-Matrix Multiplication on a GPU

We begin by describing several approaches for computing the matrix-matrix multiplication of two matrices $C = A * B$, where C is an $m \times n$ matrix, A is an $m \times z$ matrix and B is a $z \times n$ matrix.The three methods are described in [9] and [7]. The final method is contained in the Nvidia CUDA Basic Linear Algebra Subroutines (CUBLAS) library. According to the manufacturer in [8], "It is a GPU-accelerated version of the complete standard Basic Linear Algebra Subroutines (BLAS) library that delivers 6x to 17x faster performance than the latest Math Kernel Library (MKL) BLAS".

3.1.1 Method 1

The first method of matrix-matrix multiplication on the GPU is a standard algorithm of complexity $O(n^3)$. On the GPU, each block of the grid is 1 x 1 and the grid is defined to be the size of the C matrix. This approach limits the matrix size to 65535 x 65535. The matrices that are required for the fluid dynamics application are all smaller than 128 x 128.

The idea is that an SM computes a single entry of the resulting C matrix (The shaded region of C in Figure 3.1) by performing a dot product of the corresponding row in A

(The shaded region of A in Figure 3.1) and the column in B (The shaded region of B in Figure 3.1), reading each entry from A and B from the global memory one entry at a time and accumulating the products to get the entry of C. This approach is very slow and not recommended due to the extremely high number of global memory accesses required. However, it should be noted that recently GPUs are beginning to have caches for the global memory. This would speed up the calculation, but not enough for this method to be efficient. Since each SM is only computing a single element of C, the other cores are not being used. So the algorithm does not saturate the GPU at all.

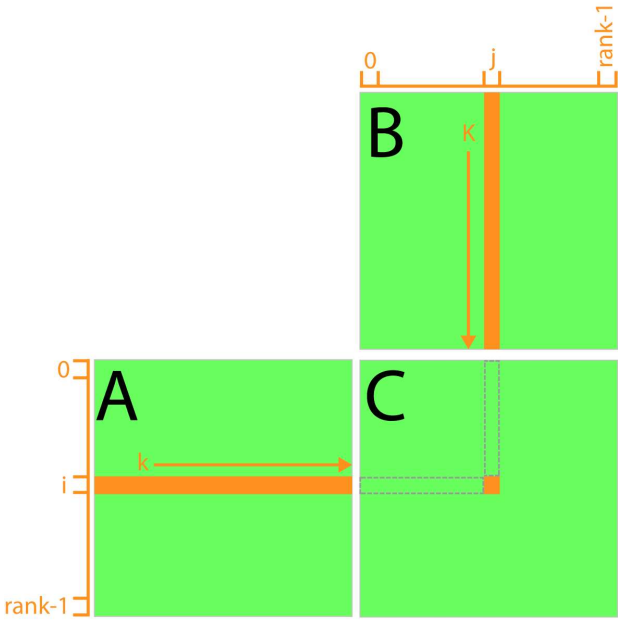

Figure 3.1: A standard approach for matrix-matrix multiplication [9]

3.1.2 Method 2

This method has each multiprocessor simultaneously compute a tile of entries of C (The shaded region of C in Figure 3.2). This saturates the multiprocessor so that all cores are being used at least some of the time. The multiprocessor still has to read entries of A and B from global memory. This method is better than Method 1.

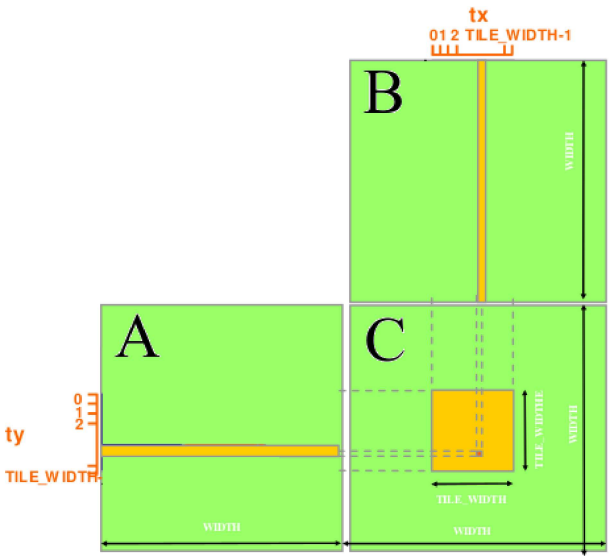

Figure 3.2: Using tiles to saturate the GPU [7]

3.1.3 Method 3

Here the three matrices are divided into smaller submatrices which are called tiles, and these tiles are multiplied together and accumulated in the manner of block-matrix multiplication. In block-matrix multiplication, the same number of operations are performed as in ordinary matrix-matrix multiplication. Partioning the matrices into tiles allows the individual tiles of A, B, and C to be stored in the shared memory, so that the cubic operation of matrix-matrix multiplication that is performed on the tiles is calling data from the shared memory instead of the global memory. This change in location results in a much faster method.

$$C_{m \times n} = A_{m \times z} * B_{z \times n}$$

If the tiles have dimensions 1 x 1 then this method becomes the first method. The size of the tile decides how many global memory accesses there will be, so having a tile the size of the matrices would be optimal because everything would be in the much faster shared

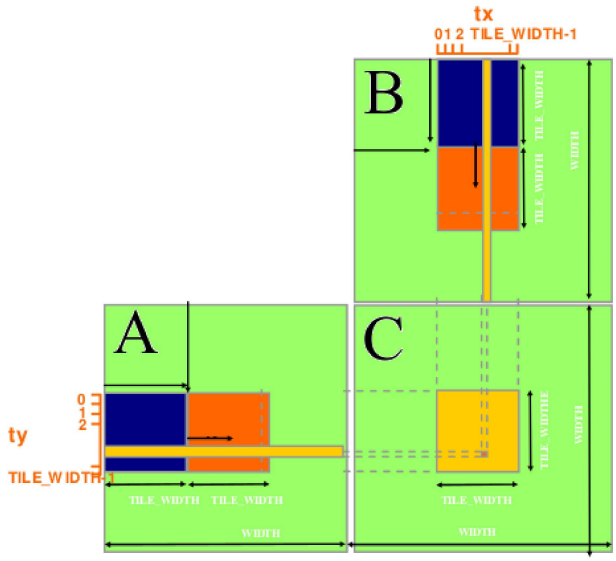

Figure 3.3: Using shared memory to reduce global memory accesses. [7]

memory of the device. This is not done because the shared memory is normally too small for a whole matrix to be copied to it. But choosing the largest tile size possible is the best option.

Recall that our matrix-matrix multiplication is $C = A * B$ where C is an $m \times n$ matrix, A is an $m \times z$ matrix, and B is a $z \times n$ matrix. So if m, z, and n are divided up into p, q, and r tiles respectively, where the tilesize is t then the number of times that the global memory is accessed by Method 3 is $O(pqrt^2)$ instead of $O(mzn)$, like Methods 1 and 2. In all three methods the number of times that C is accessed is $O(mn)$ so it is not shown in the order of accesses. Since the tilesize divides m, z, and n the number of accesses can be written in other ways. Here is one such way that also helps us see the reduction in the number of accesses. $pqrt^2 = pqrtt = pqr\left(\frac{z}{q}\right)\left(\frac{n}{r}\right) = pzn = \left(\frac{m}{t}\right)zn = \frac{1}{t}mzn$. So Method 3 has the same order of accesses but it is less than the other methods by a factor that is the tilesize t. Table 3.1 shows explicitly the sizes, where NTT represents the number of times a tile from the matrix is accessed from the global memory.

In Methods 1 and 2, computing a single entry of the C matrix requires $O(z)$ accesses to the global memory. Since C has mn entrys the total number of global accesses is $O(mzn)$

Matrix	Number of Entrys	Number of Tiles	NTT
A	$m \times z$	$p \times q$	r
B	$z \times n$	$q \times r$	p
C	$m \times n$	$p \times r$	1

Table 3.1: Tile statistics of the matrices

for all three methods. Table 3.1 shows that when computing all the tiles of the C matrix, there are pq tiles in the matrix A with each one being accessed r times, and that there are qr tiles in the matrix B with each one being accessed p times. There are pr tiles in C with each one only being accessed once. So the whole calculation requires $O(pq*r+qr*p+pr)$ accesses to tiles from memory which is $O(pqr)$, then with each tile having a size of $t \times t$ the global accesses become $O(pqrt^2)$ As the tile sizes are fixed for all matrix sizes, the number of memory accesses is cubic for both methods, but this method has a large constant factor difference in the number of global accesses. This factor is $\frac{1}{t}$. For instance, if the tiles are all square and contain 32 by 32 entries then the ratio of global memory accesses between this method and the first method is $\frac{1}{32}$, which is a significant difference. In the end, the same cubic number of matrix element accesses are occuring except that they are only going to the shared memory which is much faster than the global memory.

Tables 3.2 and 3.3 outline the various global accesses of a toy example, where A, B, C are square matrices of size $N \times N$. We count the number of times that the matrix A is accessed in global memory in the third column. The far right column shows the ratio between the third column and the first entry in the third column.

Choosing Tilesize

Selecting an unfavourable tilesize can result in large increases in computational time. So it is important to select a tilesize that is favourable for most situations or for the situations that apply to the specific problem. If there are too many global memory accesses in the code then the problem shifts to being IO bound, which is not desirable. So choosing the largest tilesize possible may be the best solution to the problem. However, there is a limit on the tilesize that is determined by the hardware itself. In the case of the Monk cluster, the multiprocessors on the GPU have a shared memory size of 49152 bytes, but there are also additional considerations. Shared memory is sometimes set to be used by

N	Tile size	Accesses	Ratio
256	256	65536	1
256	128	131072	2
256	64	262144	4
256	32	524288	8
256	16	1048576	16
256	8	2097152	32
256	4	4194304	64
256	2	8388608	128
256	1	16777216	256

Table 3.2: Toy Example

N	Tile size	Accesses	increase
6561	6561	43046721	1
6561	2187	129140163	3
6561	729	387420489	9
6561	243	1162261467	27
6561	81	3486784401	81
6561	27	10460353203	243
6561	9	31381059609	729
6561	3	94143178827	2187
6561	1	282429536481	6561

Table 3.3: Toy Example

the compiler so the whole shared memory space is not always available. Anytime a shared array is accessed with a variable position identifier, it uses shared memory for each thread to remember the pointer. This effectively doubles the shared memory required by each multiprocessor to access arrays. The more important aspect taken into consideration when selecting a tilesize is the banksize of the hardware. If the tilesize is not chosen to be the banksize then the threads will try to access data which is stored in the same bank causing multiple threads to be serialized in accessing the data. The tilesize should be some multiple of the banksize and then should be offset by 1 so that bank accesses never overlap ([2] page 24). On Monk the banksize is 32, so any shared arrays should be of sizes [32x][32y]...[32z + 1].

Table 3.4 illustrates the kernel's access pattern in the memorybanks when the banksize

is 4 and the SM has 4 cores. The left part of the table shows an undesirable tilesize while the one on the right is a preferable size. The red box indicates which entries of the tile the first 4 threads access, 1 thread being executed by 1 of the 4 cores in the SM. Each position in the tile has the number of the memorybank where the value of the tile at that position is stored.

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

1	2	3	4	1
2	3	4	1	2
3	4	1	2	3
4	1	2	3	4

Table 3.4: Tiles with the memorybank that each entry is stored in

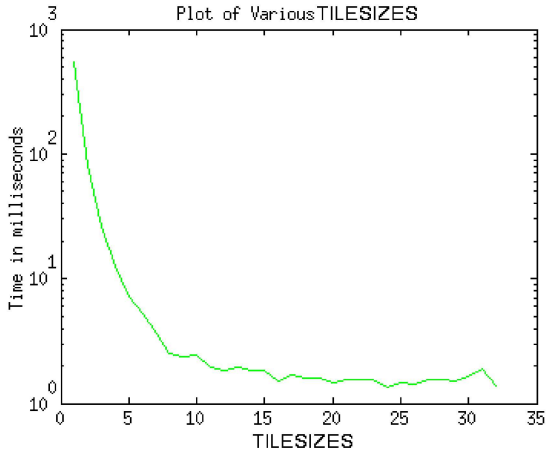

Figure 3.4: Square matrices multiplied with Method 3 using various tilesizes

The size of the matrix being multiplied can also be a factor in choosing the appropriate tilesize. For instance, Figure 3.4 shows that a tilesize of 24 is the best size to choose for that matrix. This is merely because 24 is the largest factor of 456 that is less than 32. This happens for all the matrices and 32 is the consistently best across all sizes. It should also be noted that there is very little improvement after the tilesize is exceeds 16. The

tilesize of 16 appears later in this paper as is discussed there.

Care should be taken when selecting a tilesize due to the way CUDA accesses arrays in shared memory. Say an array is stored in shared memory and then an entry of the array is specified by a dynamic variable. This will use additional shared memory. But if the position in the array is a constant known at compile time, then no additional shared memory will be used. Table 3.5 shows the amount of memory required in bytes to store tiles if the matrix is single precision. MNFDV is the amount of shared memory used by the tiles and the shared memory that is used when accessing the tiles dynamically. It should also be noted that the total amount of shared memory available is 49152 bytes, for the GPUs in Monk.

Tilesize	2 tiles	2 tiles MNFDV	Available
8 x 8	512	1024	49152
16 x 16	2048	4096	49152
32 x 32	8196	16384	49152
64 x 64	32768	65536	49152

Table 3.5: Shared memory usage for various tilesizes

Table 3.5 shows that the largest tilesize we can choose is 32 *times*32. Ideally we always choose the largest tilesize possible. If we declare two tiles of size 64×64 we calculate that it will not use all of the shared memory. But if we want to access elements in the tile, where the position is referenced using a dynamic variable, then it uses 65536 bytes which is more than is available. This is how we are restricted in choosing our tilesize.

3.1.4 CUBLAS

CUBLAS is a C extension library that performs basic linear algebra operations using an Nvidia GPU. This library is updated frequently and is highly optimized, taking into consideration various aspects of individual graphics cards. A problem with CUBLAS is that it is not open source, so checking for an exact reason why something is happening faster or slower is not entirely possible. Amongst its many functions there is one for matrix-matrix multiplication.

The multiplication method that CUBLAS uses is the same as Method 3, but it uses many fine-tuned optimizations that use registers as well as shared memory for the tiles. This is why CUBLAS performs better than the previously described methods. Also note that CUBLAS is actually doing $C = \alpha \cdot \text{ops}(A) \cdot \text{ops}(B) + \beta \cdot C$ where $\text{ops}(X)$ is a function that either uses X or X^{T}, and α and β are constants.

On the surface it appears that making a function that does matrix-matrix multiplication with neither the $\text{ops}(X)$ operation nor the addition of $\beta \cdot C$ should beat CUBLAS, but a closer inspection reveals otherwise. As it turns out, the $\text{ops}(X)$ function does not incur any significant computational expense because when the tiles are being read in from global memory, it does not take any more time to read them in transposed form. Reading in a tile from C does not incur extra waiting time because the cycles spent waiting overlap with the time waiting to get the tiles from A and B numerous times. The multiplications by α and β are just two multiplications in a thread, and are insignificant compared to accessing global memory. In consideration of these details, it is actually not significantly more expensive to perform a matrix-matrix multiplication with these additional arguments.

3.2 Comparison of Methods

The various methods were implemented and tested on square matrices of increasing sizes. The results were as expected. In the chart below, all four methods are shown in a log-log plot.

In Figure 3.5, $N = m = z = n$ for our matrix sizes from $C = A * B$. Note that the number of operations in these methods are $O(N^3)$. The horizontal axis is the cube root of N^3. The vertical axis is the time in milliseconds. The red points represent Method 1, which only used 1 core per SM. The blue points represent Method 2, where all cores in an SM were used but all memory accesses are global. The green points represent Method 3, where tiles were read into the shared memory to minimize the total number of global memory accesses. The cyan points represent the CUBLAS library function cublasSgemm.

As expected, Method 1 is slow when compared with the others. Method 2 performs better than expected when compared with Method 3. Since Method 3 minimizes the global memory accesses by using tiles, we expected a great difference between it and the previous

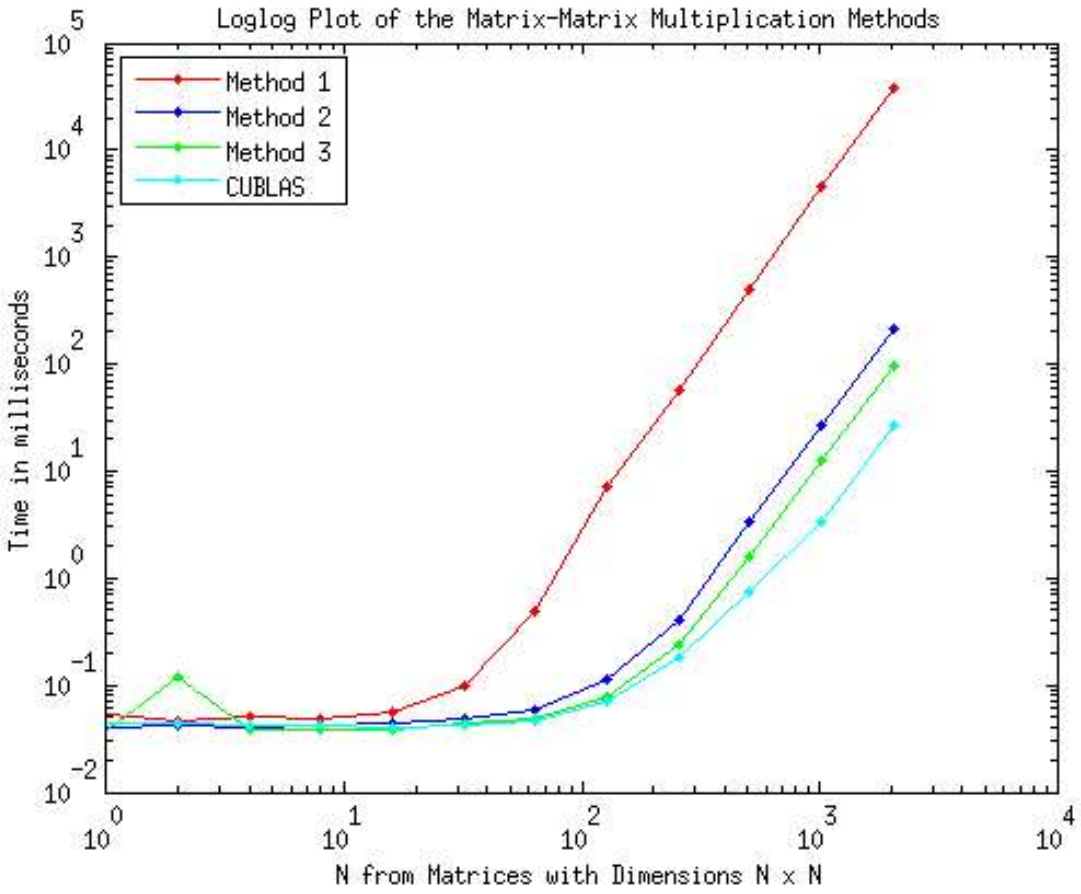

Figure 3.5: A comparison of the matrix-matrix multiplication methods described previously

methods. The explanation of this behaviour is due to the L1 cache: the first memory access into the A and B matrix is global, but the next thread that accesses the row in A or column in B already has the value stored on chip. This greatly increases the performance of the method. Since Method 2 does not use tiles for the matrices A and B, the automatic caching that is performed makes Method 2 almost as good as Method 3. This indicates that for more than a small increase in performance, optimizations must be very well planned because CUDA is not well enough exposed for developers to see how the cache and threads will be handled.

CUBLAS performed better than the other methods. This result is good because in using

CUBLAS we know that in the future it will be competitive and kept up-to-date, whereas a method designed now may not work well in the future or with other graphics cards.

In Figure 3.6, the efficiency of the methods are compared by dividing the time by the number of operations in a matrix-matrix multiplication. This allows us to see when the GPU becomes saturated and most efficient.

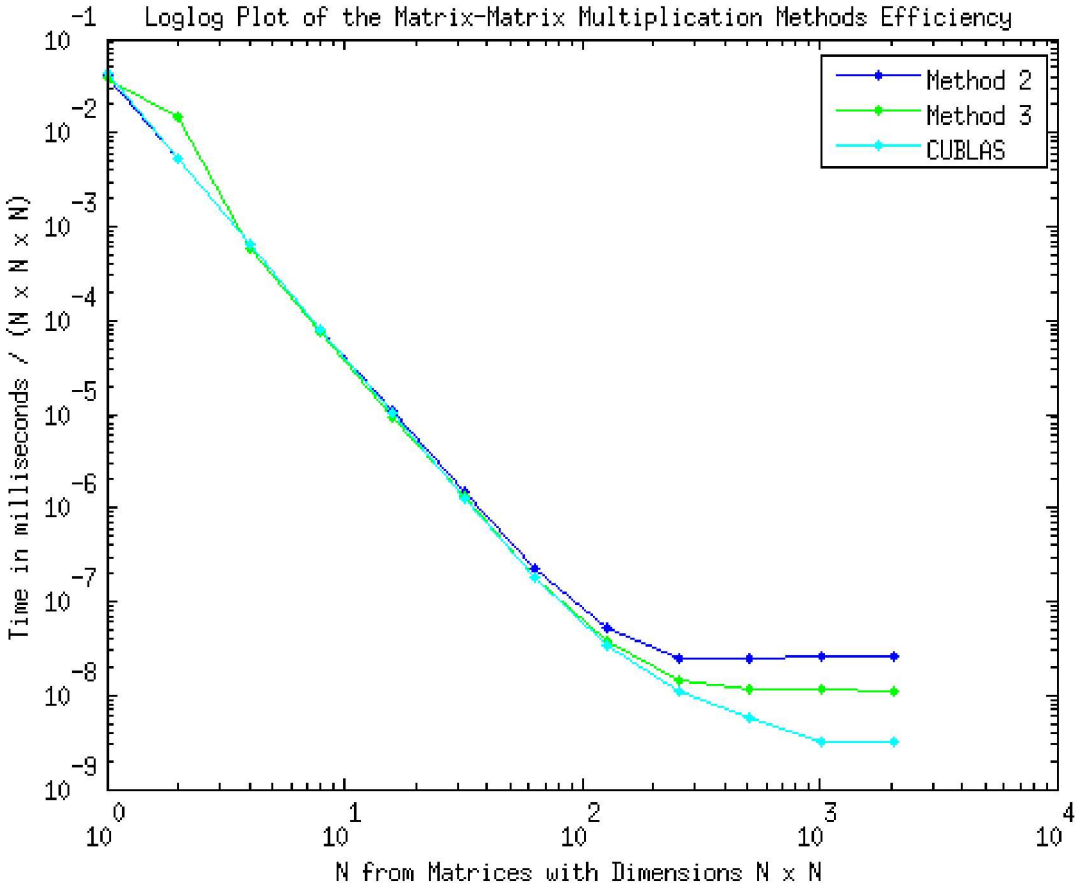

Figure 3.6: Comparison of efficiency of matrix-matrix multiplication methods described previously

In Figure 3.6, $N = m = z = n$ for our matrix sizes from $C = A * B$. Note that the number of operations in these methods are $O(N^3)$. The horizontal axis is the cube root of N^3. The vertical axis is the time in milliseconds it takes for a single operation in the $O(N^3)$ method.

It seems that Method 2 and Method 3 plateau when $N = 128$. For CUBLAS, its plateau occurs at 1024, which means the matrices are large in comparison to those of our problem. In the setting of our problem, a single matrix-matrix multiplication will not be enough to saturate the GPU, because our matrices are quite small compared to a matrix with dimensions 1024 by 1024. So we have to explore other ways to use the CUBLAS library to saturate the GPU.

This section contains results that suggest that using CUBLAS is the best way to approach our problem. It also has the benefit of being updated by the GPU producer so it seems likely that it will remain competitive in the future. Also since CUDA is guaranteed to be backwards compatible with previous versions, any future changes in the CUBLAS library, will affect our code. It seems that using the GPU manufacturer's library has many benefits.

3.3 Pinning Memory

One technique for transferring data to the GPU faster is 'pinning the memory'. Usually, when memory is requested by a process, the operating system (OS) allocates it in the first available spot that is large enough. It may then be moved around for the convienence of the OS. This is not convienent for CUDA, so there is a more suitable way to declare the memory: cudaMallocHost function, as described in [2], is the way to declare memory that is 'pinned' to a fixed location in the CPU host memory. It takes much longer to allocate because the OS tries to find the most convient spot for it to have memory unavailable until the process releases it. Because the location of the memory is pinned, it can be copied or transferred through connections on the motherboard much faster. Since the GPU is accessed through a PCIe slot, having the memory pinned allows data transfer to be unhindered and reach optimal transfer rates. The negative aspect of using the pinned memory is that it takes much longer to allocate than unpinned memory. If the program is repeatedly declaring memory and then transferring it, it could be faster to have it unpinned. In our case, it is preferable to have pinned memory because the memory is declared once, but repeatedly transferred. The speedup of transferring data using this is roughly 2x. Figure 3.7 shows the time required to transfer the data over the PCIe 2.0 connection on Monk

nodes.

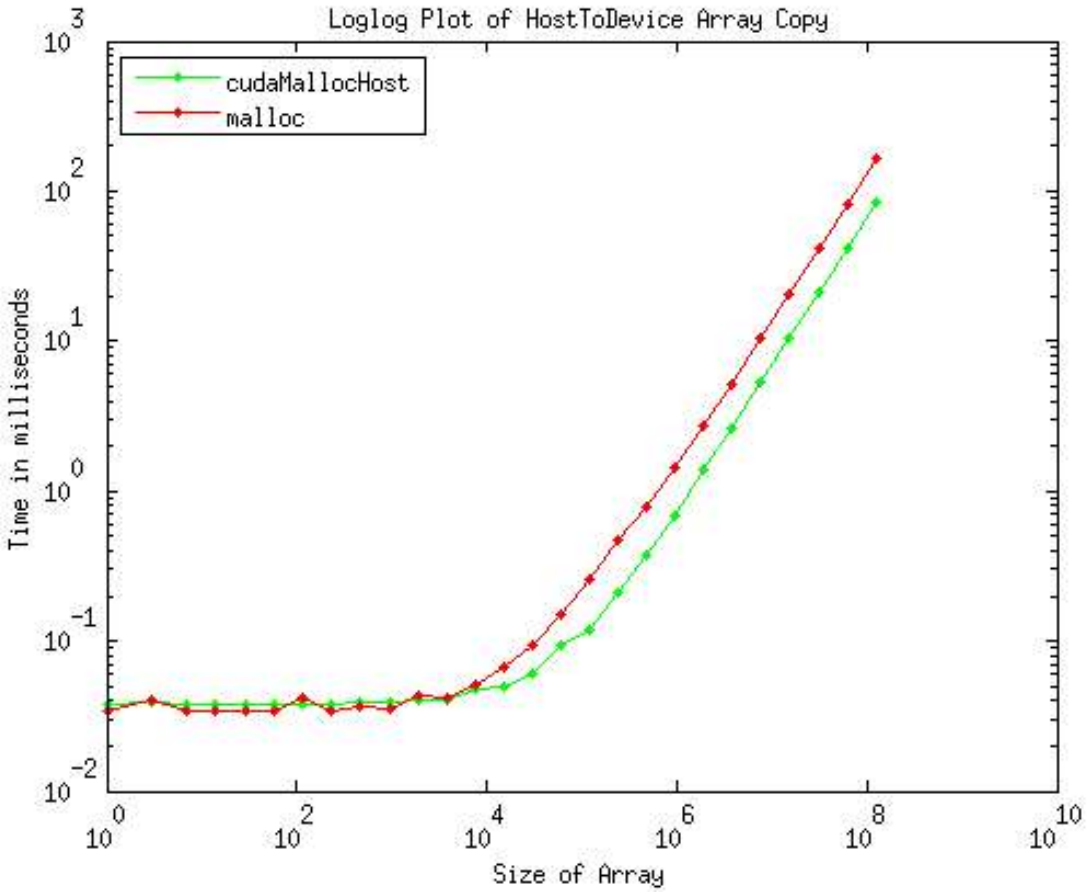

Figure 3.7: Transferring 'pinned' and 'unpinned' memory. The horizontal axis is the length of a float array being transferred from the Host to the Device. The vertical axis is the time in milliseconds. The green points represent memory that was allocated using cudaMallocHost, and the red points represent memory that was allocated using malloc.

What is noticeable is that there is a base cost for calling the memory transfer operation. For the small sizes of arrays up to 10^4 there is a plateau in the timings. After that, the cudaMallocHost memory transfer takes half the time of transferring memory declared

with malloc. The plateau represents an area of inefficient which needs to be avoided when transferring the data. Figure 3.8 describes the efficiency.

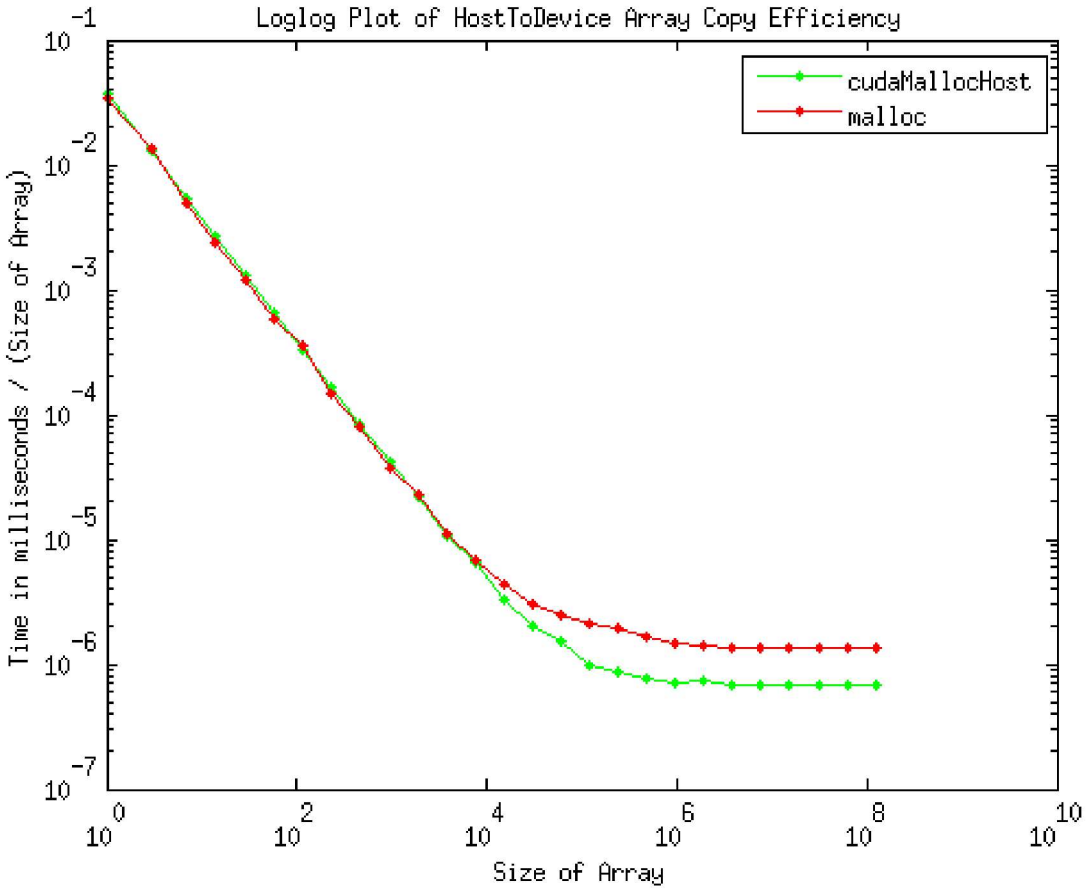

Figure 3.8: Efficiency of memory transfer methods. The horizontal axis is the length of the float array being transferred. The vertical axis is the time in milliseconds divided by the size of the array being transferred. This gives an accurate estimate of the time it takes to transfer an element in the array. The green points represent pinned memory and the red points indicate regular memory.

The plateau in Figure 3.8 indicates that the optimal speed for transferring elements has

been reached. This means that transfers should ideally be done in lengths of at least 10^4 for float arrays.

The use of pinned memory can be regulated by the user. As noted in [2] "Pinned memory should not be overused. Excessive use can reduce the overall system performance because pinned memory is a scarce resource. How much is too much is difficult to tell in advance, so as with all optimizations, test the applications and the systems they run on for optimal performance parameters" Declaring many arrays and allocating them with pinned memory may not work if the OS places them apart so that other arrays cannot fit between them, because this may use much more memory. Since array storage is contiguous, placing a pinned array of length 10 at position 0 and then another one at position 19 may be wasteful if trying to pin another array of length 10. The 9 spaces from position 10 to position 18 can not be used by the array of length 10. The way around this is to have all the small arrays stored in one large array allowing the OS to pin all of the arrays in one section of contiguous memory.

3.4 CUBLAS for matrix-matrix multiplication

3.4.1 Square Matrix Benefits

The CUBLAS library may have hidden properties which allow matrix-matrix multiplication to be performed faster if the matrices are square. In Method 3, we observed that the matrix-matrix multiplications performed faster when the matrices were accessed in tiles. That observation together with [9] suggest that multiplying square matrices may be faster than rectangular matrices. To investigate this, all square matrix-matrix products and all non-square matrix-matrix producs were timed and recorded for matrices with all dimensions less than 128.

The figure below is the results of this investigation of the $C = A * B$ where C is an $m \times n$ matrix, A is an $m \times z$ matrix and B is a $z \times n$ matrix. A matrix-matrix product consists of $O(mzn)$ operations so the cube root of mzn can be thought of as the dimension of one side of the matrix. There are many matrices of different sizes but they have been grouped by this cube root so that they are being averaged with matrix-matrix products with the same number of operations. The cube root is along the horizontal axis and the recorded time of the product is along the vertical axis in milliseconds. Green points indicate square matrix-matrix products and blue points indicate non-square matrix-matrix

products. The red points indicate an observed curiosity that will be discussed later in this subsection.

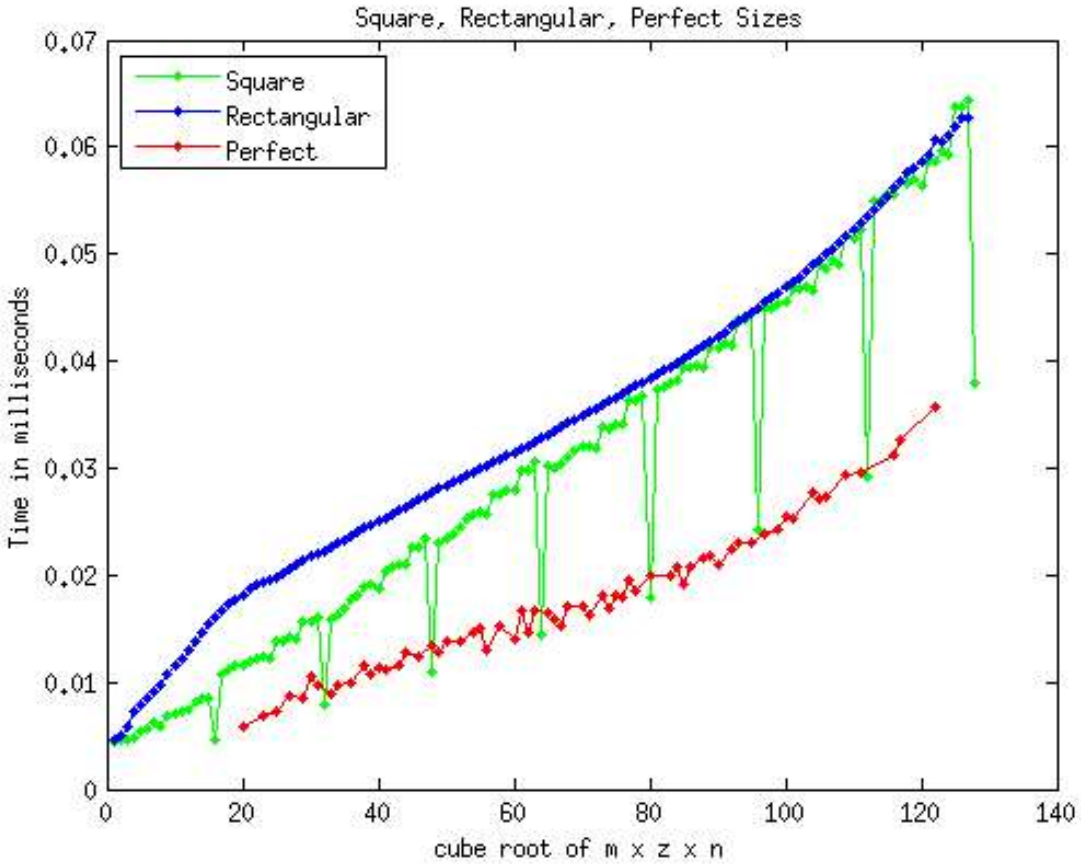

Figure 3.9: Perfect sizes do not include square matrices

From Figure 3.9, it is apparent that CUBLAS can perform matrix-matrix multiplications faster when the matrices are square rather than when they are rectangular.

Also noticeable is a pattern of consistent sharp downward spikes among the green points. They occur evenly at intervals of 16. This regular behaviour suggests that certain square

matrix sizes can perform significantly better than other shapes. These intervals of 16 suggest that perhaps matrices whose dimensions divide evenly into 16 perform better than those matrices that do not, regardless of being square or non-square. The matrix-matrix products whose matrix dimensions divided evenly into 16 were averaged separately and are the red points on Figure 3.9, they are labelled as perfect. It is apparent that matrices whose dimensions divide evenly into 16 perform significantly better than those whose dimensions do not.

The matrices that divide evenly into 16 may also plateau before the other matrices in efficiency, which may partially explain the better run times. Below is a chart of the efficiency.

Figure 3.10 shows that matrix-matrix multiplication is performed faster when the matrices have dimensions that are multiples of 16. The difference is not due to faster saturation of the GPU for matrices wih dimensions that are multiples of 16. The code is simply faster for these matrices.

3.4.2 Matrices with Padding

To pad a matrix is to increase the size of the matrix by adding more rows and columns that are filled with zeros. In the previous section, it was shown that matrix-matrix multiplication is performed faster when the matrices have dimensions that evenly divide into 16. It is also suggested in [2] and [9] that sometimes padding may increase performance. Padding the matrices to these sizes may actually decrease the time required for the matrix-matrix multiplication. When dealing with the padded matrices, the same number of meaningful operations are being carried out, even though more actual operations are being performed. To actually pad the matrix itself requires time since allocated memory is not, by default, zero in the memory type of the array, so it needs to be manually set to zero. If the matrices are being allocated to memory repeatedly, setting the entries to zero may potentially incur more time than is gained by the faster matrix-matrix multiplication.

There are many costs associated with padding the matrices. More memory is being taken up on both the GPU device and the CPU host. Depending on the system, this additional memory usage may be prohibitive, either on the device or possibly on the host. Transferring the matrices will take more time because the matrix has become larger, even though the same amount of useful data is being transferred.

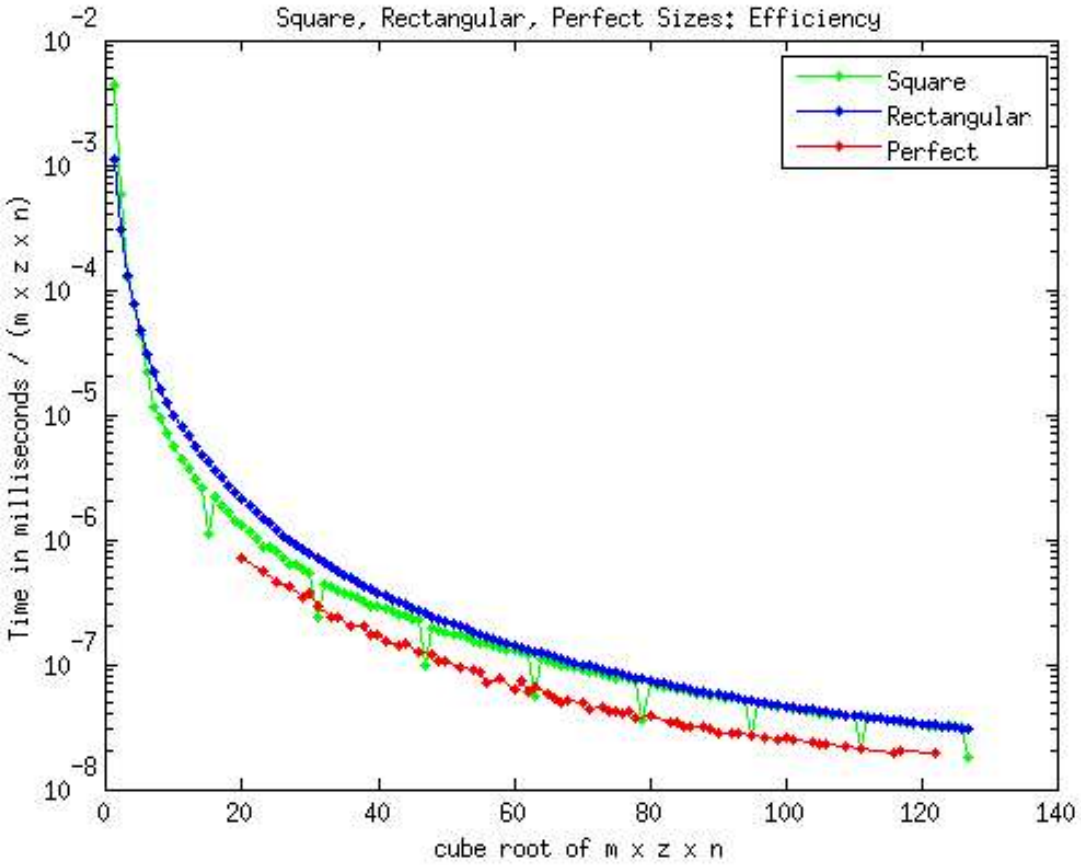

Figure 3.10: The horizontal axis is $\sqrt[3]{mzn}$ because the number of operations is $O(mzn)$. The vertical axis represents the time per operation. The blue points represent the rectangular matrices whose dimensions are not multiples of 16. The green points represent the matrices whose dimensions are square and not multiples of 16. The red points represent the matrices whose dimensions are multiples of 16.

In Figure 3.11, padded matrices are compared with unpadded matrices.

Clearly it is better to pad the matrices with zeros, so that the dimensions divide evenly

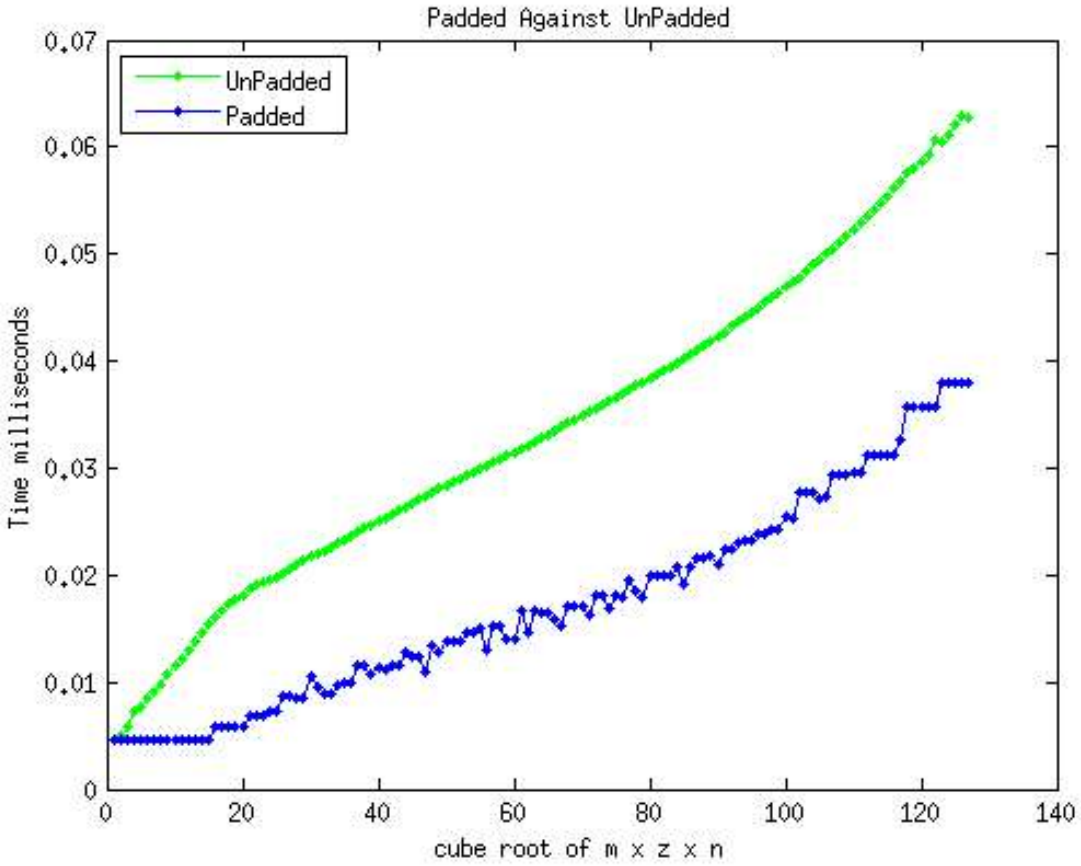

Figure 3.11: The horizontal axis is $\sqrt[3]{mzn}$ (without padding). The vertical axis is the recorded time for the multiplication in milliseconds. The green points represent the matrices without padding. The blue points represent the matrices with padding.

into 16. The padded matrix-matrix multiplications take approximately half the time of the unpadded matrix-matrix multiplications.

Figure 3.12 shows an investigation into the efficiency of padding.

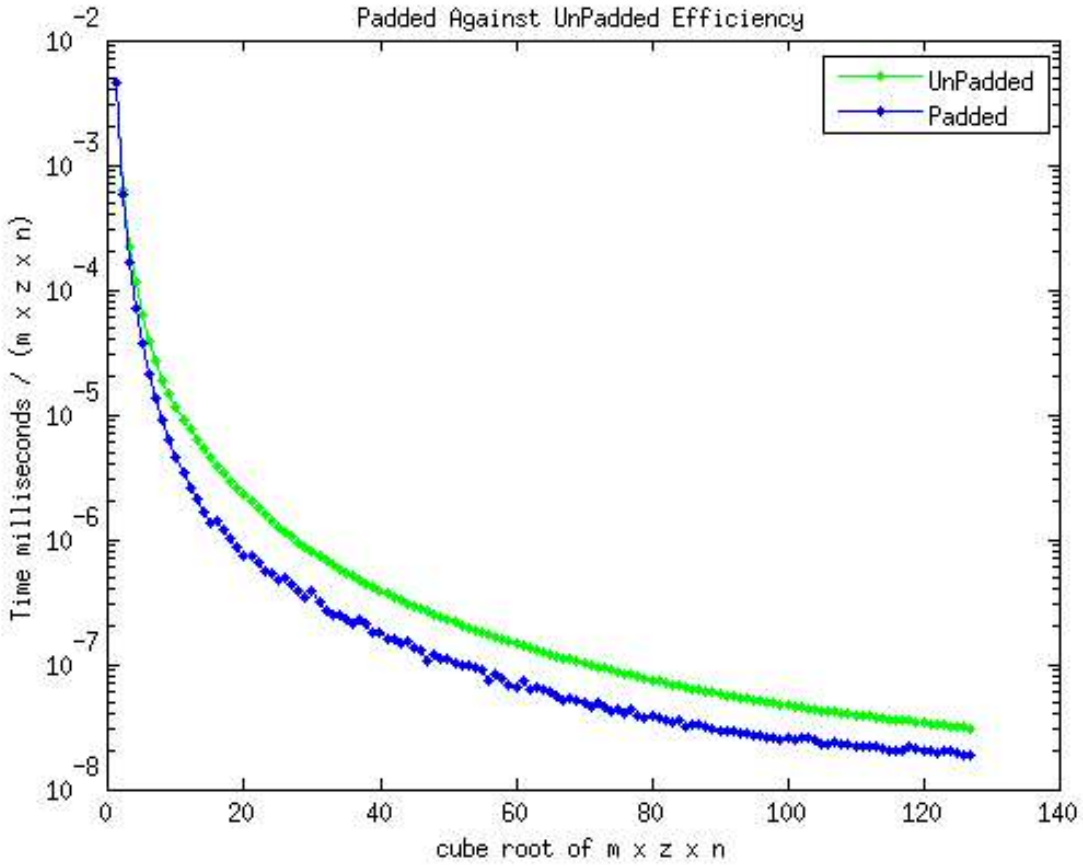

Figure 3.12: The horizontal axis is $\sqrt[3]{mzn}$. The vertical axis is the time taken by each operation. The green points represent the unpadded matrices and the blue points represent the padded matrices. It is clear that the padded matrices are more efficient even though more operations are being performed for the same number of meaningful operations in the unpadded matrices.

Looking at the charts, it is clearly far better to pad the matrices with zeros. Even the transfer times should not suffer greatly because if a matrix of size $M \times N$ is being transferred, then the additional memory that padding needs is $O(M + N)$. So when compared to the $O(MN)$ needed anyway, it is normally not too much.

3.4.3 CUBLAS batched

There is a batched version of most CUBLAS functions, which allows many CUBLAS operations to be combined within a single call, possibly improving performance by better saturating the GPU. The batched version of matrix-matrix multiplication uses a more elaborate algorithm than the nonbatched version, so that the GPU is saturated while performing the computations. This means that the same number of multiplications are performed in less time. Using the batched version is very beneficial because of this and also because multiplying many matrices requires a for loop on the CPU and multiple calls to the GPU. We avoid the calling overhead with this batched version.

Knowing how the batched version compares with the non-batched version may give a better insight into how the batched version works. Figure 3.13 is the batched version against the non-batched version for one matrix-matrix multiplication. The results were generated by averaging the times for all matrices with sizes less than 129 being multiplied together.

It is curious that the batched method performs better than the non-batched method for some matrix sizes. For matrix-matrix multiplication where the cube root of the number of operations is less than 100, it is better to use the batched method, even though there is only 1 pair of matrices being multiplied. It also suggests that the algorithm that is implemented in the library may be quite different for the non-batched method.

Figre 3.14 is a comparison of the batched and non-batched methods for 1000 matrices. There is one call to the batched method and 1000 calls to the non-batched method.

Now, the difference between the batched method and the non-batched method seems to be rather large, with the batched method performing very well. It seems that using the batched method is clearly the better option for handling our problem. As the matrices become larger, the difference between the two methods appears to be shrinking. It is likely that the difference between the two methods will continue to shrink and become negligible when the $\sqrt[3]{mzn}$ exceeds 150. In our fluid dynamics problem, $\sqrt[3]{mzn}$ ranges from 10 to 40. When inspecting that region of the graph it appears that the batched method runs in under a tenth of the time of the non-batched method. The reason for this behaviour is likely

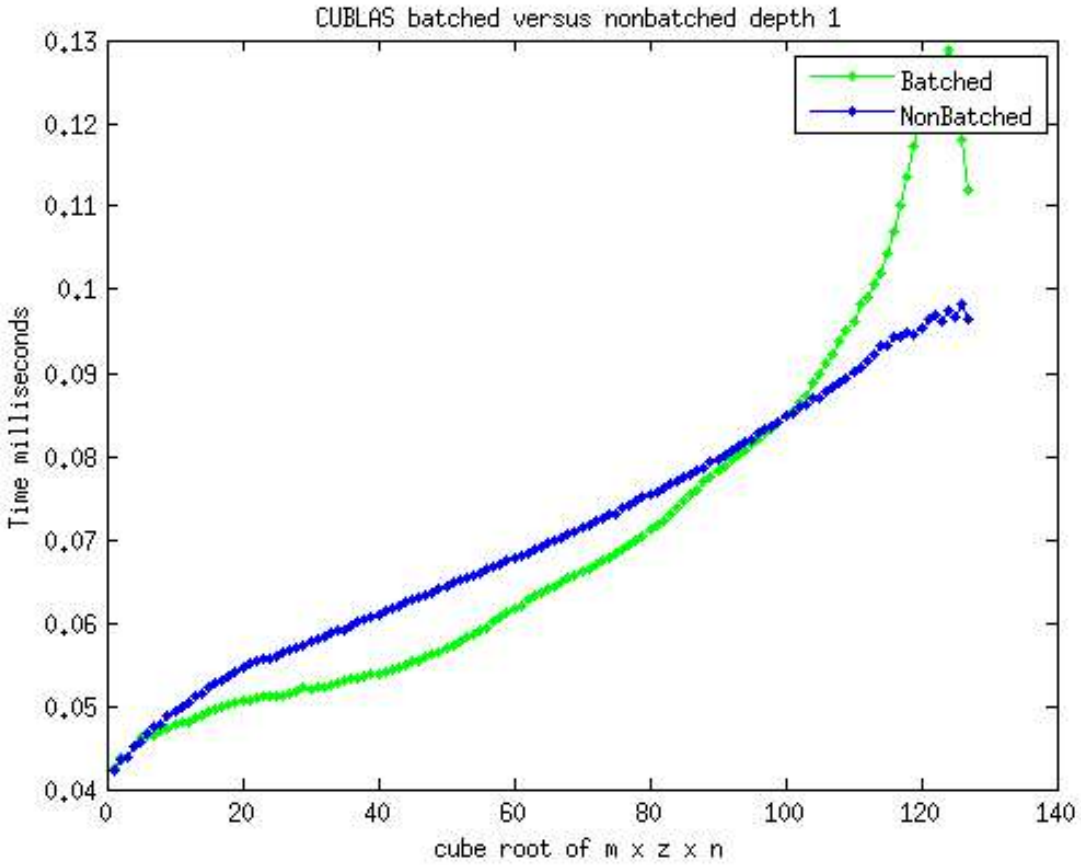

Figure 3.13: The horizontal axis is $\sqrt[3]{mzn}$. The vertical axis is recorded time in milliseconds. The green points represent the times for the batched version, and the blue points represent the non-batched version.

due to the small matrices not saturating the SMs completely, whereas the batched version executes multiple products on the same SM. Therefore, the batched version is probably approaching its optimal efficiency much faster than the non-batched method as a function of matrix size.

In Figure 3.15, the times of Figure 3.14 were divided by $\sqrt[3]{mzn}$ to yield the approximate time taken by an individual operation.

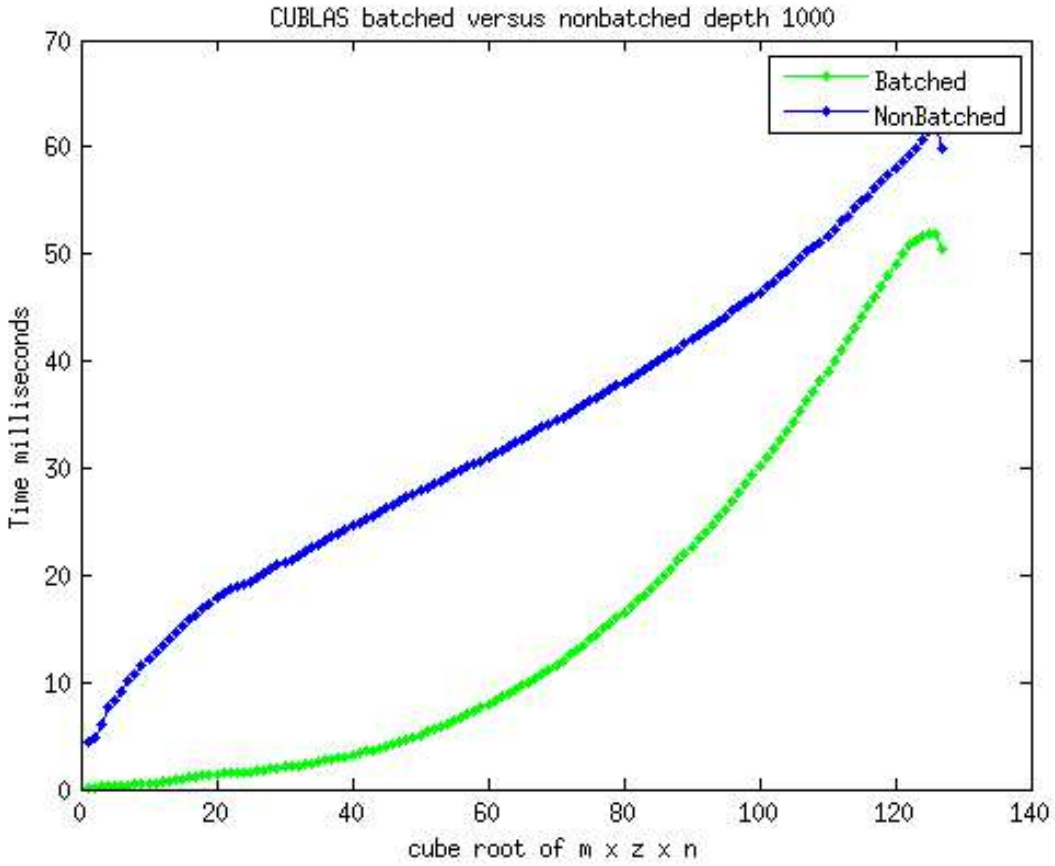

Figure 3.14: The horizontal axis is $\sqrt[3]{mzn}$. The vertical axis is the recorded time in milliseconds for 1000 matrix-matrix products to be computed. The green points represent the times for the batched version, and the blue points represent the non-batched version.

There is a plateau that approximately begins when the cube root of the number of operations reaches 40. It means that the matrix-matrix multiplication is just as efficient at 40 as it is at greater sizes when using the batched method. In our problem, $\sqrt[3]{mzn}$ ranges from 10 to 40, so for most cases batching 1000 matrix-matrix multiplications is not enough to saturate the GPU. Looking at the non-batched method shows that it is indeed less efficient. It also shows that the GPU is not being saturated. The efficiency of the batched

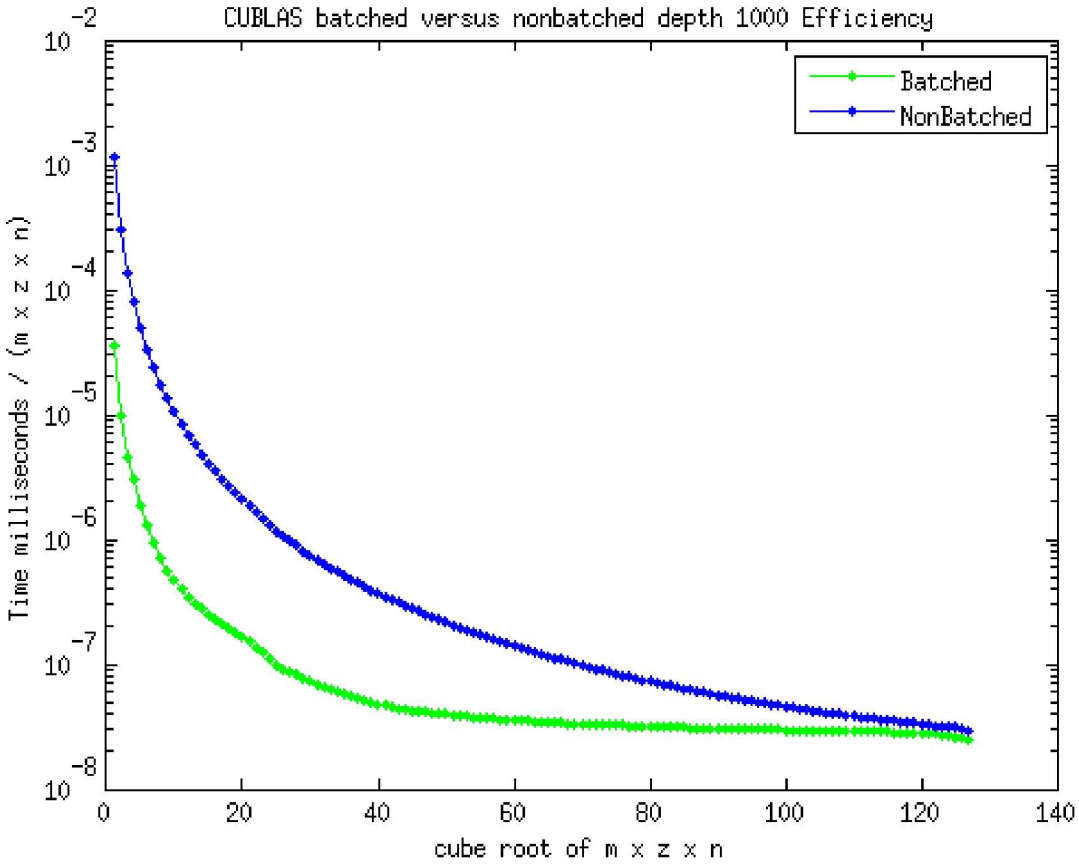

Figure 3.15: The horizontal axis is $\sqrt[3]{mzn}$. The vertical axis is recorded time in milliseconds for approximately a single operation to be performed. The green points represent the times for the batched version, and the blue points represent the non-batched version.

method is an order of magnitude better than the non-batched version between 10 and 40 on the horizontal axis. The non-batched version most likely plateaus as well, just beyond the current range.

The batched version of the CUBLAS matrix-matrix multiplication method is superior to its non-batched counterpart within the specifications of our problem. It saves CPU time by using only a single call to the GPU instead of a loop. We saw that for many matrices, the batched method performed significantly better. There is also evidence suggesting that the

batched version saturates the GPU much more effectively than many non-batched calls. Batching also helps in the following section.

3.5 Overlap of Communication and Computation

3.5.1 Overlap of Transfer and Kernel

In the layout of the GPU, it was shown that there is a separate processing unit, the copy engine, that only handles the transfer of data. The GPU is able to use the copy engine and the SMs simultaneously. This allows memory to be transferred to the GPU while it is performing computations on the data that is already present. This approach to computing on the GPU is described in [9]. The overlap should reduce the overall time required for the entire matrix-matrix multiplication result to be given to the CPU.

Ideally, the time of the transfer operations will be hidden in the computation operations. More generally, one may strive to either hide the compute time in the transfer time, or the transfer time in the compute time.

Figure 3.16 is a visual representation of what is being attempted with the overlap of computation and transfer. It shows that 33% of the time can be saved in this manner.

Another method would be to start transferring back the first set of results the moment they are finished. Unfortunately, this would prevent the third set of matrices from being sent to the GPU on time. In the case of 4 sets, the actual time for the whole procedure would still be the same as the original time. But the positions of the computation are spread out. When dealing with multiple CPUs sharing 1 GPU, this detail needs to be handled with more elaborate strategies, otherwise the gain from overlapping transfer and computation may be lost.

3.5.2 Case Study

The situtation of one CPU and one GPU will be considered. The overlapping of transfer and computation is much more difficult to balance in the setting of our problem because

Figure 3.16: In the top part, a large set of matrices is transferred over to the GPU (light red). The kernel is executed for the set of matrices (dark red). Then the large set of matrices is transferred back to the host (light red). In the bottom part, the large set of matrices is split into 4 smaller sets. The first smaller set is transferred to the GPU. When the transfer is complete, the computation begins on the first smaller set of matrices. While this is happening, the second small set of matrices is transferred over. This processes repeats itself until all of the small sets of matrices have been transferred over. The moment the last set has been sent to the GPU, the results of the first computation are transferred back while the last computation is occurring. When that transfer is finished the second set of results is transferred back, and this process is repeated until all the results have been transferred back.

the matrices A and B are bigger than C, not significantly, but just enough to cause problems. For instance, take m, z, n to be equal to 19, 32, 9. So C is 19×9, A is 19×32, and B is 32×9. Then matrix A has 608 elements, matrix B has 288 elements, and matrix C has 171 elements. Together A and B have 896 elements. These sizes are important because in this case calc the amount of memory being transferred to the GPU is roughly 5x the size of the amount of memory being transferred from the GPU. A difficulty arises because the chunksize for transferring A and B at optimal speed may not be the chunksize for transferring C at an optimal speed. Knowing how to balance this requires a significant amount of experimentation. To our advantage is the fact that all of these operations and

transfers happen in milliseconds, so a satisfactory balance can be found quickly by checking many different parameter choices.

Figure 3.17 shows a graph of the total time for a set of matrices to be transferred to the GPU, to be multiplied together, and to be transferred to the host. The set of matrices that was transferred to the GPU consisted of one thousand A and B pairs, where A is a 19×32 matrix and B is a 32×9 matrix. The kernel is a batched CUBLAS matrix-matrix multiplication function. Then 1000 C matrices, of size 19×9, are transferred to the host. The graph below shows the timings for this, as the set of 1000 matrices is split up into chunksizes. This is done in order to find an optimal overlap of memory transfer and computation.

One aspect of this chart does not appear to be correct. The time it takes for the asynchronous method to transfer 1000 matrices all at once, then perform the computation, then transfer them all back should take as long as the synchronous method, but this is not the case. This discrepancy can be tracked down to the different functions used to copy the memory over to the GPU. The asynchronous method uses cudaMemcpyAsync whereas the synchronous method uses cudaMemcpy. This suggests that if transferring pinned memory, it may be better to use cudaMemcpyAsync, even when there is no actual overlap taking place in the code.

Looking at Figure 3.17 shows that the best chunksize to use is 100 followed by 250 then 125. It is strange that there does not seem to be a reason why 100 and 250 are better chunksizes than 125. This suggests that chunksizes need to be checked for individual problems. It should be noted that while the difference in time between the 100, 125, 250 is small, the saturation of the GPU is also a factor when choosing which chunksize to use. This is important when considering the more general situation of multiple CPUs sharing one GPU. If a chunksize of 100 did not saturate the GPU but was the fastest method then it would be the better choice because the GPU could then saturate itself with the other matrix-matrix multiplications from the other CPUs. The saturation would need to be measured in a case study of multiple CPUs sharing one GPU.

Individually checking all the sets of matrices we have in this problem did not reveal any strong connection to suggest a simple rule for calculating an optimal chunksize. Table 3.6 is a summary of many graphs with the various sizes of data being transferred. The columns are CPU host to GPU device (H2D) consisting of A and B, GPU device to CPU

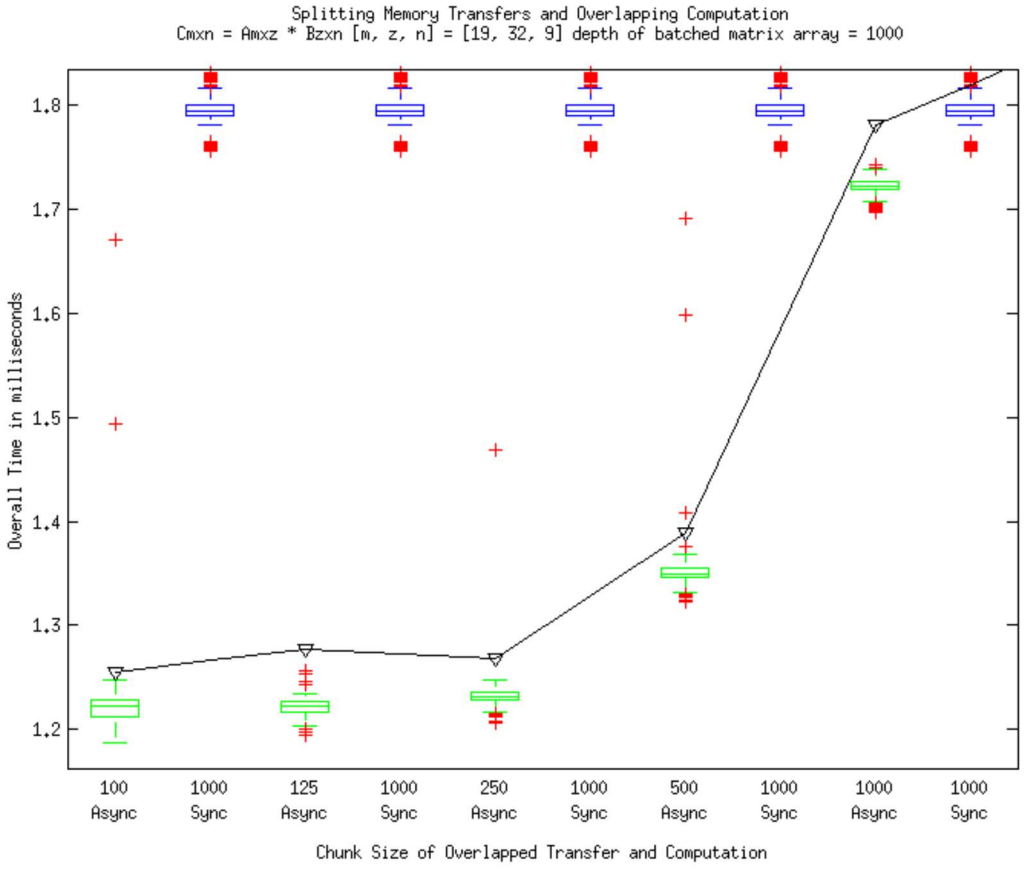

Figure 3.17: The values along the horizontal axis correspond to the method used to perform the task. The Sync values are the timings for the entire set of matrices to be transferred, computed, and sent to the host. The Sync values are repeated on the graph for comparative purposes. The Async indicates the overlapped timing results. The number above the text on the horizontal axis is the chunksize, which is the size of the sections that the 1000 matrix sets (A, B, and C) were divided into, allowing an overlap.

host (D2H) consisting of C, Total size being A, B, and C. Total Optimal Chunk Size is the Total Size column multiplied with the Optimal Chunk Length column, to show how many individual values are being transferred.

Matrix Code	H2D Size	D2H Size	Total Size	Optimal Chunk Length	Total Optimal Chunk Size
9 x 24 x 5	1344	180	1524	250	381000
9 x 24 x 8	1632	288	1920	125	240000
9 x 24 x 9	1728	324	2052	100	205200
9 x 32 x 5	1792	180	1972	250	493000
9 x 32 x 8	2176	288	2464	125	308000
9 x 32 x 9	2304	324	2628	125	328500
9 x 56 x 5	3136	180	3316	125	414500
9 x 56 x 8	3808	288	4096	100	409600
9 x 56 x 9	4032	324	4356	100	435600
19 x 24 x 5	2304	380	2684	250	671000
19 x 24 x 8	2592	608	3200	250	800000
19 x 24 x 9	2688	684	3372	250	843000
19 x 32 x 5	3072	380	3452	125	431500
19 x 32 x 8	3456	608	4064	125	508000
19 x 32 x 9	3584	684	4268	100	426800
19 x 56 x 5	5376	380	5756	125	719500
19 x 56 x 8	6048	608	6656	100	665600
19 x 56 x 9	6272	684	6956	100	695600
34 x 56 x 5	8736	680	9416	100	941600
34 x 56 x 8	9408	1088	10496	100	1049600
34 x 56 x 9	9632	1224	10856	100	1085600

Table 3.6: This is the records of transferring both A and B, but it may be the case that for our fluid dynamics problem, we may be able to store all the A matrices on the GPU. So the A matrices would only need to be transferred once during the initialization of the program.

Chapter 4

Conclusion

Implementing the best approaches that were investigated into the existing MPI code from [6] will require a lot of work. Before doing that, many parts will need to be further tested individually to see if they are beneficial or not. In terms of adapting the MPI code, the first step would be to change the arrays of structures into structures of arrays. This allows the matrices to be transferred contiguously and in user-defined chunk sizes. The best approach to use pinned memory would be only to only allocate the arrays once for each multi-core CPU, during the initialization of the program. The function would declare one incredibly large pinned memory array, divide it up with pointers, and then pass the pointers to all the processes running on the cores of that CPU. This approach seems quite infeasible because there is a lot of message passing due to the multiple processes running on each core and the multiple cores of each CPU. The other reason is that this pinned memory will not be released until the entire program ends. Pinning the memory does seem like a good idea when looking back at Figure 3.7 of the difference in transfer times but in practice, it can be troublesome.

If it is possible to keep all of the A matrices on the GPU, then that would speed up the transfers. This basically halves the total transfer to the GPU, but it may not be possible because of memory restrictions.

The matrices should be padded with zeros to the next highest multiple of 16. The performance gain is too significant to neglect using the idea, even though it uses more memory. A possibility to reduce the memory usage, would be to overlap the A and B matrices in such a way that they share padding. Since A and B are only read, there would be no problem with this, but CUBLAS does not have an option to state that a matrix is stored

starting with the last column through to the first column.

All matrices that reside on the GPU could be copied over with cudaMemcpyAsync because it was observed that it is faster both in the case of multiple transfers and a single transfer. This situation requires a more rigorous investigation into the causes of this behaviour.

Using CUBLAS's batched version of matrix-matrix multiply is required to fully saturate the GPU for small matrices, and is necessary when overlapping data transfer with kernel computation.

References

[1] Christoph Bennemann, Mark W. Beinker, Daniel Egloff, and Michael Gauckler. Teraflops for Games and Derivatives Pricing. *Wilmott Magazine*, 4, 2004.

[2] NVIDIA Corporation. NVIDIA OpenCL Best Practices Guide. Technical report, NVIDIA Corporation, 2009.

[3] Luigi Genovese. Graphic Processing Units: a Possible Answer to High Performance Computing?, 2009. 4th ABINIT Developer Workshop.

[4] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *In SC 08:Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, New Jersey, USA, 2008.

[5] Wikimedia Foundation Inc. CUDA. http://en.wikipedia.org/wiki/CUDA, August 2012.

[6] Lucian Ivan, Hans De Sterck, Scott Northrup, and Clinton Groth. Hyperbolic Conservation Laws on Three-Dimensional Cubed-Sphere Grids: A Parallel Solution-Adaptive Simulations Framework. Submitted, 2012.

[7] David Kirk and Wen mei W. Hwu. Programming Massively Parallel Processors. http://courses.engr.illinois.edu/ece498/al/lectures/lecture5-cuda-memory-spring-2010.ppt, 2010. Accessed using Google cached.

[8] NVIDIA Corporation. CUBLAS. http://developer.nvidia.com/cuda/cublas, August 2012.

[9] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. Technical report, NVIDIA Corporation, April 2012.

[10] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. pages 21–51, 2005.

[11] Sharcnet. About Sharcnet. https://www.sharcnet.ca/my/about, August 2012.

[12] Sharcnet. Monk Documentation. https://www.sharcnet.ca/help/index.php/Monk, August 2012.

[13] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of Sum-Products on GPUs through Software-Managed Cache. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318, New York, New York, USA, 2008.

[14] J. Tolke and M. Krafczyk. TeraFLOP computing on a desktop pc with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22:443–456, 2008.

[15] F. Vazquez, E. M. Garzon, J. A. Martinez, and J. J. Fernandez. The Sparse Matrix Vector Product on GPUs. Technical report, University of Almeria, 2009.

[16] Tim Warburton. General Purpose GPU Computing. http://www.caam.rice.edu/~timwar/RMMC/CUDA.html, 2012. Original source unavailable, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.

[17] Martin Weigel. Simulating Spin Models on GPUs. In *Computer Physics Communications*, volume 182, pages 1833–1836. 2011.