Analyzing All-Pairs-Shortest-Path Problem on GPUs

Yair Schiff
yair.schiff@nyu.edu
New York University Courant Institute of Mathematical Sciences

***Abstract*— The All-Pairs-Shortest-Path problem, that is finding the shortest distance between all pairs of vertices in a graph, is a well-studied problem with far reaching applications. The Floyd-Warshall (FW) algorithm for solving this problem is a popular solution owing to its favorable run-time. With its computational parallelism, this algorithm can be made to run even faster using highly parallel devices, such as Graphical Processing Units (GPUs), as evidenced by the 100x and higher gains seen when comparing sequential and parallel implementations. This work examines the implications of moving the FW algorithm from a sequential processor to a GPU and compares CUDA and OpenACC implementations for the parallel approach.**

I. Introduction

With implications for many industries and scientific pursuits, graph algorithms comprise an important topic within Computer Science. Two of the more well-established, and closely related, graph problems are the transitive closure problem and the all-pairs-shortest-path (APSP) problem. In the transitive closure problem we would like to find out whether a path exists between any two vertices in a graph. More formally, given an input graph $G(V, E)$, with V representing the set of vertices and E representing the set of edges, we would like to produce a matrix T that is $|V|\text{x}|V|$ with Boolean entries indicating whether a path exists from vertex i to j, represented as the T_{ij} entry of the output matrix. A famous result that solves this problem is known as Warshall's algorithm [1]. An extension of Warshall's algorithm that solves the APSP problem is the Floyd-Warshall (FW) algorithm [2]. In the APSP problem, we are not only interested in whether a path exists between any two vertices $i, j \in V$, but rather what the shortest distance is between all these vertex pairs. Therefore, the output matrix for this problem will not have Boolean 0 or 1 values on the entries of the matrix, but rather will contain the shortest distance between vertex i and j in entry T_{ij}.

While other well-known path algorithms exist such as, Dijkstra's algorithm [3], this paper will focus on the FW algorithm. Specifically, this paper will delve into the performance of the FW algorithm on Graphical Processing Units (GPUs) and analyze the gains relative to a standard, sequential Central processing Units (CPUs), with an emphasis on performance considerations for parallel GPU implementations of the algorithm.

To briefly outline the remainder of this work, Section II provides more detail about the FW algorithm and its run-time complexity. Section III provides a review of related work. Section IV details the parallel solution explored in this paper and Section V describes the setting (e.g. devices used) for the experiments that compare CPU and GPU implementations. Finally Sections VI and VII present the results of the experiments and conclusions, respectively.

II. Background information

The value of the FW algorithm derives from its run-time. One could more naively solve the APSP by trying every combination of paths between vertices, but this solution becomes untenable with its exponential growth. Even a slightly more sophisticated approach of would still result in $\mathcal{O}(n^4)$ run-time (where $n = |V|$). The 'greedy' approach that the FW algorithm takes enables it to cut run-time down to $\mathcal{O}(n^3)$ by solving sub-problems. For each sub-problem, numbered $1..n$, the algorithm finds the shortest path between all vertex pairs $i, j \in V$ using a subset of vertices determined by the sub-problem. That is, for the k^{th} iteration of the algorithm, the algorithm takes the solution to the $k - 1^{st}$ sub-problem and now allows paths to traverse through node k. Thus, every sub-problem k relies on the previous sub-problem $k - 1$ having been solved for all i, j pairs. This reliance on previous sub-problem solutions means that the iterations for k from 1 to n must be executed sequentially.

To further illustrate the algorithm's approach, consider for example the first sub-problem that it solves: $k = 1$. That is, for any (i, j) pair, the shortest distance between those vertices is the minimum of the current shortest distance between i and j and the sum of the distances from i to 1 and from 1 to i:

$$distance(i, j) \leftarrow \min\{distance(i, j), distance(i, 1) + distance(1, j)\}$$

Once this sub-problem 1 is solved for all i, j pairs the algorithm moves onto the next sub-problem, $k = 2$, which will choose the shortest path between i and j from among:

1) $i \rightarrow j$
2) $i \rightarrow 1 \rightarrow j$
3) $i \rightarrow 2 \rightarrow j$
4) $i \rightarrow 1 \rightarrow 2 \rightarrow j$
5) $i \rightarrow 2 \rightarrow 1 \rightarrow j$

As a final note, the FW algorithm is able to accommodate negative edge weights, but does not allow for negative cycles. Were negative cycles to exist, it is not hard to see that the algorithm would find an infinitely short path by repeatedly traversing this negative cycle. The full algorithm details can be found in Algorithm 1. Note that $distance$ is an nxn matrix that is initialized with the edge weights $e, \forall e \in E$ and with $+\infty$ if no edge exists between i and j in E. The

Algorithm 1 FW algorithm

```
procedure FW($dist$)        ▷ input matrix initialized with $e \in E$ or $+\infty$ if no edge exists
    for $k \leftarrow 1..n$ do
        for $i \leftarrow 1..n$ do
            for $j \leftarrow 1..n$ do
                $dist(i,j) \leftarrow$
                $\min\{dist(i,j), dist(i,k) + dist(k,j)\}$
```

algorithm's $\mathcal{O}(n^3)$ run-time comes from the three nested `for` loops. The outer loop solves the n sub-problems and the two inner loops run over all i, j pairs. While the outer loop must run sequentially, the potential for computational speed up arises from the fact that the inner loops can run in any order and can therefore be executed in parallel.

Finally, as it is currently written, Algorithm 1 will only find the shortest distances between (i, j) vertex pairs. To enhance this algorithm to also track the path between i and j we can modify line 5 to be as follows:

```
if $dist(i,j) > dist(i,k) + dist(k,j)$ then
    $dist(i,j) \leftarrow dist(i,k) + dist(k,j)$
    $path(i,j) \leftarrow k$
```

In this enhancement, *path* is an nxn matrix that contains the intermediate node on the path between all (i, j) pairs. If we number all vertices $v \in V$ from $1..n$, then the *path* matrix is initialized with j for all $e \in E$ and with -1, if no edge exists. To recover the solution path, one can use a recursive method that traverses from i to j by following the paths to intermediate nodes.

III. Literature review

Given its proclivity towards parallelization, there have been several works that explore implementations of the FW algorithm on GPUs. Inspection of the algorithm reveals that for each sub-problem, i.e. for each iteration of the most outer loop, there is no restriction on the ordering of the i and j indices of the inner two loops. Therefore, each index of the distance matrix can be computed independently (and in parallel) within an iteration of the outer loop. The most natural approach towards parallelization is then to simply move the distance comparison from line 5 of Algorithm 1 to a kernel function that will be executed on a GPU device by independent threads. This is the approach adopted by [4], where the authors find an average speed up by a factor of 3 compared to the CPU implementation described in Algorithm 1 for graphs of vertices ranging from 1000 to 4000. The authors of [4] also explore a different approach where only n threads (as opposed to n^2) are executed on the GPU, with each thread executing n iterations of the most inner loop. However, this approach is found to slow down the GPU gains.

Other implementations depart more significantly from the original FW algorithm. The first such departure is seen in a block-tiled variation of the algorithm, introduced in [5]. This approach aims to maximize cache efficiency by executing k iterations of the FW algorithm in three steps. The original nxn matrix representing edge weights between all $i, j \in V$ pairs of the graph is split into k^2 equally sized tiles/blocks. The tiles on the diagonal are known as the pivot blocks. In the first phase of the algorithm, FW is run on the pivot block. In the second phase, it is run on all blocks that share a row and column with the pivot block. In the third phase, the remaining blocks are calculated. Starting in the top left corner of the matrix, the pivot block is moved along the diagonal throughout the k iterations, until the entire graph has been processed k times and the shortest distances for all (i, j) pairs have been calculated. All three phases of the first two of the k iterations are shown in Figure 1. In both rows of the figure, the left most grid demonstrates the processing of the pivot block. The middle grid represents processing of the row and column blocks of the pivot block. The right grid represents processing of the remaining blocks. This tiled approach was implemented on a GPU in [6]. The authors there were able to use the improved cache usage to beat the results of [4] and demonstrated how this approach allows for the processing of graphs that exceed GPU device memory. An extension of this approach to hybrid CPU-GPU systems is found in [7].

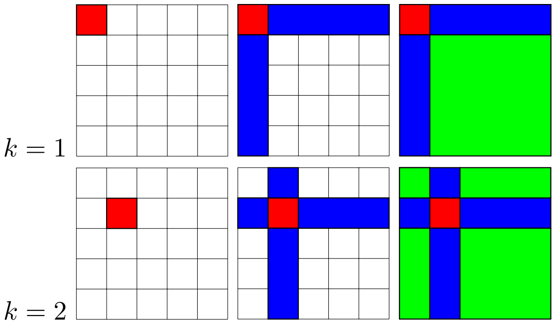

Fig. 1. Tiled FW Algorithm: First two Iterations; Phases 1-3 (Left to Right)

A more involved variation that parallelizes the FW algorithm is found in [8]. The authors of this work also process the graph in steps, however they first cut the graph into disconnected components by identifying a *cut set* of vertices whose removal creates these disconnected components. Interim shortest paths are first found for vertex pairs within components and then 'true' shortest paths are found by allowing for paths to cross across components. The authors execute their algorithm on multiple-GPU clusters gaining even greater efficiency and allowing for even larger graphs (i.e. up to a million vertices) to be processed.

IV. Proposed solution

While the various approaches described above offer a broad range of potential implementations for parallelizing the FW algorithm, this paper will focus on the more straightforward approach from [4], in order to more closely focus on the implications of moving the APSP problem to the GPU. The proposal parallelizes the inner two loops, as seen in Algorithm 2. From a CUDA programming perspective, the outer loop that iterates over k will be placed on the host

Algorithm 2 FW algorithm: Parallel

```
procedure FW_PARALLEL($dist$)
    for $k \leftarrow 1..n$ do
        forall $(i, j) \in V$ do in parallel
            $dist(i, j) \leftarrow$
            $\min\{dist(i, j), dist(i, k) + dist(k, j)\}$
```

machine, while the contents of the parallel `for` loop will be in the kernel that is launched on the GPU device. Using a single stream will ensure that iterations of k from 1 to n will be executed sequentially, while each thread in the GPU will correspond to an index in the nxn $dist$ matrix, which is initialized in a manner equivalent to the sequential implementation of the FW algorithm.

V. Experimental Setup

To compare sequential and parallel implementations of the FW algorithm, several settings were tested, ranging over various matrix sizes. For a more like-to-like comparison, the same input matrix was provided in all settings, with data taken from CAIDA AS Relationship database [9], as in [6]. To see where GPU gains are greatest, matrix input sizes ranging from $n = 500$ to $n = 5,000$ were used (for each size, the algorithm was run five times and statistics presented below represent an average of these five runs). To gain better insight into the GPU performance, the command line profiling tool `nvprof` was used. In this comparison the CPU used was a Two Intel Xeon E5-2660 (2.60 GHz - 40 cores) and the GPU that ran the parallel version of the algorithm was a GeForce GTX TITAN X. Finally, the GPU implementation was also compared to an OpenACC multi-core, parallel version that was run on a Four AMD Opteron 6272 (2.1 GHz - 64 cores) machine.

In addition to these experiments, the CUDA compiled GPU code that is used to compare against the CPU was also compared to an OpenACC implementation of the parallelized FW algorithm to see if there are additional optimizations that the OpenACC compiler automatically captured that were not capitalized on in the CUDA compiled version of the code. Here too various matrix input sizes were tested, as well as various GPU architectures, namely the GeForce GTX Titan X, the GeForce GTX Titan Z, and the GeForce GTX Titan Black.

VI. Results & Discussion

A. GPU vs. CPU/Multi-core

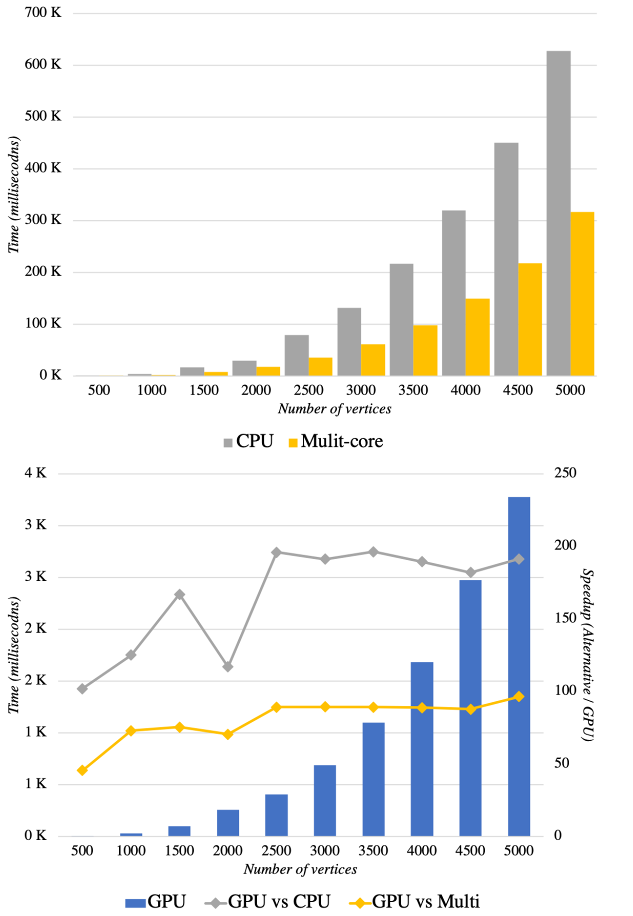

Fig. 2. Comparison of CPU and Multi-Core (top) vs. GPU (bottom) implementations

In Figure 2 we see a comparison of run-time for the parallel implementation of the algorithm on a GPU versus the run-time for the implementation on a multi-core machine and the sequential implementation on a CPU, with input sizes varying on the horizontal axis and time plotted vertically. Note the difference in scale between the primary vertical axes in the GPU plot and the CPU/multi-core plot. Along with the GPU times, we also plot the speedup (that is run-time of CPU/multi-core divided by GPU run-time) on the secondary vertical axis. With gains ranging from 50-100x versus the multi-core implementation and from 100-200x versus the CPU implementation, the parallel GPU implementation is significantly faster across all input sizes. Note also that the GPU shown in this plot is the GeForce GTX Titan X, and that the GPU ran out of memory for graphs larger than 5,000 vertices.

These gains from moving the FW algorithm to the GPU are of course driven by the parallel nature of the algorithm's two inner loops and the computational power of the GPU. However, closer analysis reveals more nuance as to what drives these gains to 100x+ speedups. Looking at the kernel code which is reproduced in Algorithm 3 (excluding the data boundary check present in the actual code), we see a highly simple kernel. Two things are noteworthy about this kernel function. The first is the minimal branch divergence that exists. Even if some threads in a warp do update their

Algorithm 3 FW algorithm: Kernel

```
procedure FW_PARALLEL_KERNEL($dist, k$)
    $i \leftarrow$ thread's vertical index
    $j \leftarrow$ thread's horizontal index
    $dist(i, j) \leftarrow \min\{dist(i, j), dist(i, k) + dist(k, j)\}$
```

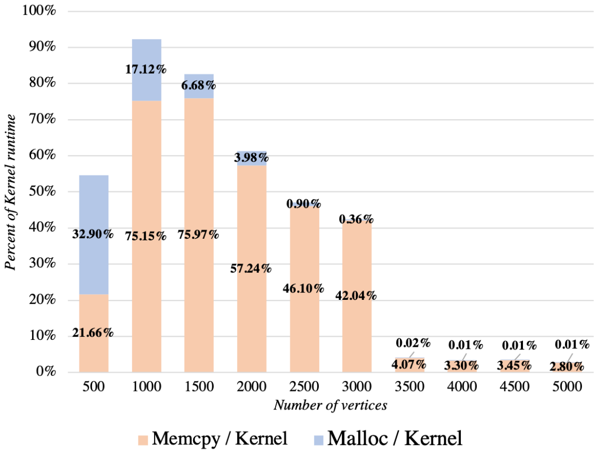

Fig. 3. cudaMalloc & cudaMemcpy as percentage of Kernel run-time

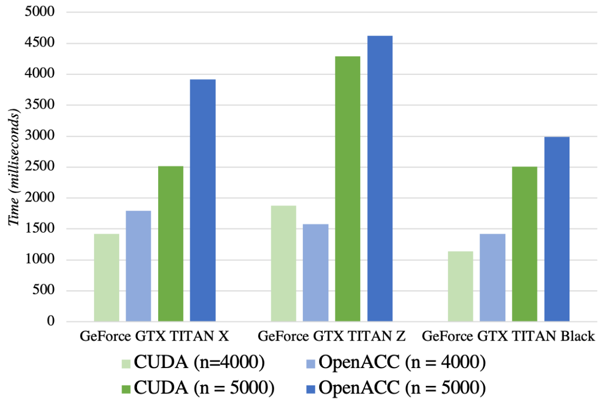

Fig. 4. CUDA vs. OpenACC GPU implementations

distance matrix location and some do not, this is a one line update which minimizes the added penalty of threads diverging. Using the `nvprof` tool with the `metrics` flag on the GeForce GTX Titan X for matrices of input size 1,000 and 4,000, revealed that indeed branch divergence was maximally efficient, with `nvprof` reporting 100% `Branch Efficiency` and `Warp Execution Efficiency`.

The second source of efficiency present in this implementation is the memory coalescing of indexes into the distance matrix. Since reads of the distance matrix differ only by the j coordinate which is determined by `threadIdx.x`, consecutive threads access consecutive memory locations of the distance matrix (given the GPU memory's row major convention). This memory coalescing better utilizes the GPU's memory bandwidth by allowing it to supply multiple entries of the distance matrix at once. However, despite this efficiency, the need to compare $dist(i, j)$ with $dist(i, k) + dist(k, j)$, does present a significant memory inefficiency. Unlike the memory accesses for the (i, j) pairs, those that rely on a k index are not coalesced. This was also seen in use of `nvprof` referenced above, which revealed that memory throughput was only about one sixth of its peak for the GeForce GTX Titan X GPU. However, these non-coalesced memory accesses are inevitable with the implementation presented in Algorithm 3. Use of shared memory would not alleviate this issue, as the data brought from $dist(i, k)$ and $dist(k, j)$ is only used once and the same non-coalesced accesses that are seen here would also be present in bringing these indices into shared memory. To truly coalesce these accesses, the algorithm itself would need to be modified and would more closely reflect tiled and phased implementations seen in other works, such as [6].

B. *Understanding GPU Performance*

Beyond understanding the overarching factors that led to the large speedups seen in Figure 2, it also worth examining why the speed ups tend to increase as large matrix sizes are provided to the algorithm. To do so, the `nvprof` tool was again utilized to highlight which aspects of the code were contributing most to the overall run-time. Figure 3 examines the run-time of the `cudaMalloc` and `cudaMemcpy` APIs (used respectively to allocate memory on the GPU and transfer memory to and from the host to the GPU) as a percentage of kernel run-time. The graph shows that for graphs with 3,000 and fewer vertices a large portion of the program's overall execution time is comprised of overhead activity, ranging from 50-90% of kernel run-time. For example, for the 1,000 vertex graph, we see that overhead activity takes almost as long as the actual execution of the algorithm in the kernel function. However, from 3,500 vertex graphs and up, we see a sharp decline in the contribution of memory allocation and movement to the overall run-time of the program. This corresponds to the jumps in speed ups seen in the bottom graph of Figure 2, indicating that greater gains are achieved when these overhead activities are amortized across larger computations.

C. *CUDA vs. OpenACC*

The final experiment compared CUDA and OpenACC implementations of the parallel algorithm on the GPU. Figure 4 shows the run-times of these two implementations across three GPU architectures for graphs of 4,000 and 5,000 vertices. The comparable run-times for both CUDA and OpenACC validate the analysis of branch divergence and memory access from above. Namely, this plot indicates that for the parallel version of the FW algorithm in Algorithm 2 the kernel in Algorithm 3 performs close to optimally, and barring more structural changes to the actual algorithm, there are not any significant compiler optimizations that would benefit the kernel implementation.

VII. Conclusions

The experiments conducted in this work underscore the gains attainable by parallelizing the seminal FW algorithm with execution on a GPU, even when using a relatively simple parallel approach with respect to the original algorithm. Moreover, in comparisons to a parallel multi-core implementation, the computational efficiency of GPUs is evident with 50x+ speed ups. Complementing the gains from parallelization are the limited branch divergence and memory coalescence for the main read accesses of the distance matrix, which further render the parallel implementation highly efficient compared to the sequential FW algorithm. Comparison of the CUDA implementation with an OpenACC GPU version of the code confirms that the kernel code presented in Algorithm 3 is near optimal for the parallel version of FW explored in this work. Finally, analysis of the overhead activities involved in transporting a problem from a CPU to a GPU, highlight that larger problem sizes benefit more greatly from parallelization and GPU execution, as the overhead costs or better amortized across computations. Extensions of this work would explore multi-GPU implementations of the FW algorithm and experiment with different memory management and data transfer schemes (e.g. unified virtual addressing, GPU-to-GPU communication, etc.) that could affect performance.

References

[1] S. Warshall, “A theorem on boolean matrices,” *Journal of the ACM (JACM)*, vol. 9, no. 1, pp. 11–12, 1962.

[2] R. W. Floyd, “Algorithm 97: shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[3] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[4] P. Harish and P. Narayanan, “Accelerating large graph algorithms on the gpu using cuda,” in *International conference on high-performance computing*. Springer, 2007, pp. 197–208.

[5] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, “A blocked all-pairs shortest-paths algorithm,” *Journal of Experimental Algorithmics (JEA)*, vol. 8, pp. 2–2, 2003.

[6] G. J. Katz and J. 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*. Eurographics Association, 2008, pp. 47–55.

[7] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, “Blocked united algorithm for the all-pairs shortest paths problem on hybrid cpu-gpu systems,” *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 12, pp. 2759–2768, 2012.

[8] G. Chapuis, H. Djidjev, R. Andonov, S. Thulasidasan, and D. Lavenier, “Efficient multi-gpu algorithm for all-pairs shortest paths,” in *IPDPS 2014*, 2014.

[9] CAIDA, “The CAIDA AS Relationships Dataset, 20181101,” 2018. [Online]. Available: http://www.caida.org/data/active/as-relationships/