Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model

Robert Hochberg

August 11, 2012

Contents

Chapter 1

Matrix Multiplication

1.1 Overview

It has become increasingly common to see supercomputing applications harness the massive parallelism of graphics cards (Graphics Processing Units, or GPUs) to speed up computations. One platform for doing so is NVIDIA's Compute Unified Device Architecture, or CUDA. We use the example of Matrix Multiplication to introduce the basics of GPU computing in the CUDA environment. It is assumed that the student is familiar with C programming, but no other background is assumed.

The goal of this module is to show the student how to offload parallel computations to the graphics card, when it is appropriate to do so, and to give some idea of how to think about code running in the massively parallel environment presented by today's graphics cards.

1.2 Counting Paths in Graphs

A matrix is a rectangular array of numbers. Nothing more. But despite their simplicity, they are one of the most useful and fundamental mathematical objects in scientific computation. Applications include computer graphics (including any time we wish to show a 3-d world on a 2-d screen) solving systems of equations (such as two equations in two unknowns) DNA sequence comparison, modeling electrical circuits or computer networks, and so on. As mathematical objects, matrices can be added and subtracted, multiplied and, sometimes, divided. Here we will be interested in multiplication.

If you are seeing matrix multiplication for the first time, then it's probably best to see it in some context that helps it to make sense. Otherwise, it can look very arbitrary. Our context will be counting the number of paths in a network. Such questions have applications in analyzing transportation networks [6], DNA sequence comparison (the number of optimal alignments of two DNA sequences can be modeled by counting paths in the scoring matrix) and measuring centrality of a node within a complex network [3]. If the vertices of a graph model a set of states, and the edges of the graph are weighted with probabilities of transitions between the states, then the graph models a *Markov chain* and we have a whole new collection of applications, from age-stratified population models [5] to drug design [4]. So let us learn how to count paths in a network.

Consider the graph in Figure 1.1, showing a collection of locations and some connections between them.

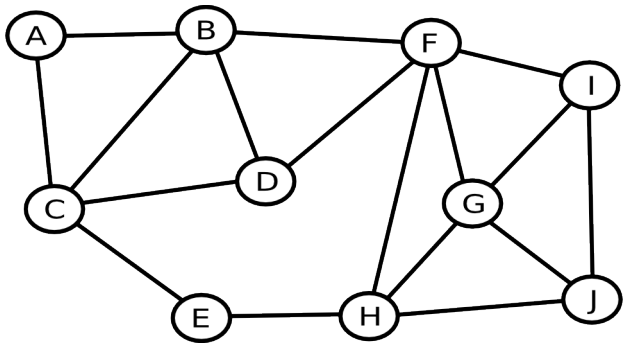


Figure 1.1: A graph with 10 vertices and 17 edges

We will be asking questions about paths in this graph. In this graph there is a path from C to J of length 4: C-D-F-G-J. There is also a shorter path from C to J: C-E-H-J which has length only 3. (Note that length counts the number of edges in the path, which is one less than the number of vertices.) You can quickly convince yourself that there are no other paths of length 3 from C to J in this graph, but there are other paths of length 4. For example, C-B-F-I-J and C-E-H-G-J. We might ask "How many paths are there from C to J of length exactly four?"

It is not too hard to find these by hand, but it is much easier if we are systematic about it. Being systematic will help us to be sure we don't miss any, and that we don't count any twice. So let us begin at vertex C and consider our first step. In one step from vertex C we may go to either A, B, D or E. From those vertices, what is needed to complete the job? Since we are looking for a path of length 4 from C to J, we need a path of length 3 to J from A, B, D or E. Shown in Figure 1.2 is a table giving the number of paths of length 3 between every pair of vertices. (We will see soon how to create this table.)

from/to	A	B	C	D	E	F	G	H	I	J
A	2	6	6	3	1	3	1	2	1	0
B	6	6	8	7	3	9	3	3	2	3
C	6	8	4	8	5	4	3	2	2	1
D	3	7	8	4	2	8	3	3	2	3
E	1	3	5	2	0	3	2	5	3	1
F	3	9	4	8	3	6	10	11	10	4
G	1	3	3	3	2	10	8	9	8	8
H	2	3	2	3	5	11	9	4	4	9
I	1	2	2	2	3	10	8	4	4	8
J	0	3	1	3	1	4	8	9	8	4

Figure 1.2: Number of paths of length 3 between all pairs of vertices.

We now reason as follows: If our first step went to vertex A, then there would be 0 ways to finish the job, since the table shows 0 ways to walk form A to J in exactly 3 steps. A glance at the graph shows that to be true. If our first step went to B, however, then there would be 3 ways to finish the job, as shown in the (B, J) entry in the table. Similarly, from D there are 3 ways to finish the job, and from E there is one way. Adding, we find that there are 7 ways altogether to walk from C to J in exactly 4 steps.

Here is another way to format that calculation: Let us build a 1×10 vector showing the number of paths of length 1 from C to each of the 10 vertices in our graph. The vector would look like this: $[1, 1, 0, 1, 1, 0, 0, 0, 0, 0]$, where the entries are in the order [A, B, C, D, E, F, G, H, I, J]. The entries of the right column in the table in Figure 1.2 show the number of paths of length 3 *to* J from each of the vertices. Our computation amounted to multiplying together the vector and that column, as shown in Figure 1.3.

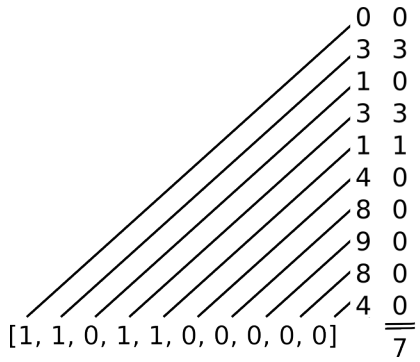


Figure 1.3: Multiplying "paths of length 1 *from* C" by "paths of length 3 *to* J"

The horizontal vector shows the number of paths *from* C, and the first vertical column shows the number of paths of length 3 *to* J. The diagonal line segments connect the numbers that we multiply together, and the product is shown in the rightmost column. There you can see the numbers we added together to get "7" in our previous calculation. The only difference is that we included a "0" entry for those vertices that have no connection to C. For example, the top "0" in the right-most column corresponds to the observation that "there is 1 path of length one from C to A, but 0 paths of length three from A to J," and the "3" entry below that corresponds to "there is 1 path of length one from C to B, and 3 paths of length three from B to J."

This is the basic structure of matrix multiplication. We multiply row entries by column entries, and then add the products.

What we have done is compute a single entry in a table showing the number of paths from C to J of length 4. This table is shown in Figure 1.4.

from/to	A	B	C	D	E	F	G	H	I	J
A	.	.	.	.	.	.	.	.	.	.
B	.	.	.	.	.	.	.	.	.	.
C	.	.	.	.	.	.	.	.	.	7
D	.	.	.	.	.	.	.	.	.	.
E	.	.	.	.	.	.	.	.	.	.
F	.	.	.	.	.	.	.	.	.	.
G	.	.	.	.	.	.	.	.	.	.
H	.	.	.	.	.	.	.	.	.	.
I	.	.	.	.	.	.	.	.	.	.
J	.	.	.	.	.	.	.	.	.	.

Figure 1.4: Number of paths of length 4 between all pairs of vertices. Only the (C, J) entry has been computed.

Suppose that we wished to complete this table. What would be involved? We would need to find a vector for each vertex showing the number of paths of length 1 to each of the vertices, just as we did for vertex C above. If we stack these vectors, they form a 10×10 table of their own. This table is shown in Figure 1.5.

This table can be thought of in two ways: First, it shows the number of paths of length 1 between each pair of vertices. Alternatively, you can think of it as simply showing which pairs of vertices are connected by an edge (they have a "1" in the table) and which pairs are not connected (they have a "0" in the table). When we think of that table in the second way, it's called an *adjacency matrix*, "adjacent" being the term used to say that two vertices have an edge between them — they are *adjacent*.

from/to	A	B	C	D	E	F	G	H	I	J
A	0	1	1	0	0	0	0	0	0	0
B	1	0	1	1	0	1	0	0	0	0
C	1	1	0	1	1	0	0	0	0	0
D	0	1	1	0	0	1	0	0	0	0
E	0	0	1	0	0	0	0	1	0	0
F	0	1	0	1	0	0	1	1	1	0
G	0	0	0	0	0	1	0	1	1	1
H	0	0	0	0	1	1	1	0	0	1
I	0	0	0	0	0	1	1	0	0	1
J	0	0	0	0	0	0	1	1	1	0

Figure 1.5: Number of paths of length 1 between all pairs of vertices.

Implementation Comment: Note that our adjacency matrix is symmetric, since an edge from X to Y is also an edge from Y to X. This means that in an implementation we would need to store only those entries above the main diagonal, from which the lower entries could be inferred. This cuts in approximately half the number of entries we need to store. Or, said another way, in the same amount of memory, we could store a graph that is about 70% larger if we take advantage of this symmetry.

1.3 Multiplying Matrices

We are now in a position to describe matrix multiplication. We re-write each of the tables as matrices, as shown in Figure 1.6. The product of these two matrices is the matrix showing the number of paths of length 4 between each pair of vertices.

The top-right entry in the product matrix would give the number of paths of length 4 from A to J in the graph. One such path would be A-B-F-I-J, and another would be A-C-E-H-J. To get the total number, we would multiply together the top row in the first matrix (the row showing paths of length 1 *from* A) and the rightmost column of the second matrix (the column showing the number of paths of length 3 *to* J. We get the sum

$$0 \cdot 0 + 1 \cdot 3 + 1 \cdot 1 + 0 \cdot 3 + 0 \cdot 1 + 0 \cdot 4 + 0 \cdot 8 + 0 \cdot 9 + 0 \cdot 8 + 0 \cdot 4 = 4.$$

So the rightmost entry in the top row of the product would be "4".

Let's introduce a bit of notation regarding matrices. An $m \times n$ matrix has m rows and n columns, and the (i, j) entry is the entry in the ith row, jth column. If M is the name of the matrix,

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \cdot \left(\begin{array}{c|c|c|c|c|c|c|c|c|c} 2 & 6 & 6 & 3 & 1 & 3 & 1 & 2 & 1 & 0 \\ 6 & 6 & 8 & 7 & 3 & 9 & 3 & 3 & 2 & 3 \\ 6 & 8 & 4 & 8 & 5 & 4 & 3 & 2 & 2 & 1 \\ 3 & 7 & 8 & 4 & 2 & 8 & 3 & 3 & 2 & 3 \\ 1 & 3 & 5 & 2 & 0 & 3 & 2 & 5 & 3 & 1 \\ 3 & 9 & 4 & 8 & 3 & 6 & 10 & 11 & 10 & 4 \\ 1 & 3 & 3 & 3 & 2 & 10 & 8 & 9 & 8 & 8 \\ 2 & 3 & 2 & 3 & 5 & 11 & 9 & 4 & 4 & 9 \\ 1 & 2 & 2 & 2 & 3 & 10 & 8 & 4 & 4 & 8 \\ 0 & 3 & 1 & 3 & 1 & 4 & 8 & 9 & 8 & 4 \end{array}\right)$$

Figure 1.6: Multiplying the "paths of length 1" matrix by the "paths of length 3" matrix.

then the notation $M_{i,j}$ means the (i, j) entry. Generalizing the discussion above, we can compute the product of two $n \times n$ matrices A and B to obtain another $n \times n$ matrix as follows: To find the (i, j) entry in the product matrix, we multiply the ith row of A by the jth column of B, computing n products and summing them. In Figure 1.6 we indicated this by delineating the rows of the first matrix and the columns of the second matrix.

Practice 1: Three Entries
Before reading further, make sure that you can multiply matrices. Let P be the product of the two matrices in Figure 1.6. Show that $P_{2,3} = 22, P_{5,5} = 10$ and $P_{10,10} = 25$.

The table in Figure 1.7 shows the numbers of paths of length 4 between all pairs of vertices.

Now we can turn our attention to the question of finding the matrix showing the number of paths of length 3. (You've probably already figured it out.) Just as we obtained the matrix showing the number of paths of length 4 from the matrix showing the number of paths of length 3, we can obtain the length-3 matrix from the length-2 matrix by multiplying by the adjacency matrix. This computation is shown in Figure 1.8.

And how do we obtain the length-2 matrix? By multiplying together two copies of the adjacency matrix. (Recall that the adjacency matrix gives the number of paths of length 1.)

from/to	A	B	C	D	E	F	G	H	I	J
A	12	14	12	15	8	13	6	5	4	4
B	14	30	22	23	11	21	17	18	15	8
C	12	22	27	16	6	23	9	13	8	7
D	15	23	16	23	11	19	16	16	14	8
E	8	11	6	11	10	15	12	6	6	10
F	13	21	23	19	15	48	31	23	20	31
G	6	17	9	16	12	31	35	28	26	25
H	5	18	13	16	6	23	28	34	29	17
I	4	15	8	14	6	20	26	29	26	16
J	4	8	7	8	10	31	25	17	16	25

Figure 1.7: Number of paths of length 4 between all pairs of vertices.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 4 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 2 & 4 & 1 & 0 & 2 & 0 & 1 & 0 & 0 \\ 2 & 2 & 1 & 3 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 2 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 2 & 1 & 1 & 5 & 2 & 1 & 1 & 3 \\ 0 & 1 & 0 & 1 & 1 & 2 & 4 & 2 & 2 & 2 \\ 0 & 1 & 1 & 1 & 0 & 1 & 2 & 4 & 3 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 2 & 3 & 3 & 1 \\ 0 & 0 & 0 & 0 & 1 & 3 & 2 & 1 & 1 & 3 \end{pmatrix}$$

Figure 1.8: Multiplying the "paths of length 1" matrix by the "paths of length 2" matrix.

1.4 Non-Square Matrices

It is possible to find the product of two matrices that are not square. In general, we can multiply an $r \times s$ matrix by an $s \times t$ matrix in exactly the same way as we'd multiply together two square matrices: An example is shown below.

Practice 2: Multiplying non-square matrices
Verify the matrix product shown below. Please make sure that you understand how the product is arrived at before moving on.

$$\begin{pmatrix} 1 & 2 \\ -1 & 3 \\ 2 & -1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & -1 & 1 \\ 4 & 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 10 & 6 & 3 & 3 \\ 10 & 9 & 7 & 2 \\ 0 & -3 & -4 & 1 \end{pmatrix}$$

Note that in order for this product to be well-defined, the number of columns in the matrix on the left must equal the number of rows in the matrix on the right. This is the value s in the preceding paragraph.

Chapter 2

Implementing in CUDA

We now turn to the subject of implementing matrix multiplication on a CUDA-enabled graphics card. We will begin with a description of programming in CUDA, then implement matrix multiplication, and then implement it in such a way that we take advantage of the faster "shared memory" on the GPU.

2.1 The CUDA Programming Model

We will use CUDA, NVIDIA's *Compute Unified Device Architecture* computing model, for the examples in this module. CUDA and the nvcc compiler enable the user to write programs in C which will run on an ordinary host computer (Windows, Linux, etc...) with the additional capability of running SIMT code on an attached CUDA-enabled GPU device. (In CUDA parlance, the terms "host" and "device" refer to the ordinary computer and graphic card respectively. The term SIMT means *single instruction, multiple thread* and refers to a model wherein many threads running in parallel all execute the same instruction at the same time, but on data specific to that thread.) We refer the reader to the NVIDIA CUDA C Programming Guide [2] for a detailed overview of the CUDA programming model and hardware. (Version 4.0, dated 5/6/11, is packaged with this module.) Here, we will introduce specifics about programming with CUDA *as we need them*, rather than all at once up front. For readers who like to see it all up front, please see the Programming Guide, sections 2.1 to 3.2.

2.1.1 Threads, Blocks and Grids

Threads in CUDA each have their own program counter and registers. All threads share a memory address space called "global memory,", and threads within the same block share access to a very fast "shared memory" that is more limited in size. Within the same block threads share the instruction stream and execute instructions in parallel. When thread execution diverges, then the different branches of execution are run serially, until the divergent section has completed, at which point all threads in the block execute in parallel again, until the next divergence within that block. CUDA devices run many threads simultaneously. For example, the NVIDIA Tesla C2075 has 14 multiprocessors, each of which has 32 cores, so that 448 threads may be running simultaneously, while the NVIDIA GT 330M (the card in my laptop) has 6 multiprocessors with 8 cores each, letting 48 threads run simultaneously. We will multiply our matrices by making each thread responsible for computing a single entry in the product matrix. Thus if we are multiplying a 100×200 matrix by a 200×500 matrix, we would be launching 50,000 threads total to compute the entries of the 100×500 product matrix.

Threads running under CUDA must be grouped into blocks, and a block can hold at most 512 or 1024 threads. We would thus have to launch multiple blocks to compute the product in the previous example. The phrase *compute capability* is the term NVIDIA uses to describe the general computing power of its GPUs. (See Appendix F of [2] for more about compute capabilities.) For devices of compute capability below 2.0, a block may have at most 512 threads. For 2.0 and above, it's 1024.Threads within the same block have access to very fast "shared memory" which is shared the way threads of a process share memory on an ordinary computer, while threads in different blocks must communicate via off-chip "global memory," which is much slower.

Blocks of threads may be one-, two- or three-dimensional, as the programmer prefers. Since matrices are two-dimensional, we will use two-dimensional blocks. And since we want our code to run on devices of all compute capabilities, we will use blocks of size 16×32 so that they contain 512 threads. All blocks must have the same dimensions as all other blocks, and we must launch enough blocks to cover all entries of the product matrix. Therefore if our matrix's dimensions are not both multiples of 16, then some threads will not be computing elements of the product matrix. Figure 2.1 shows the blocks used to compute a product matrix of size 70×200.

Note that even though all 512 threads in every block will run, the first thing our threads will do is locate themselves relative to the matrix, and those threads that lie outside the matrix will immediately terminate.

The blocks themselves form a two-dimensional *grid* of blocks. In the example in Figure 2.1 we have a 5×7 grid of thread blocks. When we launch our computation on the GPU, we specify the dimensions of the grid, and the dimensions of the blocks. See Section 2.2 of [2] for more on

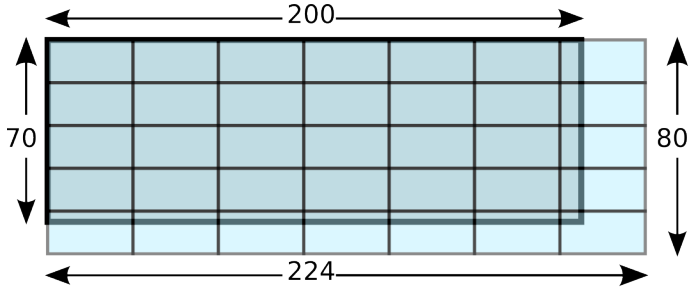


Figure 2.1: A 5×7 grid of 16×32 blocks covering a 70×200 product matrix.

the thread hierarchy.

2.1.2 Thread Location: Dimension and Index

Every running thread has access to several structures that help it to discover its location within the grid and blocks. These are described in Section B.4 of [2].

gridDim	The dimensions of the grid. Use gridDim.x, gridDim.y and gridDim.z to access the dimensions. In the example of Figure 2.1, gridDim.x = 7, gridDim.y = 5 and gridDim.z = 0.
blockIdx	The location of a block within the grid. All 512 threads in the middle block of the grid would find blockIdx.x = 3, blockIdx.y = 2, blockIdx.z = 0, if they queried this structure.
blockDim	The dimensions of the blocks. All threads in the example would find blockDim.x = 32, blockDim.y = 16 and blockDim.z = 0.
threadIdx	This structure gives the location of a thread within its own block. Thus the top-left thread of each block would find threadIdx.x = threadIdx.y = threadIdx.z = 0, while the top-right threads would find threadIdx.x = 31, threadIdx.y = 0 and threadIdx.z = 0.

2.1.3 Kernels and Multiprocessors

A *kernel* is the unit of work that the main program running on the host computer offloads to the GPU for computation on that device. In CUDA, launching a kernel requires specifying three things:

- The dimensions of the grid

- The dimensions of the blocks
- The kernel function to run on the device.

Kernel functions are specified by declaring them `__global__` in the code, and a special syntax is used in the code to launch these functions on the GPU, while specifying the block and grid dimensions. These kernel functions serve as the entry points for the GPU computation, much the same way `main()` serves as the entry point in an ordinary C program. In the example below, which comes from the code included with this module, we launch the `global` function `MatMulKernel` on the device, passing parameters `d_A, d_B` and `d_C`. The variables `dimGrid` and `dimBlock` are of type `dim3`, which holds a triple of unsigned integers. These are the variables to which the threads have access as described above.

```
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

When a kernel is launched on the GPU a grid of thread blocks is created and the blocks are queued to be run on the GPU's multiprocessors. In the example of Figure 2.1, 35 blocks would be created. These blocks are given to multiprocessors as they become available, and once started on a multiprocessor, the threads of that block will run to completion on that multiprocessor. The CUDA programming model requires that these blocks be able to compute in any order, that is, the programmer may make no assumptions about the order in which the GPU schedules and runs the blocks of threads. The program must run correctly regardless of the order in which the blocks are scheduled. This restriction may make programming the kernel a bit more difficult, but the benefit is that more parallelism is achieved, and the program will run without modification on cards with more multiprocessors, scaling optimally. See Section 2.1 of [2] for more on kernels.

My laptop's video card has six multiprocessors. If the 35 blocks of the previous example were to be scheduled on my laptop's card, a block would be placed on each multiprocessor, and as the threads of each block terminated, a new block would be placed on that multiprocessor for computation, and so on, until all 35 blocks had been processed. The host code could then be informed that the GPU computation is done, and the host can read the product off of the card.

Actually, the picture is a bit better than just described. Suppose that the threads of some block ask for data from the device's global memory. There would ensue a relatively long wait as that request is filled, during which time the multiprocessor would sit idle. To minimize processor idle time the CUDA scheduler may put a second block on the same multiprocessor and run the threads of that second block as long as they are ready to run. With two blocks resident on a single multiprocessor, it is possible to keep the processors busy by switching to the other block

whenever the threads of one block have to wait. Under CUDA, it is possible to have up to eight blocks resident on a multiprocessor at any given time, as long as that multiprocessor has enough registers and shared memory to hold those blocks. (Memory is discussed below.) GPU multiprocessors are designed with very large numbers of registers (each of my multiprocessors, for example, has 16,384 32-bit registers) so that context switching, that is, switching from one block to another, takes, literally, no time at all.

2.1.4 An Implementation of Matrix Multiplication

We are now in a position to write a kernel that allows host code to offload matrix multiplication to the GPU. The kernel function is shown below — the entire program is shown in Chapter 3.

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Each thread computes one element of C
  // by accumulating results into Cvalue
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row > A.height || col > B.width) return;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e] *
                              B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```

The first line contains the `__global__` keyword declaring that this is an entry-point function for running code on the device. The declaration `float Cvalue = 0` sets aside a register to hold this float value where we will accumulate the product of the row and column entries.

The next two lines help the thread to discover its row and column within the matrix. *It is a good idea to make sure you understand those two lines before moving on.* The `if` statement in the next line terminates the thread if its row or column place it outside the bounds of the product matrix. This will happen only in those blocks that overhang either the right or bottom side of the matrix.

The next three lines loop over the entries of the row of A and the column of B (these have the same size) needed to compute the (row, col)-entry of the product, and the sum of these products

are accumulated in the `Cvalue` variable. Matrices A and B are stored in the device's global memory in *row major* order, meaning that the matrix is stored as a one-dimensional array, with the first row followed by the second row, and so on. Thus to find the index in this linear array of the (i, j)-entry of matrix A, for example, we compute ($i \times$ width of A) to find the starting index of the ith row, and then add j to go to the jth entry in that row. Finally, the last line of the kernel copies this product into the appropriate element of the product matrix C, in the device's global memory.

2.1.5 Quick Outline of the Code in Chapter 3

There are three functions in `multNoShare.c`:

- **main()**
  This function generates two random matrices with dimensions that are read from the commandline. Matrix A is filled randomly with numbers from {0, 1, 2}, and Matrix B is filled from {0, 1}. Then `main()` calls MatMul() which multiplies these and places the product in Matrix C. The included Makefile generates an executable called `multNoShare`, which may be used as follows: `multNoShare 10 15 8` will generate two matrices. The first, matrix A, will be of dimensions 10×15 and the second, matrix B, will have dimensions 15×8. The 10×8 product matrix will be computed, and then all three matrices will be printed out, up to the first 10 rows and columns.
- **MatMul(Matrix A, Matrix B, Matrix C)**
  This function takes two matrices A and B as input, and fills the entries of matrix C with the product. It first allocates memory on the device for matrix A, and copies A onto the device's global memory. Then it does the same for matrix B. It allocates space on the device for C, the matrix product, computes the number of blocks needed to cover the product matrix, and then launches a kernel with that many blocks. When the kernel is done, it reads matrix C off of the device, and frees the global memory.
- **MatMulKernel(Matrix A, Matrix B, Matrix C)**
  As described above, this runs on the device and computes the product matrix. It assumes that A and B are already in the device's global memory, and places the product in the device's global memory, so that the host can read it from there.

As we will see below, and as discussed in Section 3.2.2.2 of [1] and Section 3.2.3 of [2], the code just given works and is fast, but could be so much faster if we take advantage of *shared memory*.

2.2 The Memory Hierarchy

As mentioned above, multiprocessors have a large number of 32-bit registers: 8k for devices of compute capabilities 1.0 and 1.1, 16k for devices of compute capability 1.2 and 1.3, and 32k for devices of compute capability 2.0 or above. (See Appendix F of [2].) Here we describe the various kinds of memory available on a GPU.

Registers	Registers are the fastest memory, accessible without any latency on each clock cycle, just as on a regular CPU. A thread's registers cannot be shared with other threads.
Shared Memory	Shared memory is comparable to L1 cache memory on a regular CPU. It resides close to the multiprocessor, and has very short access times. Shared memory is shared among all the threads of a given block. Section 3.2.2 of the Cuda C Best Practices Guide [1] has more on shared memory optimization considerations.
Global Memory	Global memory resides on the device, but off-chip from the multiprocessors, so that access times to global memory can be 100 times greater than to shared memory. All threads in the kernel have access to all data in global memory.
Local Memory	Thread-specific memory stored where global memory is stored. Variables are stored in a thread's local memory if the compiler decides that there are not enough registers to hold the thread's data. This memory is slow, even though it's called "local."
Constant Memory	64k of Constant memory resides off-chip from the multiprocessors, and is read-only. The host code writes to the device's constant memory before launching the kernel, and the kernel may then read this memory. Constant memory access is cached — each multiprocessor can cache up to 8k of constant memory, so that subsequent reads from constant memory can be very fast. All threads have access to constant memory.
Texture Memory	Specialized memory for surface texture mapping, not discussed in this module.

2.2.1 Matrix Multiplication with Shared Memory

In light of the memory hierarchy described above, let us see what optimizations we might consider. Looking at the loop in the kernel code, we notice that each thread loads $(2 \times \text{A.width})$ elements

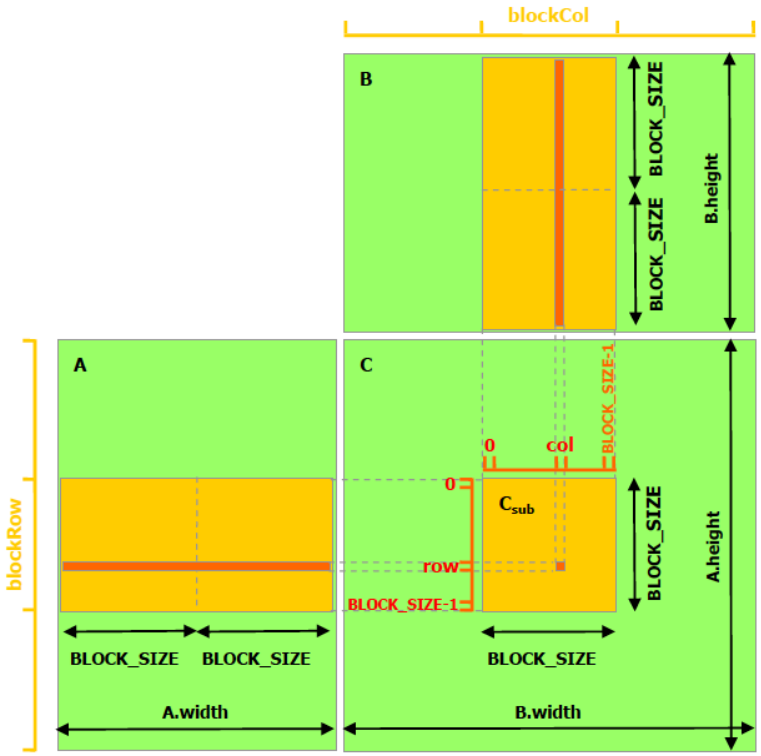


Figure 2.2: Multiplying two matrices using shared memory to hold submatrices of A and B. *From the NVIDIA CUDA C Programming Guide.*

from global memory — two for each iteration through the loop, one from matrix A and one from matrix B. Since accesses to global memory are relatively slow, this can bog down the kernel, leaving the threads idle for hundreds of clock cycles, for each access.

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
                        ⋮
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e] *
                               B.elements[e * B.width + col];
                        ⋮
}
```

One way to reduce the number of accesses to global memory is to have the threads load portions of matrices A and B into shared memory, where we can access them much more quickly. Ideally,

we would load both matrices entirely into shared memory, but unfortunately, shared memory is a rather scarce resource, and won't hold two large matrices. Devices of compute capability 1.x have 16kB of shared memory per multiprocessor, and devices of compute capability 2.x have 48kB. (See Appendix F of The CUDA C Programming Guide [2] for more stats related to compute capability.) So we will content ourselves with loading portions of A and B into shared memory as needed, and making as much use of them as possible while they are there.

One way of doing this is suggested by Figure 2.2, taken from page 29 of [2]. Matrix A is shown on the left and matrix B is shown at the top, with matrix C, their product, on the bottom-right. This is a nice way to lay out the matrices visually, since each element of C is the product of the row to its left in A, and the column above it in B. The row and column sort of "aim" for their product in C. In that figure, and in our sample code, we will use square thread blocks of dimension `BLOCK_SIZE` × `BLOCK_SIZE` and will assume that the dimensions of A and B are all multiples of `BLOCK_SIZE`.

Again, each thread will be responsible for computing one element of the product matrix C. For reference, consider the item highlighted in red in the matrix C, in Figure 2.2. (Note that the yellow square in matrix C represents one thread-block's-worth of elements, whereas the red box inside the yellow square represents a single entry in C, and hence a single thread.) Our thread computes this element in C by multiplying together the red row shown in A, and the red column shown in B, but it will do it in pieces, as we will now discuss.

We may decompose matrices A and B into non-overlapping submatrices of size `BLOCK_SIZE`× `BLOCK_SIZE`. If we look at our red row and red column, they will pass through the same number of these submatrices, since they are of equal length. If we load the left-most of those submatrices of matrix A into shared memory, and the top-most of those submatrices of matrix B into shared memory, then we can compute the first `BLOCK_SIZE` products and add them together just by reading the shared memory. But here is the benefit: as long as we have those submatrices in shared memory, every thread in our thread block (computing the `BLOCK_SIZE`×`BLOCK_SIZE` submatrix of C) can compute that portion of their sum as well from the same data in shared memory.

When each thread has computed this sum, we can load the next `BLOCK_SIZE`× `BLOCK_SIZE` submatrices from A and B, and continue adding the term-by-term products to our value in C. And after all of the submatrices have been processed, we will have computed our entries in C. The kernel code for this portion of our program is shown below.

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Block row and column
  int blockRow = blockIdx.y, blockCol = blockIdx.x;

  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

  // Each thread computes 1 element of Csub accumulating results into Cvalue
  float Cvalue = 0.0;

  // Thread row and column within Csub
  int row = threadIdx.y, col = threadIdx.x;

  // Loop over all the sub-matrices of A and B required to compute Csub
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrices Asub of A and Bsub of B
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    Matrix Bsub = GetSubMatrix(B, m, blockCol);

    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);

    __syncthreads();

    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
      Cvalue += As[row][e] * Bs[e][col];

    __syncthreads();
  }

  // Each thread writes one element of Csub to memory
  SetElement(Csub, row, col, Cvalue);
}
```

2.2.2 Worth the Trouble?

The code that takes advantage of shared memory in this way is modestly more complex, and it is reasonable to ask whether the benefits outweigh the costs. A good way to analyze this is to consider a single element of matrix A, and ask how many times it is loaded from global memory. In our first program, which did not take advantage of shared memory, an entry is loaded by each thread that needs it for its computation. A glance at Figure 2.2 (ignoring the blocks for now) shows that we load each entry of A once for each column of B. But in our shared memory code, we load this entry only when we load the block of A containing that entry, and that will happen once for each *block* in the product matrix C in the same row as that entry. That is, it will be loaded (width of B)/(`BLOCK_SIZE`) times. If `BLOCK_SIZE` equals 16, then we load this entry from global memory 1/16th as many times as we do without making use of shared memory. In our timings below, we will show that this translates into real time savings.

2.2.3 Shared Memory, Device Functions and the `syncthreads()` Command

Note the calls to `GetSubMatrix(), GetElement()` and `SetElement`. These are functions that the kernel calls, and which run on the device. Device functions are declared by using the `__device__` keyword, as you can see by examining the code in Chapter 4. There is nothing special about writing a function for the device, as long as you bear in mind that these are functions that will be run by the threads, and so will have access to the built-in variables `threadIdx`, for example.

The `__shared__` keyword is used in our kernel to declare the arrays `As` and `Bs` as shared, requesting that the compiler store them in the multiprocessor's shared memory. It is important that there be enough shared memory available to accommodate at least one block, so let us do a quick computation to see how we're doing. We will set `BLOCK_SIZE` to 16 in our code, so that As and Bs will each hold 256 floats. A float takes 4 bytes, so we have requested $2 \times 256 \times 4$ bytes = 2k of shared memory. Thus with 16k of shared memory available (as on devices of compute capability 1.x) our multiprocessors would have room for eight blocks, which is the maximum allowed. This is very good, since the more blocks that are resident on a single multiprocessor, the more likely that the scheduler will be able to find threads that can run whenever other threads are waiting for memory requests to be serviced.

2.2.4 Calling the Kernel

The kernel in the `MatMulKernel()` kernel given above makes use of the `__syncthreads()` call. Whenever a thread calls `__syncthreads()`, all threads in that thread's block must compute up to this point before any thread is allowed to pass that point. With the first call to `__syncthreads()` we thus insure that every entry of the submatrices of A and B have been loaded into shared memory before any thread begins its computations based on those values. The second call to `__syncthreads()` ensures that every element of the submatrix of C has been processed before we begin loading the next submatrix of A or B into shared memory. Note that while the `__syncthreads()` primitive enables this sort of inter-thread synchronization, its use does minimize parallelism and may degrade performance if not used wisely and sparingly.

Let's take a look at the host code related to invocation of the kernel:

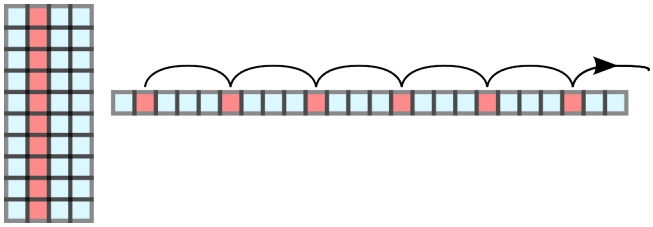
```
void MatMul(const Matrix A, const Matrix B, Matrix C) \{
                      ...
   // Invoke kernel
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
  MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
  err = cudaThreadSynchronize();
  printf("Run kernel: %s\n", cudaGetErrorString(err));
                      ...
\}
```

The `dim3` structure defined in the CUDA libraries is very useful. It defines a triple of `unsigned int`s used to hold the dimensions of a grid or block. Notice that the constructor given has only two arguments. When less than three arguments are given, the missing arguments are initialized to '0'. These two lines define `dimBlock` to be a square with side `BLOCK_SIZE`, and dimGrid to be large enough to cover the entire matrix. (Recall that we are assuming that the height and width of the matrices are multiples of `BLOCK_SIZE`.

Then comes the kernel invocation. `MatMulKernel` is the name of the function, which must have been declared `__global__`, followed by special syntax recognized by the `nvcc` compiler. Between the '`<<<`' and '`>>>`' come two `dim3` structures specifying first the dimensions of the grid, then of the blocks. (If you have one-dimensional blocks or grids you may use an `unsigned int` for these parameters, in which case the compiler initializes the second and third components of the blocks or grid to '0'.) Finishing the invocation line are the parameters to be passed to the kernel function, just as in ordinary C.

2.2.5 An Aside on Stride

It may be worthwhile to say something about the `stride` variable used in this code. As mentioned above, matrices are stored in *row major* order, meaning that if we look in global memory for our matrix we will find it stored in a one-dimensional array consisting of the first row, followed by the second row, followed by the third row, etc... The *stride* refers to the width of the matrix, because that is how big one's steps must be if one wishes to "walk" down a column of the matrix. For example, if we have a 10×4 matrix in memory, then the elements of the first column of the matrix lie 4 apart in memory, as shown in the figure below.



We give matrices a "stride" in this code because for submatrices, the size of a step one must take in memory in order to walk down a column is not the width of *that submatrix*, but rather the width of the parent matrix, which is its stride.

2.3 Compiling and Running the Included Code

The included Makefile compiles two executables. As you experiment with the source code, this Makefile may help you to speed up your work.

- `make multNoShare` compiles an executable called `multNoShare` that runs the matrix multiplication code without shared memory, as presented in Chapter 3. It generates two random matrices as described in Section 2.1.5, computes their product and prints out the three matrices up to the first 10 rows and columns. If you would like to see how long it takes to multiply two 100×100 matrices, you can type `time multNoShare 100 100 100`.
- `make multShare` makes an executable called `multShare` which implements the shared memory version given in Chapter 4. It is used the same way as `multNoShare`.
- `make all` will make both executables.

IMPORTANT NOTE: Many text editors facilitate writing C code by automatically indenting code and highlighting the code by context, and does so by recognizing the "`.c`" extension of the file. Most of these editors do not recognize the "`.cu`" extension. I've therefore been writing the

code as "`.c`" files, and then having the `Makefile` copy them to "`.cu`" files before compilation. This is because the `nvcc` compiler requires the "`.cu`" extension. There are other ways to deal with this discrepancy... Pick your own. But be aware that this is the workflow that the included `Makefile` expects.

In the activity below you will be asked to experiment with the programs on matrices of various sizes. Suppose you wish to run the program on matrices whose sizes will exceed the amount of memory available on the graphics card. The program will compile just fine, but at runtime, when your program tries to allocate memory on the graphics card to hold the matrix elements, a memory allocation error will result. And though the program will not crash, the results will be invalid. It is therefore important to check the return value of the memory allocation call:

```
cudaError_t err = cudaMalloc(&d_A.elements, size);
printf("CUDA malloc A: %s",cudaGetErrorString(err));
```

Practice 3: Running the Programs
Let's do a few experiments with the two programs we just made:

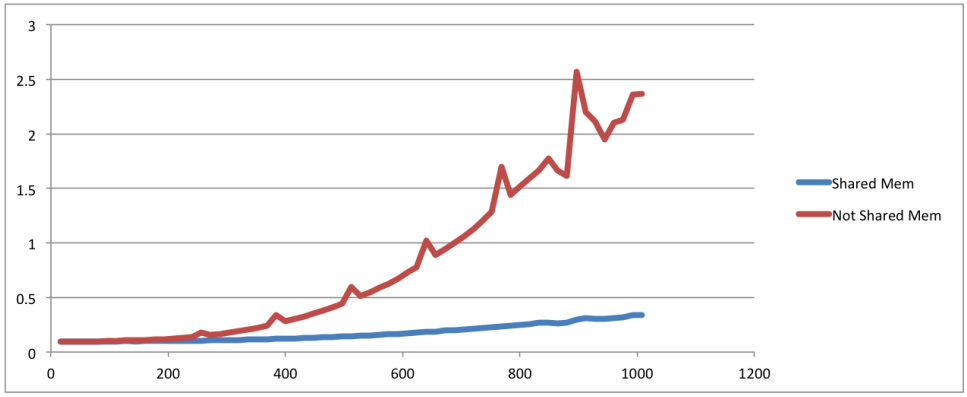
- Run the program to see how large a pair of square matrices you can multiply without getting a warning about memory. For example, to multiply a pair of matrices both of dimensions 10×10 without using shared memory, type `multNoShare 10 10 10`. Does the use of the shared memory version (`multShare`) have any effect on the size of the largest pair you can multiply?
- You can time how long it takes to multiply the matrices by typing "`time multNoShare 10 10 10`". The line that says "real" tells how long it took in actual seconds as experienced by the user — so-called "wall time." Run this for matrices of various sizes to get an idea of how the running time varies with matrix size, and how the versions that do or don't use shared memory compare in terms of "wall time."
- Collate your data into a spreadsheet, and compare your timing results with those in the next section.

2.4 Some Timing Results

We ran both variants of the matrix multiplication code on devices of compute capabilities 1.1, 1.2, 1.3 and 2.0. The results are shown in the following sections.

2.4.1 Capability 1.1 - NVIDIA GeForce GT 9600M in a MacBook Pro Laptop, 4 multiprocessors, 32 cores

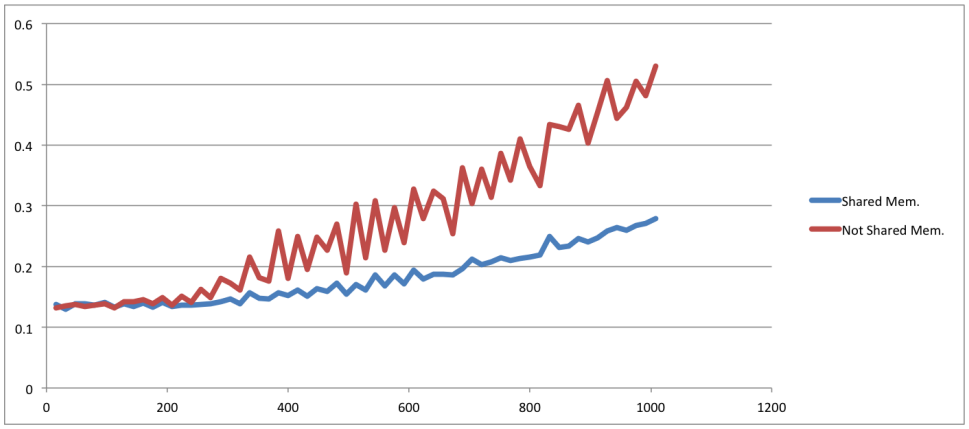
The difference in running times is dramatic on devices of compute capability 1.1. By the time we get to matrices of size 1000×1000, we have a factor of seven improvement in the running time, and the ratio seems to be increasing as matrix size increases.



Running time vs. matrix size.

2.4.2 Capability 1.2 - NVIDIA GeForce GT 330M in a MacBook Pro Laptop, 6 multiprocessors, 48 cores

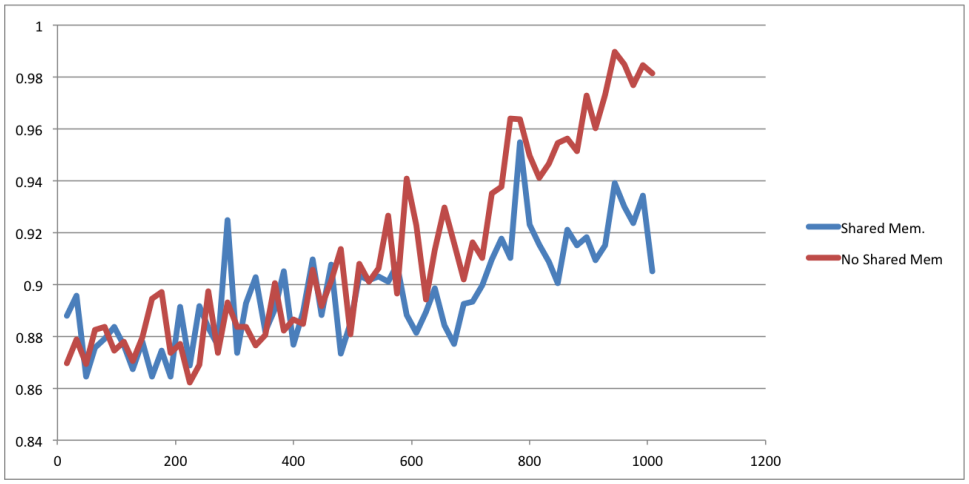
On this device the ratio approaches 2 as matrix size approaches 1000. The improvement in running time for the non-shared memory version over the timings on the 1.1 device can be accounted for by the 50% increase in multiprocessors. But the dramatic improvement in the running times for the shared memory version likely has more to do with how devices of compute capability 1.2 coalesce memory accesses, in a better way than devices of compute capability 1.1 did. See section F.3.2 of the Cuda C Programming Guide [2].



Running time vs. matrix size.

2.4.3 Capability 1.3 - NVIDIA Tesla C1060 running in Earlham's cluster, 30 multiprocessors, 240 cores

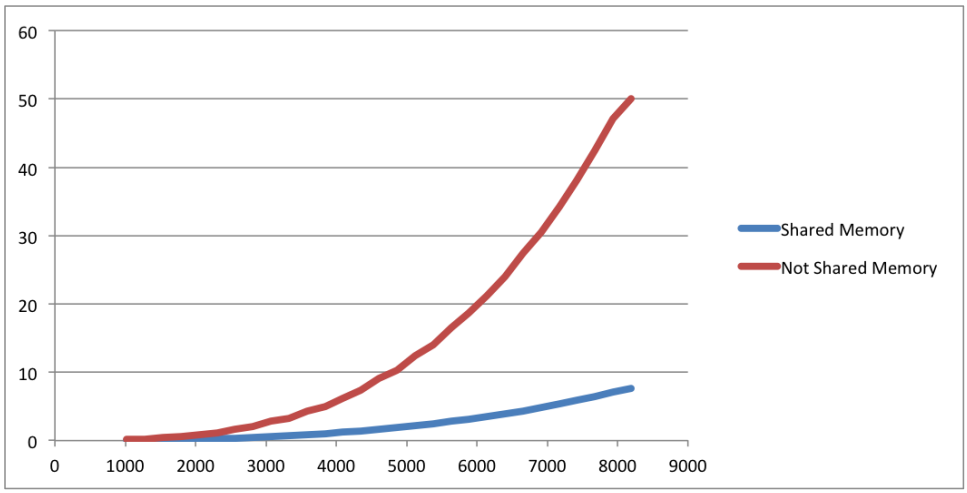
On this device, we see practically no difference in running time between the two versions of the program. Note that the vertical axis starts at 0.84, not at 0. So there is only about a 10% difference in running time.



Running time vs. matrix size.

But those were small matrices! If we re-run the comparison for large matrices, of sizes 1000 to 8000, we see a clearer pattern of runtime differences. In the figure below, we obtain a factor of greater than 6.5 in the ratio of the running times as the size approaches 8000. This clearly

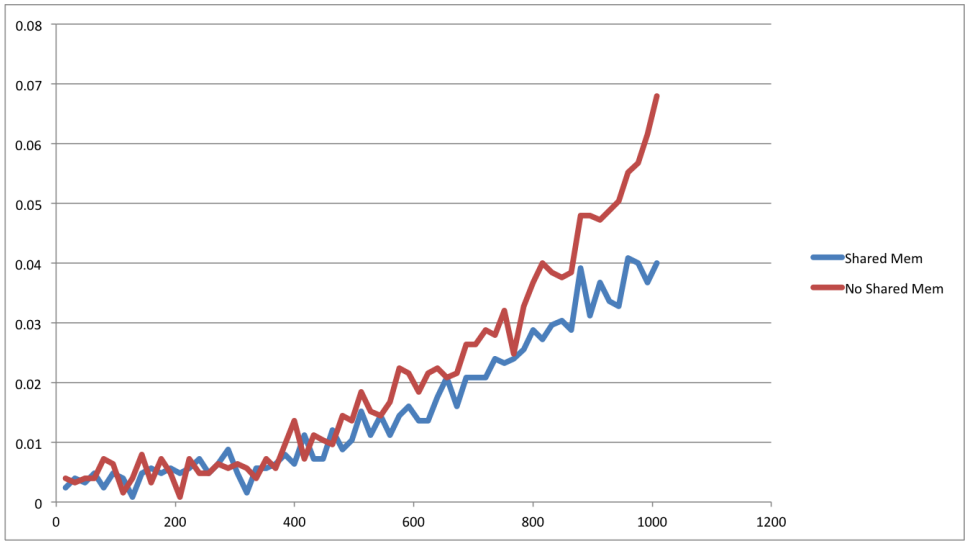
illustrates the importance of taking advantage of shared memory when possible.



Running time vs. matrix size.

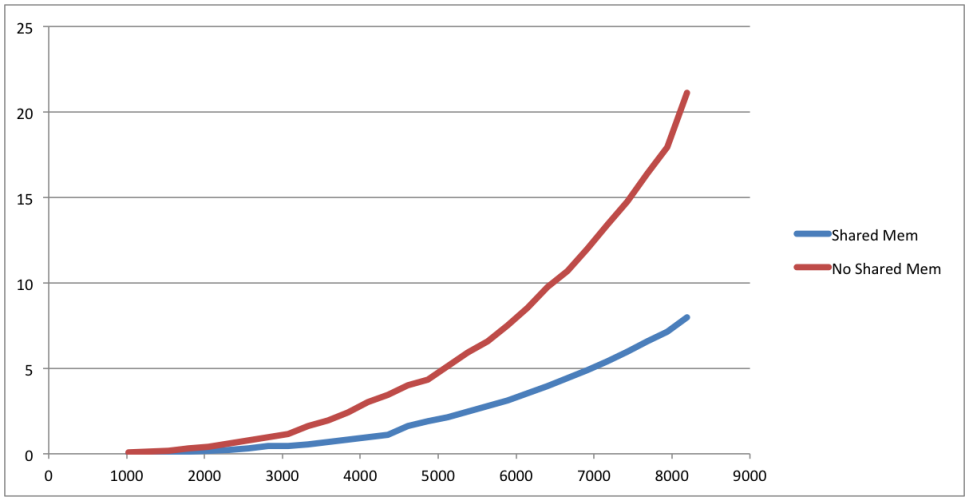
2.4.4 Capability 2.0 - NVIDIA Tesla M2070 at the Texas Advanced Computing Center, 14 multiprocessors, 448 cores

On this device of compute capability 2.0, we approach a ratio of about 1.7 by the time the matrices are 1000×1000. In this cluster, the NVIDIA cards are located in a module off-site from the motherboard, so that memory transfers between the host and device will take longer than on a motherboard with the card resident in a PCI slot. That accounts for the overall longer running time, even though the TACC device itself is much faster.



Running time vs. matrix size.

As the size of the matrices continues to grow, the benefit obtained by using shared memory increases, as shown in the figure below. For matrices of size 8000×8000, the shared memory version is faster by a factor of 2.6.



Running time vs. matrix size.

2.5 Exploration Questions

1. **Test of Concepts:** Here is the matrix showing the number of paths of length 4 between the pairs of vertices of a graph:

$$\begin{pmatrix} 9 & 3 & 6 & 11 & 1 \\ 3 & 15 & 8 & 7 & 11 \\ 6 & 8 & 8 & 8 & 6 \\ 11 & 7 & 8 & 15 & 3 \\ 1 & 11 & 6 & 3 & 9 \end{pmatrix}$$

   - How many vertices does the graph have?
   - Assume that the vertices of the graph are labeled A, B, C, ... and that the rows and columns are given alphabetically. Show that there are 208 paths of length 8 from A to B in that graph.

2. **Largest Matrices:** The program 'deviceQuery' is included with the CUDA SDK. It gives you specifications on the CUDA-enabled card(s) installed in your system. Run this program to find out how much RAM your card has, and use this to determine the theoretical largest value N for which you can multiply two $N \times N$ matrices on your card, and hold the product. Then compare this theoretical value with the actual largest N for which `multShare N N N` returns with no errors.

3. **Test of CUDA Acuity:** Modify the code for the shared memory version of the program so that it works even if the dimensions of the matrices are not multiples of the size of the thread blocks. Use the code for the no-shared memory version as a model.

4. **Explorations with Pascal's Triangle** The matrix on the left below contains the entries of Pascal's Triangle. The left column and the main diagonal consist of ones, all entries above the main diagonal are zeros, and each other entry is the sum of the entry directly above it and the entry diagonally above and to the left. The matrix on the right is the same, except that the diagonals are alternately positive and negative.

   - What is the product of the two matrices given here?
   - Modify `multShare.c` so that it generates larger versions of these two matrices and verify that the program produces the correct product.
   - How large can your matrices get while still using integers?
   - Modify the code to use doubles, so that you can multiply larger matrices. Now how large can your matrices be and still produce the correct result?

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 3 & 1 & 0 & 0 & 0 & 0 \\ 1 & 4 & 6 & 4 & 1 & 0 & 0 & 0 \\ 1 & 5 & 10 & 10 & 5 & 1 & 0 & 0 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 & 0 \\ 1 & 7 & 21 & 35 & 35 & 21 & 7 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -3 & 1 & 0 & 0 & 0 & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & 0 \\ -1 & 5 & -10 & 10 & -5 & 1 & 0 & 0 \\ 1 & -6 & 15 & -20 & 15 & -6 & 1 & 0 \\ -1 & 7 & -21 & 35 & -35 & 21 & -7 & 1 \end{pmatrix}$$

Chapter 3

Code for Simple Matrix Multiplication

3.1 multNoShare.h

```
/*
 * multNoShare.h
 *
 * Robert Hochberg
 * January 24, 2012
 *
 * Based nearly entirely on the code from the CUDA C Programming Guide
 */

#include <stdio.h>

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
  int width;
  int height;
  float* elements;
} Matrix;
```

```
// Thread block size
#define BLOCK_SIZE 16

__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

3.2 multNoShare.c

```
/*
 * multNoShare.c
 *
 * Robert Hochberg
 * January 24, 2012
 *
 * Based nearly entirely on the code from the CUDA C Programming Guide
 */

#include "multNoShare.h"

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C) {

  // Load A and B to device memory
  Matrix d_A;
  d_A.width = A.width;
  d_A.height = A.height;
  size_t size = A.width * A.height * sizeof(float);
  cudaError_t err = cudaMalloc(&d_A.elements, size);
  printf("CUDA malloc A: %s\n",cudaGetErrorString(err));
  err = cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
  printf("Copy A to device: %s\n",cudaGetErrorString(err));

  Matrix d_B;
  d_B.width = B.width;
  d_B.height = B.height;
  size = B.width * B.height * sizeof(float);
  err = cudaMalloc(&d_B.elements, size);
```

```
  printf("CUDA malloc B: %s\n",cudaGetErrorString(err));
  err = cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);
  printf("Copy B to device: %s\n",cudaGetErrorString(err));

  // Allocate C in device memory
  Matrix d_C;
  d_C.width = C.width;
  d_C.height = C.height;
  size = C.width * C.height * sizeof(float);
  err = cudaMalloc(&d_C.elements, size);
  printf("CUDA malloc C: %s\n",cudaGetErrorString(err));

  // Invoke kernel
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid((B.width + dimBlock.x - 1) / dimBlock.x,
       (A.height + dimBlock.y - 1) / dimBlock.y);
  MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
  err = cudaThreadSynchronize();
  printf("Run kernel: %s\n", cudaGetErrorString(err));

  // Read C from device memory
  err = cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
  printf("Copy C off of device: %s\n",cudaGetErrorString(err));

  // Free device memory
  cudaFree(d_A.elements);
  cudaFree(d_B.elements);
  // cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Each thread computes one element of C
  // by accumulating results into Cvalue
  float Cvalue = 0.0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row > A.height || col > B.width) return;
```

```
  for (int e = 0; e < A.width; ++e)
    Cvalue += (A.elements[row * A.width + e]) * (B.elements[e * B.width + col]);
  C.elements[row * C.width + col] = Cvalue;
}

// Usage: multNoShare a1 a2 b2
int main(int argc, char* argv[]){
  Matrix A, B, C;
  int a1, a2, b1, b2;
  // Read some values from the commandline
  a1 = atoi(argv[1]); /* Height of A */
  a2 = atoi(argv[2]); /* Width  of A */
  b1 = a2;           /* Height of B */
  b2 = atoi(argv[3]); /* Width  of B */

  A.height = a1;
  A.width = a2;
  A.elements = (float*)malloc(A.width * A.height * sizeof(float));

  B.height = b1;
  B.width = b2;
  B.elements = (float*)malloc(B.width * B.height * sizeof(float));

  C.height = A.height;
  C.width = B.width;
  C.elements = (float*)malloc(C.width * C.height * sizeof(float));

  for(int i = 0; i < A.height; i++)
    for(int j = 0; j < A.width; j++)
      A.elements[i*A.width + j] = (float)(arc4random() % 3);

  for(int i = 0; i < B.height; i++)
    for(int j = 0; j < B.width; j++)
      B.elements[i*B.width + j] = (float)(arc4random() % 2);

  MatMul(A, B, C);

  // Print up to a 10x10 portion of the three matrices
```

```
  for(int i = 0; i < min(10, A.height); i++){
    for(int j = 0; j < min(10, A.width); j++)
      printf("%f ", A.elements[i*A.width + j]);
    printf("\n");
  }
  printf("\n");

  for(int i = 0; i < min(10, B.height); i++){
    for(int j = 0; j < min(10, B.width); j++)
      printf("%f ", B.elements[i*B.width + j]);
    printf("\n");
  }
  printf("\n");

  for(int i = 0; i < min(10, C.height); i++){
    for(int j = 0; j < min(10, C.width); j++)
      printf("%f ", C.elements[i*C.width + j]);
    printf("\n");
  }
  printf("\n");
}
```

Chapter 4

Code for Matrix Multiplication using Shared Memory

4.1 multShare.h

```
/*
 * multShare.h
 *
 * Robert Hochberg
 * January 24, 2012
 *
 * Based nearly entirely on the code from the CUDA C Programming Guide
 */

#include <stdio.h>

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
  int width;
  int height;
  float* elements;
  int stride;
} Matrix;
```

```
// Thread block size
#define BLOCK_SIZE 16

__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

4.2 multShare.c

```
//*
 * multShare.c
 *
 * Robert Hochberg
 * January 24, 2012
 *
 * Based nearly entirely on the code from the CUDA C Programming Guide
 */

#include "multShare.h"

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C) {
  // Load A and B to device memory
  Matrix d_A;
  d_A.width = d_A.stride = A.width;
  d_A.height = A.height;
  size_t size = A.width * A.height * sizeof(float);
  cudaError_t err = cudaMalloc(&d_A.elements, size);
  printf("CUDA malloc A: %s\n",cudaGetErrorString(err));
  cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

  Matrix d_B;
  d_B.width = d_B.stride = B.width;
  d_B.height = B.height;
  size = B.width * B.height * sizeof(float);
  err = cudaMalloc(&d_B.elements, size);
  printf("CUDA malloc B: %s\n",cudaGetErrorString(err));
```

```
  cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

  // Allocate C in device memory
  Matrix d_C;
  d_C.width = d_C.stride = C.width;
  d_C.height = C.height;
  size = C.width * C.height * sizeof(float);
  err = cudaMalloc(&d_C.elements, size);
  printf("CUDA malloc C: %s\n",cudaGetErrorString(err));

  // Invoke kernel
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    err = cudaThreadSynchronize();
    printf("Run kernel: %s\n", cudaGetErrorString(err));

  // Read C from device memory
  err = cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
  printf("Copy C off of device: %s\n",cudaGetErrorString(err));

  // Free device memory
  cudaFree(d_A.elements);
  cudaFree(d_B.elements);
  cudaFree(d_C.elements);
}

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col) {
  return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value) {
  A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
```

```
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
  Matrix Asub;
  Asub.width = BLOCK_SIZE;
  Asub.height = BLOCK_SIZE;
  Asub.stride = A.stride;
  Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
  return Asub;
}


// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0.0;

  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;

  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);

    // Get sub-matrix Bsub of B
```

```
    Matrix Bsub = GetSubMatrix(B, m, blockCol);

    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);

    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();

    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
      Cvalue += As[row][e] * Bs[e][col];

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
  }

  // Write Csub to device memory
  // Each thread writes one element
  SetElement(Csub, row, col, Cvalue);
}



int main(int argc, char* argv[]){
  Matrix A, B, C;
  int a1, a2, b1, b2;
  a1 = atoi(argv[1]); /* Height of A */
  a2 = atoi(argv[2]); /* Width  of A */
  b1 = a2;           /* Height of B */
```

```
b2 = atoi(argv[3]); /* Width  of B */

A.height = a1;
A.width = a2;
A.elements = (float*)malloc(A.width * A.height * sizeof(float));

B.height = b1;
B.width = b2;
B.elements = (float*)malloc(B.width * B.height * sizeof(float));

C.height = A.height;
C.width = B.width;
C.elements = (float*)malloc(C.width * C.height * sizeof(float));

for(int i = 0; i < A.height; i++)
  for(int j = 0; j < A.width; j++)
    A.elements[i*A.width + j] = (arc4random() % 3);

for(int i = 0; i < B.height; i++)
  for(int j = 0; j < B.width; j++)
    B.elements[i*B.width + j] = (arc4random() % 2);

MatMul(A, B, C);

for(int i = 0; i < min(10, A.height); i++){
  for(int j = 0; j < min(10, A.width); j++)
    printf("%f ", A.elements[i*A.width + j]);
  printf("\n");
}
printf("\n");

for(int i = 0; i < min(10, B.height); i++){
  for(int j = 0; j < min(10, B.width); j++)
    printf("%f ", B.elements[i*B.width + j]);
  printf("\n");
}
printf("\n");
```

```
  for(int i = 0; i < min(10, C.height); i++){
    for(int j = 0; j < min(10, C.width); j++)
      printf("%f ", C.elements[i*C.width + j]);
    printf("\n");
  }
  printf("\n");
}
```

Bibliography

[1] The NVIDIA Corporation. *The CUDA C Best Practices Guide v4.0*. NVIDIA Corporation, 2011.

[2] The NVIDIA Corporation. *The CUDA C Programming Guide v4.0*. NVIDIA Corporation, 2011.

[3] Shlomo Havlin and Reuven Cohen. *Complex Networks: Structure, Robustness, Function*. Cambridge University Press, 2010.

[4] Peter S. Kutchukian, David Lou, and Eugene I. Shakhnovich. Fog: Fragment optimized growth algorithm for the de novo generation of molecules occupying druglike chemical space. *Journal of Chemical Information and Modeling*, 49(7):1630–1642, 2009. PMID: 19527020.

[5] P. H. LESLIE. On the use of matrices in certain population mathematics. *Biometrika*, 33(3):183–212, 1945.

[6] Qiang Meng, Der-Horng Lee, and Ruey Long Cheu. Counting the different efficient paths for transportation networks and its applications. *Journal of Advanced Transportation*, 39(2):193 – 220, 2005.