# Dynamic Programming with CUDA — Part I Assessment

Robert Hochberg

August 21, 2012

# Assessment of Student Knowledge

1. **Understanding of the Algorithm:** A student who can answer the **Theory**, **Speedup Considerations** and **The Actual Shortest Path** questions below, has likely understood the Floyd-Warshall algorithm and the basics of parallel programming. No CUDA knowledge could be inferred, though.
2. **Understanding of CUDA:** Students who can compile and run the sample code, and answer the `__syncthreads()` and **Jumps in the Green Graph** questions has likely understood the CUDA programming model, particularly the important notions of warps, blocks and the memory model. No working knowledge or programming skills could be inferred, though.
3. **Comfort with the Environment:** Students who can work through **Is CUDA Worth the Trouble?**, generating numerical comparison data and finding where the running times of the linear and parallel algorithms are about the same, demonstrate comfort with the compile/run process and generating test graphs.
4. **CUDA Mastery:** Students who can solve the `cudaMallocPitch:` and/or **Columns Instead of Rows** problems have likely gained enough skill to begin writing programs from scratch in the CUDA environment in a way that takes advantage of its capabilities.

# Solutions

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 $k$th 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.

> Consider an entry $(k, j)$ in the $k$th row of $M_{k-1}$. When its thread considers updating that entry for matrix $M_k$ it checks the inequality $d_{k-1}(k, j) < d_{k-1}(k, j) + d_{k-1}(j, k)$, which of course will never be satisfied. Thus entries in the $k$th row will not change on the $k$th iteration of the algorithm, that is, when computing $M_k$. Same for the entries in the $k$th column. Thus whether these blocks are computed before or after any other blocks is irrelevant.

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.

> Yes, disaster could result without this synchronization call. The shared variable `trkc` needs to be read before any thread is allowed to use it. Since only the first thread in the block reads this value from global memory, it is possible that threads from other warps may dash ahead while the first thread is still waiting for this value, and if they do, they will compute using the wrong value for `trkc`.

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.

> Code for this is given in the files `fwPitch.c`, `fwHelpersPitch.cpp` and `fwHelpersPitch.h`. Use `make -f Makefile-assess fwPitch` to generate the executable `fwPitch` which can be used for testing.

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.

> Code for this is given in the files `fwCols.c` and `fwHelpersCols.cpp`, Use `make -f Makefile-assess fwCols` to generate the executable `fwCols` which can be used for testing.

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?

On my MacBook Pro (compute capability 1.2) the running times are about equal for graphs on 600 vertices. For smaller graphs, the non-parallel version runs quicker. *Much* quicker for very small graphs. But is it *ever* worth it? Yes. Even for a graph with only 1000 vertices, the running time is about 4 seconds for the non-parallel version, and about 2 seconds for the CUDA-enabled version. For 2000 vertices, those times are 36 seconds vs. 9 seconds. (Times are wall times, and include all memory transfers to and from the device.)

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 ($T(n) = O(2^n)$)?

> The algorithm we are using here is still the Floyd-Warshall algorithm, with its triply nested loops. It therefore has running time $O(n^3)$. All we are doing with CUDA is decreasing the constant in front of the $n^3$ term. This is a fundamental fact about parallelizing algorithms: If you have a fixed-size computing device, then regardless of how well it computes in parallel, it can't decrease the running time for larger and larger problems by more than a constant factor. The good news is that that factor gets larger and larger every year as machines get better and better.

7. **Jumps in the Green Graph:** The green plot in the run-times Figure 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 four curves. I don't know the answer to that question.)

> The jumps happen at the multiples of 256, which is the size of our thread blocks. When the matrix has width 1024, for example, we need four thread blocks to cover a row. When the width increases to 1025, we add a fifth thread block *for each row*. Thus the total number of thread blocks, which had been going up by 4 each time a new vertex was added, suddenly jumps by 1029. So even though most of the threads in the blocks overhanging the right edges of the matrix will exit immediately, they still have to be created and run, giving us these large jumps at the multiples of 256.

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 $j$.

Suppose that we have created a new matrix `devPathArray` to hold these steps. Then we can change the lines:

```
if(betterMaybe < devArray[arrayIndex])
  devArray[arrayIndex] =  betterMaybe;
```

to:

```
if(betterMaybe < devArray[arrayIndex]){
  devArray[arrayIndex] =  betterMaybe;
  devPathArray[arrayIndex] = k;
}
```

The array `devPathArray` should be initialized so that its $(i, j)$-entry is $j$ for all $i$ and $j$.
To read the shortest path from vertex $i$ to vertex $j$ after the algorithm has run, first print out vertex $i$, and set variable $k$ equal to $i$. Thereafter, print out the $(k, j)$-entry of the matrix, and then set $k$ equal to this entry, finishing when $k$ is equal to $j$.