Dynamic Programming with CUDA — Part I

Robert Hochberg

August 21, 2012

Contents

Chapter 1

Content

1.1 Overview

This module provides a quick review of dynamic programming, but the student is assumed to have seen it before. The parallel programming environment is NVIDIA's CUDA environment for graphics cards (GPGPU - general purpose graphics processing units). The CUDA environment simultaneously operates with a fast shared memory and a much slower global memory, and thus has aspects of shared-memory parallel computing and distributed computing. Specifics for programming in CUDA are included where appropriate, but the reader is also referred to the NVIDIA CUDA C Programming Guide [5], and the CUDA API Reference Manual [3].

1.2 Dynamic Programming Review

Dynamic programming describes a broad class of problem-solving algorithms, typically involving optimization of some sort. What characterizes a problem suitable for dynamic programming is that solutions to these problem instances can be constructed from solutions to smaller problem instances, where these smaller problem instances are sub-problems of the original problem. A classic example is that of finding the length of a shortest path in a directed graph that has no cycles, so let us begin with Figure 1.1.

Suppose we want to find the length of the shortest path from A to B on this graph,

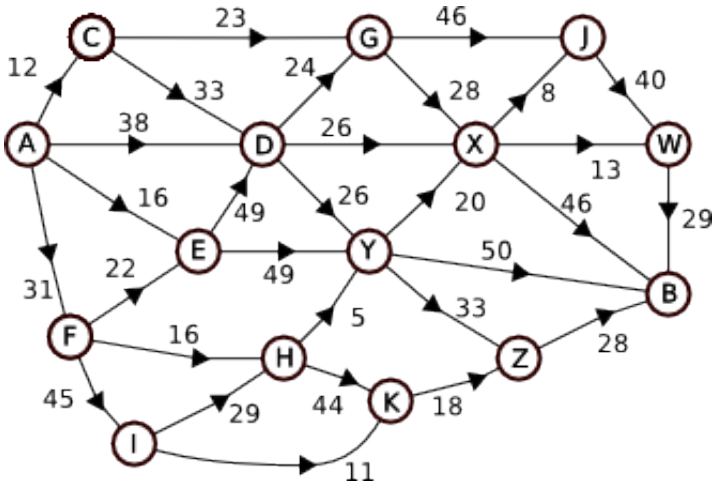


Figure 1.1: An acyclic directed graph with weights on its edges

where the path must respect the directions of the arrows, the numbers represent the length of each of the connecting segments, and the length of a path is the sum of the lengths of its segments. (*We assume in this module that all weights on edges are non-negative integers.*) In the dynamic programming style of thinking we ask ourselves, What smaller problems could we solve, so that their solutions would give us the answer to the original, larger problem? In this case, suppose I told you that the lengths of the shortest paths from vertex A to vertices W, X, Y and Z respectively were 76, 63, 52 and 85. (W, X, Y and Z are called *predecessors* to vertex B. In general, the predecessors of a vertex v are those vertices having an arc from them directly into v.) You could then find the length of the shortest path from A to B. The thinking goes as follows: In order to get to vertex B, your second-to-last step has to be one of the vertices W, X, Y or Z. If W is the second-to-last vertex, then the shortest the path could be is 76 (the shortest distance to W from A) plus 29, the length of the edge from W to B, for a total of 105. If the second-to-last vertex is X, then this value is 63 + 46 = 109. The values for all four possibilities are shown in the table below.

Second-to-last vertex	Distance from A	Distance to B	Total
W	76	29	105
X	63	46	109
Y	52	50	102
Z	85	28	113

Of these four possibilities, the shortest total distance is achieved by passing through

vertex Y on our way to B, and this shortest length is 102. Reasoning in this way, we can discover the length of the shortest path from A to B, that is, we can solve the original problem, *if* we know the solutions to the smaller problems.

Of course, this is helpful only if we have solutions to those smaller problems, that is, if we know the distances to W, X, Y and Z. We find those solutions in the same way. For example, to find the length of the shortest path to W, we would want to find the lengths of the shortest paths to J and X. The table below summarizes the smaller problems that must be solved to find the shortest paths to each of W, X, Y and Z.

To vertex	Smaller problems that need to be solved
W	J, X
X	G, D, Y
Y	D, E, H
Z	Y, K

And here we come to one of the main techniques of dynamic programming: The table seems to indicate that we would need to solve 10 smaller problems in order to find all the distances to W, X, Y and Z. But this table has repeated entries. In fact, we need to solve only six smaller problems. The presence of *overlapping sub-problems* is another characteristic of problems amenable to solution by dynamic programming.

A complete dynamic programming solution to this shortest path problem could be completed in two ways:

1. **Recursively.** We could write a function `shortestPath(vertex` v`)` as a recursive function that calls `shortestPath(vertex` w`)` for each predecessor w of v, add the lengths of the last steps and select the minimum sum, as we did above. Each time we compute a shortest path to some vertex v, we store the length of that path in a lookup table so that we never have to compute that value again. (Thus a function call `shortestPath(`v`)` actually first looks up v in its table, and if that value has already been computed, simply returns it. This process, called *memoization*, is one of the main ingredients of dynamic programming.)

2. **Iteratively.** We build the table in such an order that whenever we wish to compute the shortest path to some vertex, we have already computed the distances to its predecessors. In our example, we might process the vertices in this order: A, C, F, E, D, ..., avoiding the need to make recursive calls.

Spot Check 1 *Finish the ordering of the vertices A, C, F, E, D, ... given above, so that for every vertex in the list, each of its predecessors appear earlier in the list.* Answers are at the back of the module.

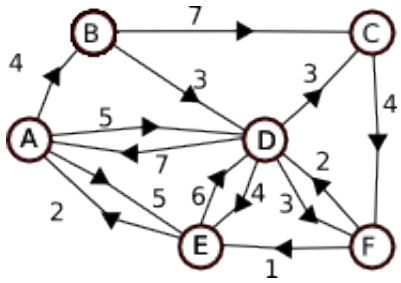
1.3 The Floyd-Warshall Algorithm

In this module we will focus on the the Floyd-Warshall algorithm, also called the *all pairs shortest path* algorithm. In its most basic form, it provides an $O(n^3)$ algorithm (that is, an algorithm whose running time is roughly proportional to the cube of the number of vertices in the graph) which is very easy to program, because it is so easy to understand.

The Floyd-Warshall algorithm is a very widely-used graph algorithm with applications in scheduling, bioinformatics, routing, and many other areas. Any time a problem can be expressed in terms of distances in graphs, the Floyd-Warshall algorithm may provide the quickest way to get an answer. Many academic papers have been written about this algorithm, and recently a few that involve CUDA [[6, 1]]. Here we will give an introduction to the algorithm and a basic implementation that works well for graphs on up to 20,000 vertices or so, on the newer NVIDIA GPU cards. *Our code for this section is written for graphs with integer weights. In this module, we assume all weights are non-negative integers.*

Figure 1.2 shows a directed graph with six vertices. To the right is the starting matrix for the Floyd-Warshall algorithm. Its rows and columns are indexed by the vertices of the graph, and the (v, w) entry of the matrix gives ∞ if there is no arc from v to w, or the weight on the arc if there is one. For convenience, let us assume the vertices are ordered, so that we may speak of "the first vertex" or "the first three vertices." We may interpret this matrix as giving the distances from v to w for all pairs (v, w) of vertices of the graph, *under the assumption that we may use no other vertices along the path.* Thus, while there is a path from F to C of length 5, via the intermediate vertex D, there is no path from F to C that does not use an intermediate vertex, so we call the distance "infinite" to say "you can't get there from here." We denote this matrix M_0.

The algorithm now proceeds as follows. On step 1, we ask whether any of these distances can be improved if we allow the first vertex, and *only* the first vertex (vertex A in our example) as an intermediate vertex along paths. For example, the distance from E to B in M_0 is infinite, but if we allow A as an intermediate vertex, the



M_0	A	B	C	D	E	F
A	0	4	∞	5	5	∞
B	∞	0	7	3	∞	∞
C	∞	∞	0	∞	∞	4
D	7	∞	3	0	4	3
E	2	∞	∞	6	0	∞
F	∞	∞	∞	2	1	0

Figure 1.2: A directed graph and its initial distance matrix

distance drops to 6. Let us denote the distance from v to w in matrix M_0 by $d_0(v, w)$. Then using A as an intermediate vertex gives an improvement in the distance from v to w if and only if $d_0(v, A) + d_0(A, w) < d_0(v, w)$. If we check this inequality for every pair of vertices, then we can build a new matrix M_1 containing the length of the shortest path from v to w, possibly using A as an intermediate vertex. If we call this distance $d_1(v, w)$, then we have

$$d_1(v, w) = \min[\ d_0(v, w), \quad d_0(v, A) + d_0(A, w)\ ]. \tag{1.1}$$

Spot Check 2 *The matrix M_1 is partially computed and shown to the right. Finish constructing this matrix. Note that the finished matrix will have only two entries that differ from the original matrix M_0.*

M_0	A	B	C	D	E	F
A	0	4	∞	5	5	∞
B	∞	0	7	3	∞	∞
C			0			
D	7		3	0	4	3
E	2				0	
F				2	1	0

In general, for $0 \le i \le n$, where n is the number of vertices in the graph, let M_i be the matrix containing the shortest distances $d_i(v, w)$ between pairs of vertices where we allow only the first i vertices to be used as intermediate vertices. (These are the "sub-problems" that we talked about in section 1.2.) Then we may form the matrix M_i from the matrix M_{i-1}, for $i \ge 1$, by observing that

$$d_i(v, w) = \min[\ d_{i-1}(v, w), \quad d_{i-1}(v, A_i) + d_{i-1}(A_i, w)\ ]. \tag{1.2}$$

where A_i is the ith vertex. The algorithm assumes that all entries of matrix M_k are computed before we begin computing M_{k+1}. We won't prove the correctness of the Floyd-Warshall algorithm. Proofs can be found in standard books on algorithms, such as [2].

1.4 Parallel Considerations

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 that 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 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.

A direct parallelization of the Floyd-Warshall algorithm could go as follows: Let us have a processor for each node of the matrix. On the kth step of the algorithm processor (i, j) computes $d_k(i, j)$ from the matrix M_{k-1} using equation (1.2). This parallel implementation would complete in n steps, giving an algorithm with $O(n^2)$ processors and running time $O(n)$.

For the graph in Figure 1.2 this might work as follows: Initialize matrix M_0, create 36 threads (one per entry in the matrix), and on step k, for $1 \leq k \leq 6$, each thread uses equation (1.2) to find its value in matrix M_k. Note that we must synchronize these threads, for it would do no good for the thread computing entry $(0, 2)$ to do all of its updates before the threads for entries $(0, 1)$ and $(1, 2)$ have begun theirs. In our code we will perform this synchronization by launching each iteration in a seperate *kernel*.

1.4.1 CUDA Kernels

A kernel is a unit of work to be performed on a CUDA-enabled video card. A C program running on a host computer sets up this kernel, puts any necessary data onto the video card, and then tells the card to launch the kernel. When it terminates, the user may read from the video card to extract the results of the run. These three lines of code are typical of this process:

```
cudaMemcpy(devArray, graph, 36*sizeof(int), cudaMemcpyHostToDevice);
fwStepK <<< 1, 36 >>> (0, devArray, 6, 6);
cudaMemcpy(graph, devArray, 36*sizeof(int), cudaMemcpyDeviceToHost);
```

The first line copies M_0 onto the device and the third line reads M_1 off of the device. The second line is the kernel invocation. CUDA uses the special syntax `function_name <<< blocksPerGrid, threadsPerBlock >>> (arguments)` which is recognized by the `nvcc` compiler. (Blocks are discussed in the next section.) This function is compiled for the graphics device, with the appropriate hooks added to the compiled host code. The full implementation of the Floyd-Warshall algorithm would look like this:

```
cudaMemcpy(devArray, graph, 36*sizeof(int), cudaMemcpyHostToDevice);
for(int k = 0; k < 6; k++)
   fwStepK <<< 1, 36 >>> (k, devArray, 6, 6);
cudaMemcpy(graph, devArray, 36*sizeof(int), cudaMemcpyDeviceToHost);
```

Notice that we did 6 invocations of the CUDA kernel before reading the result. In general, writing between the host and device should be minimized, since it is a relatively slow process when compared to computing on the device. But invoking a kernel multiple times on a device is fast, as long as the data is already there, as it is in our case.

Spot Check 3 *There are four entries in M_5 which are still ∞, for the graph in Figure 1.2. These correspond to entries for which there is still no path even if you are allowed to use A, B, C, D and E as intermediate vertices. Determine these entries.*
You will have the chance to check your answers below in Spot Check 4 by compiling and running the code included with the module.

Let us take a look at the code for the kernel:

```
__global__ void fwStepK(int k, int devArray[], int Na, int N)
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(col >= N)
    return;

  int arrayIndex = Na * blockIdx.y + col;
  __shared__ int trkc;        /* this row, kth column */

  // Improve by using the intermediate k, if we can
  if(threadIdx.x == 0)
    trkc = devArray[Na * blockIdx.y + k];
  __syncthreads();

  if(trkc == INT_MAX)
    return;
  int tckr = devArray[k*Na + col]; /* this column, kth row */
  if(tckr == INT_MAX)
    return;
  int betterMaybe = trkc + tckr;
  if(betterMaybe < devArray[arrayIndex])
    devArray[arrayIndex] =  betterMaybe;
```

Except for a few lines, this is perfectly ordinary C code, and should be easily understood by any C programmer. We'll take the new-looking lines one at a time.

- `__global__ void fwStepK(int k, int devArray[], int Na, int N)`
  The keyword `__global__` tells the nvcc compiler that this function should be compiled for and run on the CUDA device.

- `int col = blockIdx.x * blockDim.x + threadIdx.x;`
  When the kernel launches, the threads are grouped into blocks, and the blocks form a grid. Each thread gets the same code, so in order for a thread to identify itself, built-in variables, which have unique values for each thread, are read so that the thread can learn where it is and exactly what it should do. Blocks and grids are discussed below.

- `__shared__ int trkc;`
  The `__shared__` keyword tells the compiler that this variable should be stored

in the fast shared memory, and so will be shared by every thread in the block. Without this keyword, each thread would get its own copy of this variable, stored in a register or in the thread's local memory.

- `__syncthreads()`
  Threads within the same block can be synchronized, meaning that all threads in that block are required to reach the `__syncthreads()` statement before any threads are allowed to pass it. A kernel may have any number of these statements. Each will be an independent synchronization point for the threads.

There are probably a few other things that look odd, such as why we use a one-dimensional array to store our matrix when a two-dimensional array might be better, and why we have the variable Na when N is the number of vertices. These have to do with "blocks" of threads and the memory hierarchy, which will be explained in the next section.

1.4.2 Blocks and Grids

Let us examine the `<<<1, 36>>>` notation in the previous example. The "1" tells the device to launch one block of threads, and the "36" tells the device that each block should have 36 threads. Thus we launch 36 threads altogether, one for each matrix entry. Threads running on a device are grouped into *blocks*. Threads in the same block can share the very fast *shared memory* with each other. Figure 1.3 (taken from Section 2.3 of the NVIDIA CUDA C Programming Guide) shows the memory hierarchy involved with CUDA kernels. Threads in different blocks may communicate via *global memory* which, while still on the device, is much slower than shared memory. There is therefore some advantage to launching threads within the same block. CUDA limits block size to 512 or 1024 threads (depending on the device). Additionally, block size is limited by the amount of shared memory available on the device, and the amount needed by each thread.

When the CUDA scheduler sees an available multiprocessor, it looks for a block of threads that needs to run, and puts the entire block onto that multiprocessor. Blocks are automatically grouped into *warps* of size 32, so that a block of size 512 will consist of 16 warps. If a multiprocessor has 8 cores, then the warp is divided into quarter-warps of 8 threads each, and these are run on the cores. If the multiprocessor has 32 cores, then the whole warp can be processed together. When one warp finishes, the next warp in the block may start. If the threads of one warp get blocked, for example, because of a memory read, then the scheduler simply looks for

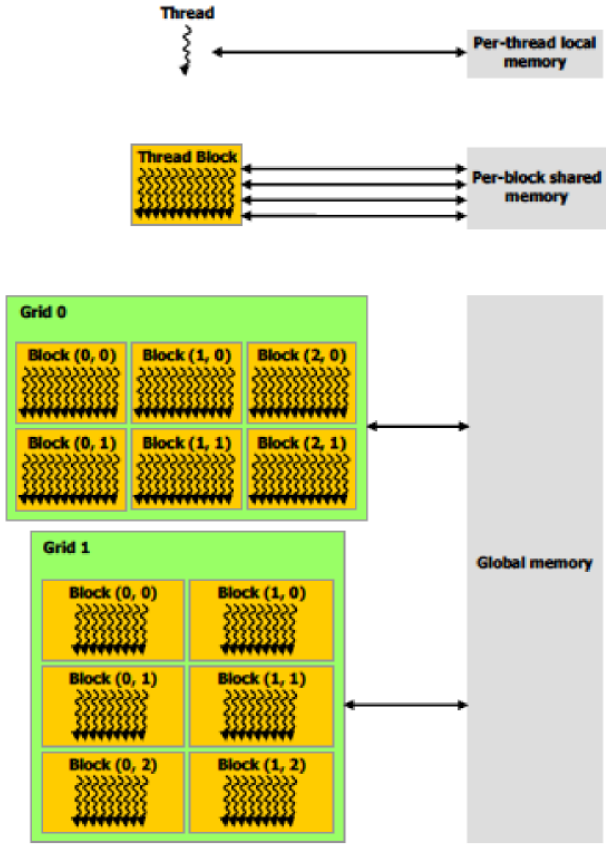


Figure 1.3: The CUDA Memory Hierarchy. Taken from the CUDA C Programming Guide

another warp that is ready to run and schedules it. For the full (more interesting) picture of block/warp/thread scheduling, see Chapter 4 of the CUDA C Programming Guide.

The user has no control over the order in which blocks are scheduled onto processors, and therefore the blocks must be able to run in any order, independent of one another. This unique requirement means that the code will scale naturally when run on a device with more multiprocessors. For example, this document is being written on a MacBook Pro running CUDA on its built-in NVIDIA GT 330M graphics card. This card has 6 multiprocessors, each with 8 cores, giving 48 cores altogether. A machine with NVIDIA's GTX470 graphics card has access to 14 multiprocessors, each with 32 cores, for 448 cores total. Code written for this MacBook will run *as*

written on the GTX470, but will benefit from the greater number of cores. Since the blocks are independent, they can be distributed over the greater number of multiprocessors without any worry about race conditions or synchronization problems. (See Appendix F of the CUDA C Programming Guide for more.)

Given this requirement, how can we make sure that every thread has finished computing its entry in M_k before any begins to compute its entry in M_{k+1}? Our implementation of Floyd-Warshall uses separate kernels, as shown in the previous code snippet. We can achieve synchronization by forcing each kernel to run to completion before the next kernel is launched by means of the `cudaThreadSynchronize()` function.

```
cudaMemcpy(devArray, graph, 36*sizeof(int), cudaMemcpyHostToDevice);
for(int k = 0; k < 6; k++) {
   fwStepK <<< 1, 36 >>> (k, devArray, 6, 6);
   cudaThreadSynchronize();
}
cudaMemcpy(graph, devArray, 36*sizeof(int), cudaMemcpyDeviceToHost);
```

Without this statement, it is possible that blocks from one kernel may still be running when blocks from the next kernel are put on another multiprocessor. (Only devices of compute capability 2.1 or higher have the capability of running multiple kernels simultaneously. See section 3.2.5 of the CUDA C Programming Guide, as well as Appendix F.)

We are now ready to examine the built-in variables (Section B.4 in the CUDA C Programming Guide.)

As shown in Figure 1.4, the threads in a CUDA kernel are grouped into blocks, and these blocks are grouped into a grid. In the Figure, the blocks and grid are both two-dimensional, but these can be one-, two- or three-dimensional depending on the programmer's need. Our kernel given above uses a two-dimensional layout of the blocks and grid, since that most naturally conforms to the shape of the matrices we are using.

When a thread is created several built-in variables are defined. When a thread reads these variables, the results locate the thread within its block and the grid. Some of these use the `uint3` data type, which is a 3-dimensional vector of unsigned ints, or the `dim3` type which is based on `uint3`. If variable `D` is of type uint3 or dim3, then its components can read as `D.x`, `D.y` and `D.z`.

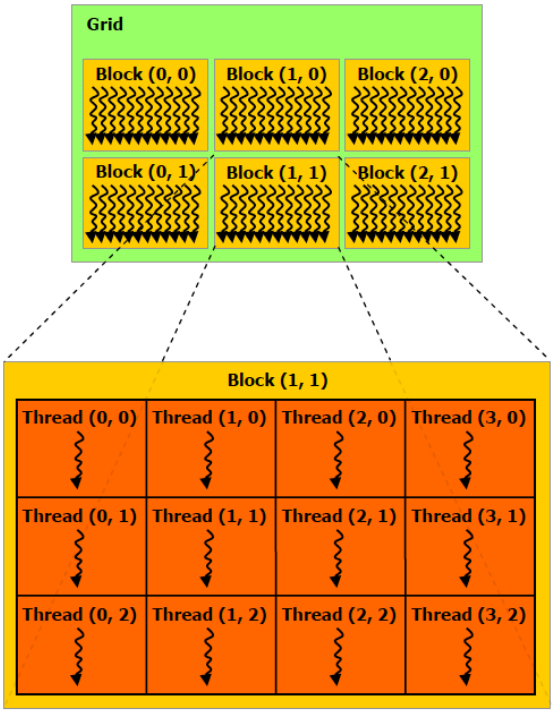


Figure 1.4: Grids and Blocks of Threads. Taken from the CUDA C Programming Guide

- `gridDim`. This is of type `dim3`, and gives the dimensions of the grid. In our kernel, this was equal to `(1, 1, 1)`.
- `blockIdx`. This is of type `uint3` and gives the coordinates of the thread's block containing the thread, within the grid. For all of the threads in our six-vertex graph example, this would equal `(1, 1, 1)`.
- `blockDim`. This is of type `dim3`, and gives the dimensions of the blocks. In our example, this is equal to `(36, 1, 1)` for every thread.
- `threadIdx`. This is of type `uint3` and gives the coordinates of the thread within its block. In our example, this is `(i, 1, 1)` for the i'th thread in the block, $0 \leq i < 36$.

An example helps us understand how to use these variables. Suppose we are processing a matrix of size 500×800, with one thread per matrix element, and blocks of dimensions 32×32. We would launch our kernel with a grid having dimension

$500/32 \times 800/32$, with each dimension rounded up to the nearest integer so that the entire matrix is covered, giving a 16×25 grid. When the kernel is launched, $16 \times 25 = 400$ blocks will be created, each with $32 \times 32 = 1024$ threads. Then for each thread,

```
col = blockIdx.x * blockDim.x + threadIdx.x
row = blockIdx.y * blockDim.y + threadIdx.y
```

as suggested by Figure 1.4. In `fw.c` we chose blocks of dimension 1×256. (See the definition `int threadsPerBlock = 256;` in the `main` method. To best understand why we chose blocks of dimensions 1×256 in our kernel, let us assume that we have a graph with 1,000 vertices instead of just six. In this case we would have a million threads logically arranged in a $1,000 \times 1,000$ matrix. On devices of compute capability 1.x (which includes this MacBook Pro's card — see Appendices A and F of the CUDA C Programming Guide for a discussion on compute capabilities. Roughly it describes how much computing power the device has in terms of API calls and hardware.) a block may contain at most 512 threads. In our case, however, it is our use of shared memory that limits block size. We will therefore have to partition the threads into blocks, all of of which must be the same size. There are many ways that this may be done. For example, we may have

- A 100×100 grid of blocks, each of size 10×10 (blocks are square submatrices)
- A 2×1000 grid of blocks, each of size 500×1 (blocks are portions of columns)
- Or even a grid of 64×64 blocks, each of size 16×16. (blocks are squares submatrices, with some "hanging over" the right and bottom edges of the matrix

This last may seem odd, but sometimes it makes sense to use sizes that are powers of 2, in order to facilitate coalescing of accesses to global memory. (See section F.3.2 of the CUDA C Programming Guide for more on coalescing.)

In our code we use blocks that comprise portions of *rows* of the matrix, for the following reason. We store our matrix as a one-dimensional array in row-major order; that is, the first row comprises the first entries in the array, then the second row, then the third, and so on. Thus the rows are stored contiguously in memory, while the columns get chopped up and scattered. The left side of Figure 1.5 shows the situation where a block of threads comprises a portion of a row of the matrix. When computing $d_k(i, j)$ from M_{k-1}, which is stored in global memory, thread (i, j) needs access to the (i, k) entry of M_{k-1}, and the (k, j) entry of that same matrix.

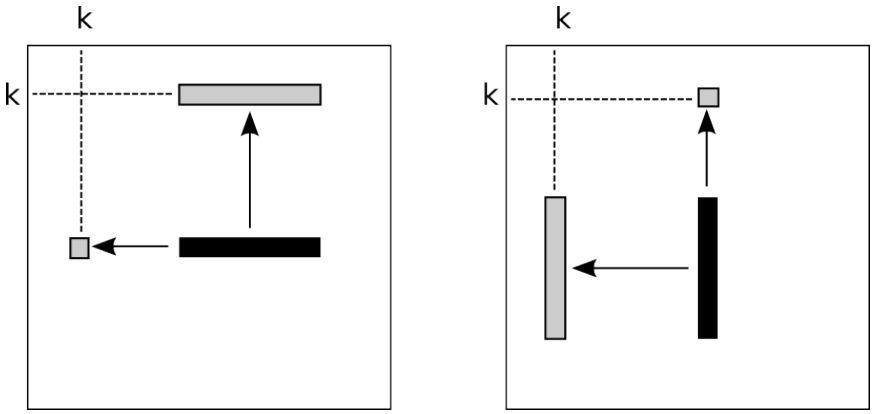


Figure 1.5: Left: Blocks are rows of the matrix. Right: Blocks are columns.

The threads in the block will all share the (i, k) entry. This entry (the little square on the left side of the figure) is read by the first thread in the block, that is, the thread having `threadIdx.x == 0` and stored in the shared variable `trkc`. After that one read from (slow) global memory, all threads in the block will have access to it via the (fast) shared memory.

When the threads in this block ask for the matrix entry (k, j), all those threads have different values of j, so they are all asking for different entries in global memory. These entries are depicted by the gray rectangle at the top of the matrix. Notice that they comprise a contiguous portion of a row, and thus a contiguous portion of the array in global memory. When the addresses that the threads are asking for are contiguous in global memory, the hardware can coalesce these requests into a single memory transfer. As NVIDIA says,

> "*Perhaps the single most important performance consideration in programming for the CUDA architecture is coalescing global memory accesses.*"
>
> —Section 3.2.1 of the CUDA C Best Practices Guide [4]

Contrast this with the case of having our blocks be portions of the columns of the matrix, as shown on the right of Figure 1.5. There we could share the single entry at the top of the matrix, but we'd need to load all of the entries in the vertical block on the left side of that matrix. These would not be stored contiguously, and so would require more accesses to the (slow) global memory.

1.4.3 Segment alignment and warps

Now we are in a position to explain why we introduce the variable `Na` in the code. GPUs of all compute capabilities logically break a block of threads into *warps* of 32 consecutive threads. So if a block has 128 threads, then threads $0 \ldots 31$ will comprise the first warp, threads $32 \ldots 63$ will comprise the second warp, threads $64 \ldots 95$ will comprise the third warp, and threads $96 \ldots 127$ will comprise the last warp. When processing a block, the *warp scheduler* looks for a warp that has threads that are ready to run (that is, not waiting for I/O of some sort) and will put them on the multiprocessor. These will then run together, even if threads from a different warp in the same block are far ahead or behind.

When programming in CUDA it is important for several reasons to be aware of warps. For example, on devices of compute capability 1.2 or greater, you can do an `AND` operation across all threads in a warp in a single instruction. (See section B.12 of the CUDA C Programming Guide.) What matters for us here is that it is at the level of the warp where global memory accesses are coalesced. When the 32 threads of a warp each ask for an `int` from global memory, and those 32 `int`s are stored consecutively, then these may be coalesced into a single 128-byte access. *But only if that block of 128 bytes begins on a memory address which is a multiple of 128.* This is the notion of *alignment*.

Here is the code / pseudocode for the portion of `fwHelpers.cpp` that builds the initial matrix M_0:

```
int alignment = 32;
Na = alignment * ( (N + alignment - 1) / alignment);

int* a = (int*) malloc(N * Na * sizeof(int) );
for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++)
    if( there's an edge from vertex i to vertex j )
      a[i * Na + j] = the edge's weight
    else
      a[i * Na + j] = INT_MAX;
return a;
```

The variable `N` represents the number of vertices in the graph. `Na` rounds this number up to a multiple of `alignment`, which is set to 32 in our code snippet. As a result, the

`malloc` statement reserves enough memory to hold a matrix of height `N` and width `Na`. This over-allocation wastes some memory, but now every row of the matrix will begin on a memory address which is a multiple of `32 * sizeof(int)`, that is, 128 for 32-bit machines or 256 for 64-bit machines. This is relevant because coalesced accesses can take place only when the blocks of memory being accessed are properly *aligned*, meaning that they begin at a memory location that is a multiple of 64, 128, 256, etc... as appropriate. (Again, see section F.3.2 of the CUDA C Programming Guide for more on coalescing.)

The various versions of `cudaMalloc` are guaranteed to return pointers aligned to 256 bytes. (See Section 3.2.1.2 of the CUDA C Best Practices Guide.) Our matrices will therefore begin at a memory address which is a multiple of 256, and all subsequent rows will begin on multiples of 128 and/or 256. And if our block sizes are multiples of 32, then all rows requested by our blocks will begin at addresses which are multiples of 128 (32 `int`s = 128 or 256 bytes). All of this to encourage coalescing!

Chapter 2

Lab Exploration

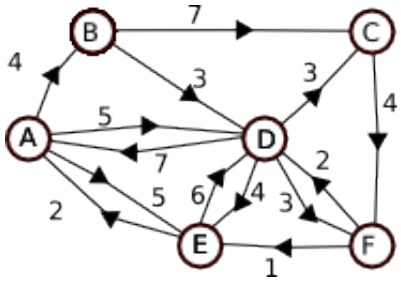
2.1 Installation of CUDA

This has probably been done for you by your instructor, who can give you the details of the installation. If not, see Appendix A of the NVIDIA CUDA C Programming Guide to find out what graphics cards support CUDA, and then use the appropriate NVIDIA CUDA C Getting Started Guide (for Windows, Linux or Mac) to install the developers kit and drivers.

You probably have a successful install when you can run `nvcc --version` and receive a message from the compiler, and `deviceQuery` to find out the details of your CUDA-enabled card.

2.2 First run

Find the directory containing the files `fw.c, fwHelpers.h, fwHelpers.cpp` and `Makefile`. These are available in zipped format from this module's website. Type `make fw` to build the executable `fw`. This is the implementation of the Floyd-Warshall algorithm discussed above. The "`fwHelpers`" files include code for reading a graph from a file, and writing out the adjacency matrix of the graph before, during and/or after the algorithm's run.




```
A B 4
A D 5
A E 5
B C 7
B D 3
C F 4
D A 7
D C 3
D E 4
D F 3
E A 2
E D 6
F D 2
F E 1
--END--
```

Figure 2.1: A directed graph and its description in a text file

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.

2.2.1 File format for describing graphs

You'll notice that the code in `fwHelpers.cpp` for reading a graph from a file is rather longer than you might expect. This is to make it easy for the programmer to describe graphs in the file. Vertices are named by arbitrary ASCII strings of non-whitespace characters, and an edge from V to W of length l is described in the file by placing `V W` l on a single line. The last line of graph file must be "`--END--`". The program reads the number of edges and vertices, so there is no need to specify these in the input file. The input file for the graph of Figure 1.2 is shown in Figure 2.1, and is included with this module as `graph1.txt`. The file `graph2.txt` is the input for the graph shown in Figure 1.1.

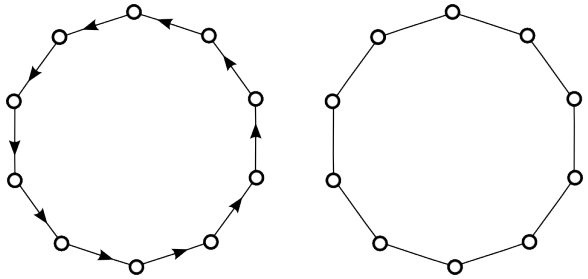


Figure 2.2: A directed 10-cycle (left) and an undirected 10-cycle (right)

Also included with this module is a simple C program for generating directed and undirected cycles. Type `make cycleProgs` to get two executables, `makeDirCycle` and `makeUndirCycle`. Then type, for example, `makeDirCycle 10` to generate a directed 10-cycle, and `makeUndirCycle 10` to generate an undirected 10-cycle, as shown in Figure 2.2. An undirected cycle on n vertices is a good test case for the Floyd-Warshall algorithm, since it takes a full n iterations of the algorithm to find all shortest paths.

2.2.2 Compile and run

Compile the program by typing `make fw` at a command line. This generates an executable called `fw`. To run the program, type `fw graphFileName`. For example, to run the algorithm on the graph in Figure 1.1 you would type `fw graph2.txt`. By default, the program prints out the distance matrix for up to the first ten vertices of the graph, after the algorithm terminates. You can insert extra `printArray()` commands before or in the loop in `main()` to see the matrices $M_0, M_1, \ldots$. See the two commented-out lines in the loop in `fw.c`.

You'll notice that in the code we make use of the variable `err` which is of type `cudaError_t`. This is very helpful for debugging, since most CUDA commands involving memory allocation and copy return an error value which can be tested, and even printed using the function `cudaGetErrorString()`, as shown in the code.

Spot Check 4 *This is a continuation of Spot Check 3. Uncomment out the three lines in the* `for` *loop in* `main()` *that print the partial tables and type* `make fw` *to compile the program. Run it with the input graph* `graph1.txt` *by typing* `./fw graph1.txt` *at the commandline, and check your answers from Spot Check 3.*

2.2.3 Running on Lonestar, a queue-based supercomputer running CUDA

If you can get an acount on Lonestar, a scientific computing cluster of the Texas Advanced Computing Center, then here are the steps for running the code in this module.

1. Compile fw as discussed above.
2. Create a file called sgeFile.txt, with the following contents (except with your own email address!):

```
#!/bin/bash
#$ -V #Inherit the submission environment
#$ -cwd # Start job in submission directory
#$ -N fwRun # job name
#$ -j y # Combine stderr and stdout
#$ -o $JOB_NAME.o$JOB_ID # Name of the output file
#$ -q gpu # Queue name gpu
#$ -pe 1way 12 # 1 node having 12 cores
#$ -l h_rt=00:30:00 # Run time (hh:mm:ss) 30 mins
#$ -M robh@shodor.org # email notification - use yours!
#$ -m be # Email at Begin and End of job
set -x # Echo commands, use set echo with csh
ibrun fw graph.txt # Run the program
```

   Then type `qsub sgeFile.txt` to add your request to the queue. You will receive an email when the job is started, and another when the job ends. The system will also create an output file called `fwRun.o314432`, where the "314432" is replaced by the system-assigned job id.

2.3 Scalability of the Code

We tested our CUDA-enabled code on machines with four different compute capabilities. (See appendix F of the CUDA C Programming guide for a discussion of compute capabilities.) For our tests, we created undirected cycles of sizes 10, 20, 30, ..., 1800, and ran our algorithm on each of them ten times and took the average.

We wished to evaluate the speed of the CUDA cards themselves, and correct for the overhead of copying to and from the card in our tests. This can be important. For example, my MacBook Pro will run faster on graphs up to about 500 vertices than Lonestar at the Texas Advanced Computing Center, which runs a much more powerful graphics processor. The reason is that the graphics cards at Lonestar reside in an expansion box off the motherboard, making memory transfers to and from the card a bit slower. So even though Lonestar's card is about six times faster than that on my Mac, jobs with a high memory transfer to compute ratio may take longer.

To correct for memory transfers to and from the card, we compiled a second version of `fw` that copied the matrix to the card, did not invoke the kernel, and then copied the memory back off the card. We alternated runs with no kernel with runs having the kernel, completing ten runs of each. We took the difference of the averages, and these differences are shown in Figure 2.3.

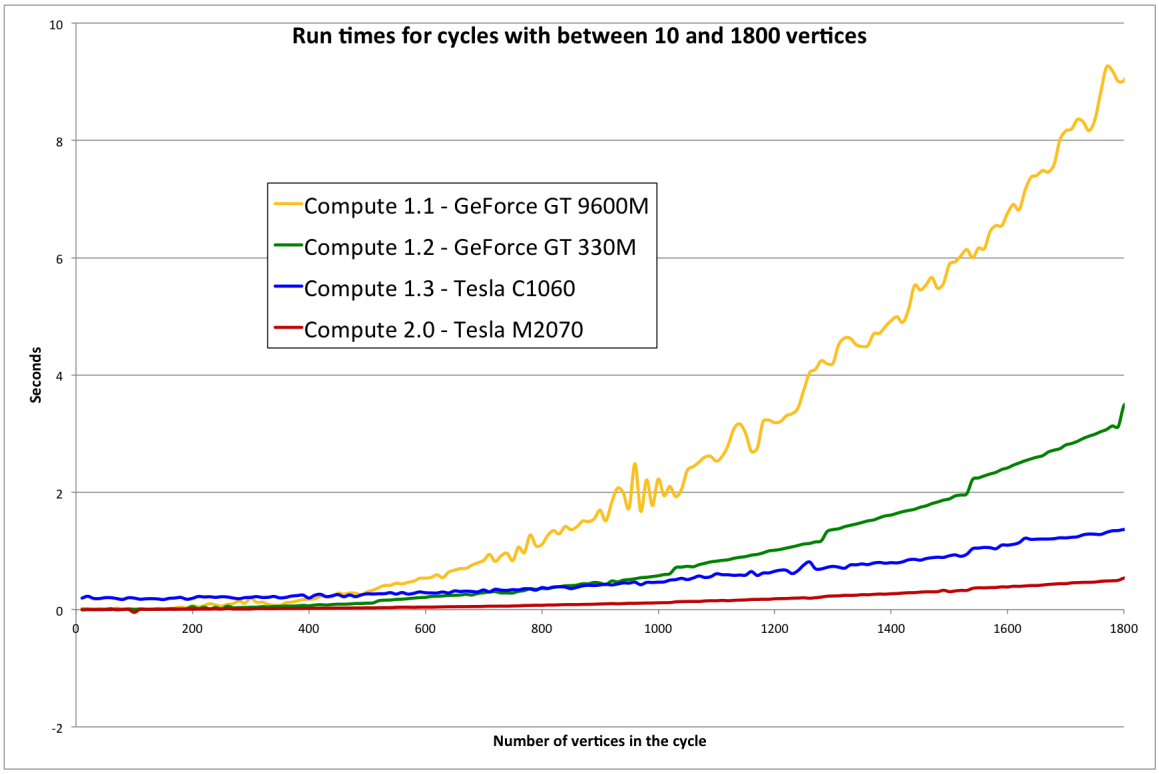


Figure 2.3: Kernel run times for `fw` on graphs having between 10 and 1800 vertices, on machines with four different compute capabilities.

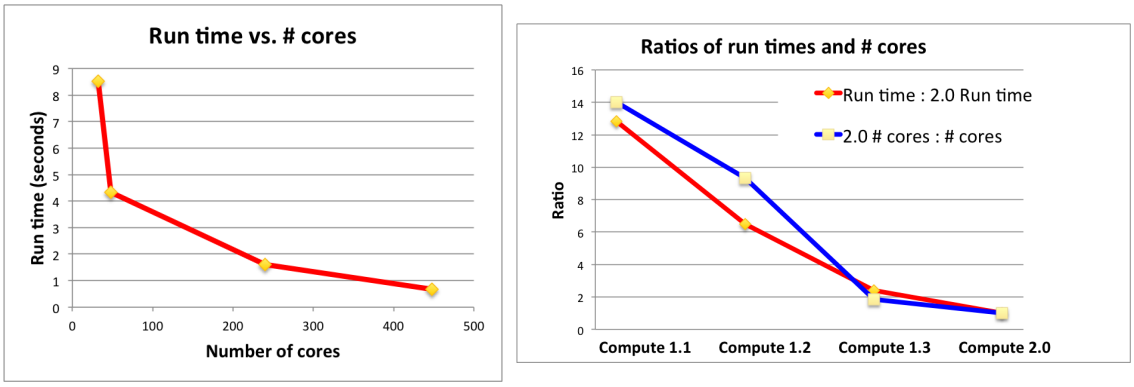


Figure 2.4: Left: Run time vs. # cores on a 1980 vertex cycle. Right: Plots of ratios have similar shapes.

The table below gives some details about these four video cards:

Card	Comp. Cap.	# Multiprocs.	# Cores/MP	Total # cores
GeForce GT 9600M	1.1	4	8	32
GeForce GT 330M	1.2	6	8	48
Tesla C1060	1.3	30	8	240
Tesla M2070	2.0	14	32	448

In an ideal situation, the run time of a problem would be inversely proportional to the number of cores available to it. The CUDA architecture comes close to providing that for us, at least for the four machines we've tested our algorithm on. Figure 2.4 shows on the left a plot of run time vs. number of cores, and on the right a plot showing that the fractions

$$\frac{\text{Run time on machine } k}{\text{Run time on Longhorn}} \quad \text{and} \quad \frac{\text{\# processors on Longhorn}}{\text{\# processors on machine } k}$$

have roughly the same shape when plotted.

2.4 Explorations

1. **Theory:** The blocks of threads in the kernel are required to be independent. That is, the program must produce the correct result regardless of the order in which the blocks are scheduled. But consider the kth row and column of the

matrix. Every entry (i, j) of the matrix is looking at entries (i, k) and (k, j) when doing its update. But these three entries may be in entirely different thread blocks, and so the order in which they are processed might make a difference in the final outcome. Show that it does not.

2. **__syncthreads():** The kernel function in `fw.c` contains a single `__syncthreads()` call. Experiment to see whether this call is really necessary. Then explain how the wrong answer could be arrived at without this call.

3. **cudaMallocPitch:** CUDA provides a more natural mechanism for allocating a two-dimensional array; the `cudaMallocPitch` command. (See Section 5.3.2.1.2 of the CUDA C Programming Guide.) Re-write the code using this method instead of making use of our `ALIGNMENT` and `Na` variables. See if there is any improvement in the running time.

4. **Columns Instead of Rows:** Re-write the code using blocks of threads that are columns instead of rows, and compare the running time with the current, row-based blocks.

5. **Is CUDA Worth the Trouble?** You can substitute the following lines of code for `main()` in `fw.c`,

```
int main(int argc, char* argv[])\{
  int N = 0;
  int Na = 0;
  int* graph = readGraph(N, Na, argv[1]);
  printf("Read %s with %d vertices, Na = %d\n", argv[1], N, Na);
  printArray(N, graph);
  for(int k = 0; k < N; k++)
    for(int i = 0; i < N; i++)
      if(graph[i*N + k] != INT_MAX)
        for(int j = 0; j < N; j++)
          if(graph[k*N + j] != INT_MAX)
            if(graph[i*N + k] + graph[k*N + j] < graph[i*N + j])
              graph[i*N + j] = graph[i*N + k] + graph[k*N + j];
  printArray(N, graph);
\}
```

and change the line in `fwHelpers.cpp`

```
Na = alignment*((N + alignment-1)/alignment);
```

to

```
Na = N;
```

and have a single-processor, non-parallel version of the Floyd-Warshall algorithm. (This is the typical implementation.) Compare these running times against the CUDA-based solutions. At what point is it worth it to launch a CUDA kernel to do the work? Is it ever *really* worth it?

6. **Speedup Considerations:** Suppose that we have the very latest system with K CUDA-enabled cards, each with M multi-processors, each of which has C cores. On this system we run the algorithm described in this module on graphs with n vertices. As n gets large, what function describes the growth of the running time $T(n)$ for solving the problem on n vertices? For example, would you say it grows linearly ($T(n) = O(n)$)? quadratically ($T(n) = O(n^2)$)? exponentially ($T(n) = O(2^n)$)?

7. **Jumps in the Green Graph:** The green plot in Figure 2.3 seems to take a leap upward with some regularity. Explain why this might be happening. (Another question might be to ask why we *don't* see this happen in the other three curves. I don't know the answer to that question.)

8. **The Actual Shortest Path:** The code currently produces a matrix whose (i, j)-entry is the length of the shortest path from vertex i to vertex j. Modify the code so that the program produces another matrix whose (i, j)-entry is the vertex you should step to from vertex i if you are traveling along a shortest path from vertex i to vertex j. For example, the (A,G)-entry in the graph of Figure 1.1 would be "C". Then show how to use this matrix to build the whole path from i to j.

9. **Strong Scaling:** The phrase *strong scaling* is used to describe how well running time improves when more processors are applied to the solution of some fixed problem. For example, we may wish to measure how quickly we can solve the all-pairs shortest path problem on a graph with 1000 vertices on a single-CPU system, a CUDA-capable system with 48 cores, and a CUDA-capable system with 448 cores. Can the problem be solved in 1/448 the time with 448 cores as compared to the single-CPU system? If you have different CUDA-capable systems available with different numbers of cores, compare running times on some large, fixed-size problem to see how well our implementation of the Floyd-Warshall algorithm scales.

Chapter 3

The Code

3.1 fw.c

```
#include <stdio.h>
#include "fwHelpers.h"

/*****************************************
 * Kernel to improve the shortest paths by considering the kth vertex
 * as an intermediate.
 *
 * arg:  k = the index of the vertex we are using as intermediate
 * arg:  devArray = the array containing the matrix
 * arg:  N  = the number of vertices in the graph
 * arg:  Na = the width of the matrix as stored on the device's memory.
 *
 * The graph is stored as an N x Na matrix, with the (i, j) matrix entry
 * being stored as devArray[i*Na + j]
 *
 *****************************************/
__global__ void fwStepK(int k, int devArray[], int Na, int N){
  int col = blockIdx.x * blockDim.x + threadIdx.x; /* This thread's matrix column */
  if(col >= N)
    return;
```

```
  int arrayIndex = Na * blockIdx.y + col;
  __shared__ int trkc;        /* this row, kth column */

  // Improve by using the intermediate k, if we can
  if(threadIdx.x == 0)
    trkc = devArray[Na * blockIdx.y + k];
  __syncthreads();

  if(trkc == INT_MAX)    /* infinity */
    return;

  int tckr = devArray[k*Na + col]; /* this column, kth row */
  if(tckr == INT_MAX)    /* infinity */
    return;

  int betterMaybe = trkc + tckr;
  if(betterMaybe < devArray[arrayIndex])
    devArray[arrayIndex] =  betterMaybe;
}



/*******************************************************
 * main
 *
 * We read in the graph from a graph file, in this format:
 * First we read in N, the number of vertices
 * The vertices are assumed to be numbered 1..N
 * Then we read in triples of the form s d w
 *   where s is the source vertex, d the destination, and w the weight
 *
 * The "best weight so far" matrix stores rows contiguously, with
 * bwsf(i, j) being the best weight FROM i TO j
 *
 * Thread blocks comprise rows of the matrix, so that we can
 * take advantage of global memory access grouping
 *******************************************************/
int main(int argc, char* argv[]){
```

```
  int N = 0; /* Number of vertices                        */
  int Na = 0; /* Width of matrix to encourage coalescing */

  int* graph = readGraph(N, Na, argv[1]); /* from fwHelpers.cpp            */
  printf("Kernel: Just read %s with %d vertices, Na = %d\n", argv[1], N, Na);

  // Copy the array to newly-allocated global memory
  int* devArray;
  cudaError_t err = cudaMalloc(&devArray, Na*N*sizeof(int));
  printf("Malloc device rules: %s\n",cudaGetErrorString(err));
  err = cudaMemcpy(devArray, graph, Na*N*sizeof(int), cudaMemcpyHostToDevice);
  printf("Pre-kernel copy memory onto device: %s\n",cudaGetErrorString(err));

  // Set up and run the kernels
  int threadsPerBlock = 256;
  dim3 blocksPerGrid((Na + threadsPerBlock - 1) / threadsPerBlock, N);

  // The kth run through this loop considers whether we might do better using
  // the kth vertex as an intermediate
  for(int k = 0; k < N; k++){
    fwStepK <<< blocksPerGrid, threadsPerBlock >>> (k, devArray, Na, N);
    err = cudaThreadSynchronize();

    // Uncomment the following line when debugging the kernel
    // printf("Kernel: using %d as intermediate: error = %s\n", k, cudaGetErrorString

    // Uncomment the following two lines to print intermediate results
    // err = cudaMemcpy(graph, devArray, Na*N*sizeof(int), cudaMemcpyDeviceToHost);
    // printArray(Na, graph);
  }

  err = cudaMemcpy(graph, devArray, Na*N*sizeof(int), cudaMemcpyDeviceToHost);
  printf("Post-kernel copy memory off of device: %s\n",cudaGetErrorString(err));

  printArray(Na, graph);
  free(graph);
  cudaFree(devArray);
}
```

3.2 fwHelpers.cpp

```
#include "fwHelpers.h"

map<string, int> nameToNum;  /* names of vertices */
map<string, map<string, int> > weightMap; /* weights of edges */

int* readGraph(int& N, int& Na, char* argv){

  // Read the graph file from memory
  string vname1, vname2;
  ifstream graphFile;
  string dummyString;
  int thisWeight; /* weight of the edge just read from file */
  N = 0; /* number of vertices */
  graphFile.open(argv);

  //Read the graph into some maps
  graphFile >> vname1;
  while(vname1 != "--END--"){
    graphFile >> vname2;
    graphFile >> thisWeight;
    if(nameToNum.count(vname1) == 0){
      nameToNum[vname1] = N;
      weightMap[vname1][vname1] = 0;
      N++;
    }
    if(nameToNum.count(vname2) == 0){
      nameToNum[vname2] = N;
      weightMap[vname2][vname2] = 0;
      N++;
    }
    weightMap[vname1][vname2] = thisWeight;
    graphFile >> vname1;
  }
  graphFile.close(); // Nice and Tidy

  // "alignment" is what stored row sizes must be a multiple of
```

```
  int alignment = ALIGNMENT;
  Na = alignment*((N + alignment-1)/alignment); /* for the sizes of our arrays */
  printf("Alignment = %d\n", alignment);
  // Build the array
  int* a = (int*) malloc(N*Na*sizeof(int));
  for(int ii = 0; ii < N; ii++)
    for(int jj = 0; jj < N; jj++)
      a[ii * Na + jj] = INT_MAX;
  map<string, int>::iterator i;
  map<string, int>::iterator j;
  for(i = nameToNum.begin(); i != nameToNum.end(); ++i)
    for(j = nameToNum.begin(); j != nameToNum.end(); ++j){
      if(weightMap[(*i).first].count((*j).first) != 0){
a[Na * (*i).second + (*j).second] = weightMap[(*i).first][(*j).first];
      }
    }
  return a;
}


void printArray(int Na, int* a){
  map<string, int>::iterator i, j;
  for(i = nameToNum.begin(); i != nameToNum.end(); ++i)
    if(i->second < 10)
      printf("\t%s", i->first.c_str());
  printf("\n");
  j = nameToNum.begin();
  for(i = nameToNum.begin(); i != nameToNum.end(); ++i){
    if(i->second < 10){
      printf("%s\t", i->first.c_str());
      for(j = nameToNum.begin(); j != nameToNum.end(); ++j){
if(j->second < 10){
  int dd =  a[i->second * Na + j->second];
  if(dd != INT_MAX)
    printf("%d\t",dd);
  else
    printf("--\t");
}
```

```
      }
      printf("\n");
    }
  }
}
```

3.3 fwHelpers.h

```
#ifndef FWHELPERS_H
#define FWHELPERS_H

#ifndef ALIGNMENT
#define ALIGNMENT 64
#endif

#include <stdio.h>
#include <string>
#include <map>
#include <iostream>
#include <fstream>

using namespace std;

int* readGraph(int&N, int&Na, char* argv);
void printArray(int Na, int* a);

#endif
```

Chapter 4

Answers to Spot Checks

1. There are many possible answers. Here is just one:
   A, C, F, E, D, G, I, H, Y, X, J, W, K, Z, B.

2. The two numbers that changed are the "11" and "6" in the second column. They were both ∞ in the M_0 matrix.

M_0	A	B	C	D	E	F
A	0	4	∞	5	5	∞
B	∞	0	7	3	∞	∞
C	∞	∞	0	∞	∞	4
D	7	11	3	0	4	3
E	2	6	∞	6	0	∞
F	∞	∞	∞	2	1	0

3. The four entries that are still ∞ are those in the 'C' row corresponding to paths from C to vertices A, B, D and E.

4. Here is the output:

```
Alignment = 64
Kernel: Just read graph1.txt with 6 vertices, Na = 64
Malloc device rules: no error
Pre-kernel copy memory onto device: no error
Kernel: using 0 as intermediate: error = no error
        A       B       C       D       E       F
A       0       4       --      5       5       --
B       --      0       7       3       --      --
C       --      --      0       --      --      4
D       7       11      3       0       4       3
E       2       6       --      6       0       --
F       --      --      --      2       1       0
Kernel: using 1 as intermediate: error = no error
        A       B       C       D       E       F
A       0       4       11      5       5       --
B       --      0       7       3       --      --
C       --      --      0       --      --      4
D       7       11      3       0       4       3
E       2       6       13      6       0       --
F       --      --      --      2       1       0
Kernel: using 2 as intermediate: error = no error
        A       B       C       D       E       F
A       0       4       11      5       5       15
B       --      0       7       3       --      11
C       --      --      0       --      --      4
D       7       11      3       0       4       3
E       2       6       13      6       0       17
F       --      --      --      2       1       0
Kernel: using 3 as intermediate: error = no error
        A       B       C       D       E       F
A       0       4       8       5       5       8
B       10      0       6       3       7       6
C       --      --      0       --      --      4
D       7       11      3       0       4       3
```

```
E       2       6       9       6       0       9
F       9       13      5       2       1       0
Kernel: using 4 as intermediate: error = no error
        A       B       C       D       E       F
A       0       4       8       5       5       8
B       9       0       6       3       7       6
C       --      --      0       --      --      4
D       6       10      3       0       4       3
E       2       6       9       6       0       9
F       3       7       5       2       1       0
Kernel: using 5 as intermediate: error = no error
        A       B       C       D       E       F
A       0       4       8       5       5       8
B       9       0       6       3       7       6
C       7       11      0       6       5       4
D       6       10      3       0       4       3
E       2       6       9       6       0       9
F       3       7       5       2       1       0
Post-kernel copy memory off of device: no error
        A       B       C       D       E       F
A       0       4       8       5       5       8
B       9       0       6       3       7       6
C       7       11      0       6       5       4
D       6       10      3       0       4       3
E       2       6       9       6       0       9
F       3       7       5       2       1       0
```

Bibliography

[1] Aydin Buluç, John R. Gilbert, and Ceren Budak. Solving path problems on the gpu. *Parallel Comput.*, 36:241–253, June 2010.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[3] The NVIDIA Corporation. *The CUDA API Reference Manual, v4.0*. NVIDIA Corporation, 2011.

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

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

[6] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.