Parallel and Distributed Systems

All Pair Shortest Path on the GPU using CUDA

Marios Mitalidis
Aristotle University of Thessaloniki
mariosam@ece.auth.gr

5/11/2016

Abstract

The algorithm studied is All-Pairs Shortest Path. Given as input the adjacency matrix of a graph with non-negative weights it returns another matrix, where the element (i,j) is the cost for moving from vertex i to j. The parallel implementation is based on the method proposed by *G. J. Katz and J. T. Kider Jr.* [1].

Serial Implementation

The serial implementation is given on the following pseudo-code.

```
1. Initialize dist = adjacency_matrix
2. For k = 0...n-1
3.       For i = 0...n-1
4.            For j = 0...n-1
5.                 if (dist[i][k] + dist[k][j] < dist[i][j])
6.                       dist[i][j] = dist[i][k] + dist[k][j];
7.            endFor
8.       endFor
9. endFor
```

Parallelism Method

A. One cell per thread, no use of shared memory

In the first case, for each k value we start a kernel in CUDA. Each of the threads that wakes up calculates its id (as a function of blockIdx, threadIdx), then calculates the i and j values that correspond to it and executes the if command inside the loop.

Although this method is simple to implement, it is slower than the next, since the time to read and write from and to the device memory is significant.

B. One cell per thread, use of shared memory

In an effort to reduce the time penalty occurred when accessing the device memory, we use the shared memory. It is a block of memory close to each streaming multiprocessor (SM), that has much lower access time. First of all, the *nxn* adjacency matrix is split to *bxb* blocks, so that:

- *n mod b = 0 and*
- bxb = 1024.

The last requirement comes from the fact that each block of the matrix is processed by a block of the GPU grid. The experiments were executed on Nvidia GeForce CTX 480 Compute Capability 2.x. In other words, it is an effort to use all the threads that can be executed within a block.

It is not clear how the algorithm can split the blocks due to the data dependency between cells of other blocks. For the parallelism method we used the algorithm proposed by *G. J. Katz and J. T. Kider Jr.* [2].

More specifically, on the host code we execute a loop *n/b* times, corresponding to each stage, where three different kernels are executed (the phases of the algorithm). We execute three separate kernels, so that in each phase, we load to shared memory only the data that the answer depends from.

In the first phase we transfer to shared memory and compute the result for the primary block (the block in the main diagonal), in the second phase for all the blocks that depend from their data and the primary's block data, and for the third phase phase, the rest.

Of course, we need to take into consideration the synchronization between threads during data transfers.

C. Multiple cells per thread, use of shared memory

In the final case we try to further improve the execution time, simply by assigning multiple cells of the matrix to each thread. Thus, we reduce the overhead associated with waking up a new thread. The number of threads for each block (of the grid) is constant, and equal with the maximum, but we double the dimension of each block of the matrix, the parameter *b*. Now, each thread executes the algorithm for the following cells:

- *(threadIdx.y, threadIdx.x)*
- *(threadIdx.y, threadIdx.x + block_dim/2)*
- *(threadIdx.y + block_dim/2, threadIdx.x)*
- *(threadIdx.y + block_dim/2, threadIdx.x + block_dim/2)*

Indeed, there is improvement in execution time.

Correctness check

About the correctness check two methods were used.

A. Check of Data Access Pattern

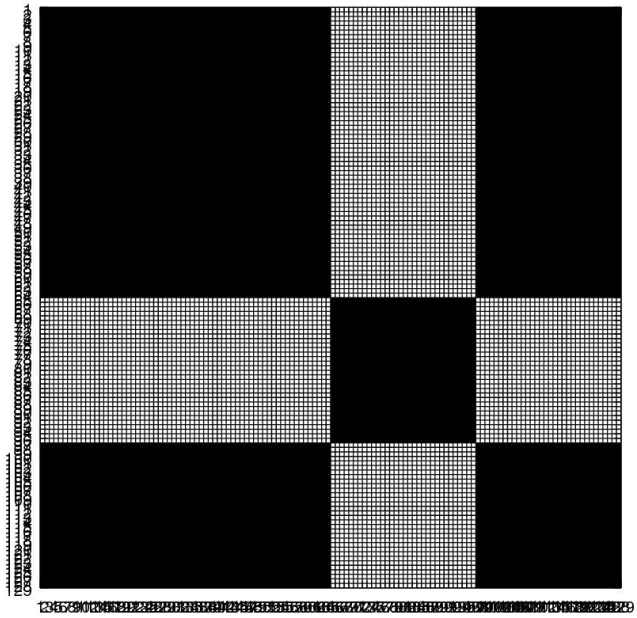
For the parallel algorithms with the multiple stages, we checked that the data access is correct. In order to achieve this, we used an Octave method that prints a boolean matrix. Thus, inserting to a kernel the command:

- `data_matrix[index] = plot_const`

and executing on Octave:

- `plot_matrix( data_matrix == plot_const )`

we can confirm that during each stage, each kernel accesses the correct blocks of the matrix.



Data access to Adjacency Matrix cells for: n=128, stage=3, kernel=2

B. Functional correctness check

The initial serial implementation is checked again simple test cases with a few nodes. Then, each parallel implementation is checked against the serial implementation for a number of test cases, and for all the values of *n*.

Results

A. Execution Time

The experiments were executed on diades system (Intel Xeon CPU @2.5GHz, 8 cores, GPU Nvidia GeForce GTX 480) of A.U.Th. . For each execution we used on core and one GPU. The programs were translated to 64-bit. Each experiment was executed 5 times and the average time was calculated.

The serial CPU implementation has $O(n^3)$ complexity.

Using the first parallel method, we improved the execution time by a factor of 83. It is simple to implement, and the time to access device memory (Peak memory bandwidth 177 GB/s) is the restrictive access.

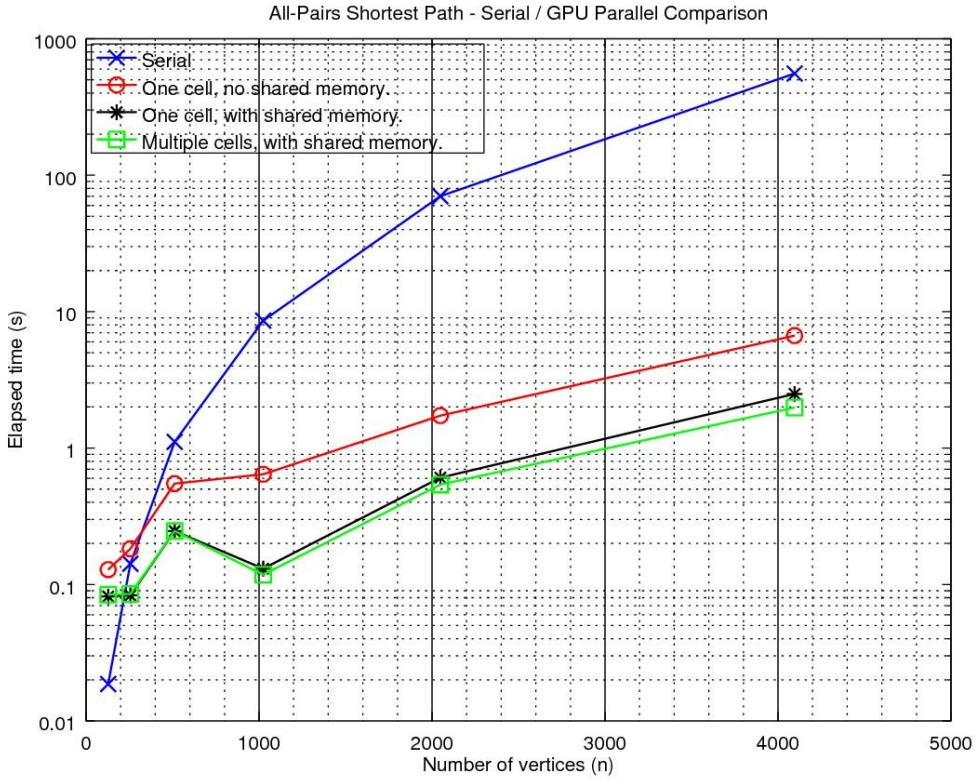
With the second method, we take advantage of the shared memory and improve the execution time 223 times in comparison with the serial implementation and 2.6 times in comparison with the first method.

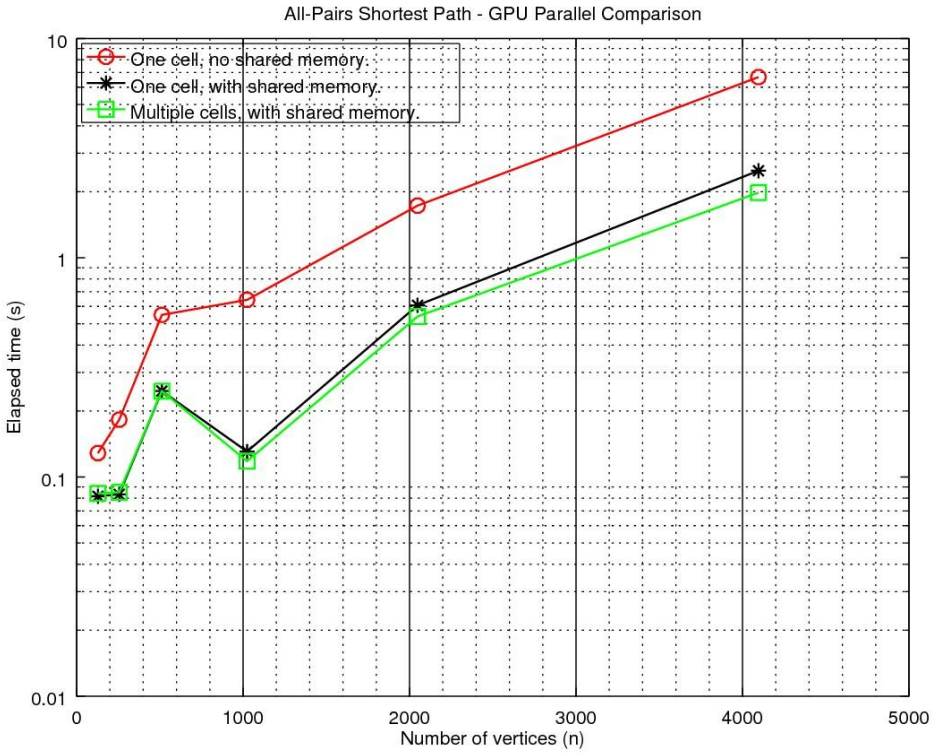
Finally, with the third method, we reduce the overhead of waking up a thread for each cell of the matrix. Each thread computes the answer for 4 cells. The execution time is improved 280 times compared with the serial implementation, 3.3 compared with the first, and 1.25 compared with the previous.

The factors of time improvement stated above are about the case n = 4096.

The following table contains the average execution time (s):

Nodes	**Serial**	**Parallel 1**	**Parallel 2**	**Parallel 3**
128	0.0186	0.12853	0.08168	0.08415
256	0.1418	0.18254	0.08332	0.08516
512	1.1101	0.54881	0.24655	0.24651
1024	8.5908	0.64264	0.13065	0.11806
2048	70.160	1.72863	0.60681	0.53981
4096	556.25	6.66874	2.49429	1.98198





Finally, it is important to note, that although we can observe that for small values of *n* the curve is not monotonous, the experiments were executed again, and this phenomenon did not re-appear. A possible explanation is that there was heavy workload of the GPU during the experiments for *n = 512*. The correct diagram is attached within the project.

Conclusions

We studied the All-Pairs Shortest Path algorithm, and implemented the algorithm proposed on [1]. Moreover, the case where the answer for multiple cells is calculated from one thread was implemented. With the parallel algorithm we realized a significant improvement in execution time.

References

[1] G. J. Katz and J. T. Kider Jr., “All-pairs shortest-paths for large graphs on the GPU”