




2008

All-Pairs Shortest-Paths for Large Graphs on the GPU

Gary J. Katz
University of Pennsylvania

Joseph T. Kider
University of Pennsylvania, kiderj@seas.upenn.edu

Follow this and additional works at: <http://repository.upenn.edu/hms>

 Part of the [Engineering Commons](#), and the [Graphics and Human Computer Interfaces Commons](#)

Recommended Citation

Katz, G. J., & Kider, J. T. (2008). All-Pairs Shortest-Paths for Large Graphs on the GPU. *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH '08)*, 47-55. <http://dx.doi.org/10.2312/EGGH/EGGH08/047-055>

This paper is posted at ScholarlyCommons. <http://repository.upenn.edu/hms/180>
For more information, please contact repository@pobox.upenn.edu.

All-Pairs Shortest-Paths for Large Graphs on the GPU

Abstract

The all-pairs shortest-path problem is an intricate part in numerous practical applications. We describe a shared memory cache efficient GPU implementation to solve transitive closure and the all-pairs shortest-path problem on directed graphs for large datasets. The proposed algorithmic design utilizes the resources available on the NVIDIA G80 GPU architecture using the CUDA API. Our solution generalizes to handle graph sizes that are inherently larger than the DRAM memory available on the GPU. Experiments demonstrate that our method is able to significantly increase processing large graphs making our method applicable for bioinformatics, internet node traffic, social networking, and routing problems.

Disciplines

Computer Sciences | Engineering | Graphics and Human Computer Interfaces

All-Pairs Shortest-Paths for Large Graphs on the GPU

Gary J. Katz^{1,2} and Joseph T. Kider Jr¹

¹University of Pennsylvania, ²Lockheed Martin

Abstract

The all-pairs shortest-path problem is an intricate part in numerous practical applications. We describe a shared memory cache efficient GPU implementation to solve transitive closure and the all-pairs shortest-path problem on directed graphs for large datasets. The proposed algorithmic design utilizes the resources available on the NVIDIA G80 GPU architecture using the CUDA API. Our solution generalizes to handle graph sizes that are inherently larger than the DRAM memory available on the GPU. Experiments demonstrate that our method is able to significantly increase processing large graphs making our method applicable for bioinformatics, internet node traffic, social networking, and routing problems.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture
Graphics Processors Parallel processing G.2.2 [Graph Theory]: Graph algorithms

1. Introduction

Graphics hardware continues to improve every year. Recent years witnessed a dramatic increase in the computational resources available on GPUs. The hundreds of GFLOPS offered on modern graphics architecture allow the developer to program a broad range of general purpose algorithms on the GPU [OLG*07]. This architecture permits many operations to achieve a substantial speedup over current general purpose processors.

GPGPU algorithms efficiently map to the stream programming model inherent in the GPU design. The stream model assumes an arithmetic intensive algorithm processing in parallel on every input producing a corresponding output. Sengupta [SHZO07] argues a GPU stream model is extremely effective for a small, bounded neighborhood of inputs, however many appealing problems necessitate more data intensive support. Data intensive GPGPU algorithms suffer from two significant problems. First, there is considerably less on board memory accessible to GPUs. For example, NVIDIA's Tesla C870 computing processor has 1.5 GB GDDR3 SDRAM of total dedicated memory and the Nvidia 9800 GX2 has 1 GB. For many algorithms to work effectively on real world problems, GPGPU algorithms must handle nontrivial solutions for datasets larger than the GPU's accessible DRAM memory. Secondly, GPU reads and writes to on-board RAM are relatively expensive operations com-

pared to a CPU's operations on main memory since there is no cache located between the threads and DRAM. A single GPU memory operation therefore has high associated latency. GPGPU algorithms must hide memory latent operations with solutions that efficiently utilize the cache's high bandwidth potential and threading capabilities with well-aligned memory accesses. If the GPU utilizes the cache's high bandwidth potential through massive parallelism correctly, it has an overall advantage compared to a CPU.

In this paper, we will first present a blocked (tiled) formulation of the transitive closure of a graph that is highly parallel and scalable. Second, we leverage this solution to solve the Floyd-Warshall (FW) algorithm on the GPU to find the shortest paths between all pairs of vertices in a graph, extending Venkataraman et al. [VSM03] single core CPU cache efficient algorithm to exploit the parallelism and efficiency of the GPU using CUDA. Our approach handles graph sizes larger than the on-board memory available to the GPU and additionally, breaks the graph in a nontrivial on-chip shared memory cache efficient manner to increase performance. Our technique provides: a substantial 60–130x speedup over a standard CPU solution $O(v^3)$, a 45–100x speedup to a blocked (tiled) CPU implementation specified by Venkataraman et al. [VSM03], and furthermore, our method provides a speedup of 5.0–6.5x relative to a standard GPU implementation proposed by Harish

and Narayanan [HN07], and a 2x–4x speed up compared to the best auto-generated highly tuned CPU code generated by Han et al. [HFP06].

The paper is organized as follows: Section 2 describes prior work of transitive closure, all-pairs shortest path problem (APSP) algorithms and implementations, and a brief overview of work to manage memory on the GPU. In Section 3, we review Warshall’s Transitive Closure and Floyd’s APSP algorithms. Section 4 provides details of implementing our shared memory cache efficient algorithm and how we generalize the solution to operate on graphs larger than the GPU’s on-board memory. Section 5 demonstrates the strength of our method by measuring the performance over previously published standard CPU implementations, blocked (tiled) CPU implementations, Han et al. [HFP06] program generator tuned code, and previous GPU implementations on both synthetic and real world dataset (CAIDA AS Relationships Dataset). Finally, we discuss the implementation issues on the GPU and future directions.

2. Prior Work

The all-pairs shortest path problem (APSP) is well studied in prior work due to its significance affecting a variety of problems: network routing, distributed computing, and bioinformatics to name a few. A straightforward solution solves the problem by running a single-source shortest-pair algorithm $|V|$ times, once for each vertex as the source [CLRS01]. If one utilizes a sophisticated min-priority queue with a Fibonacci heap data structure, this yields a $O(V^2 \lg(V) + VE)$, the asymptotically best time complexity. However, if the graph is dense with negative weight edges, the running time is $O(V^3)$, utilizing the the Floyd-Warshall algorithm, making this solution impractical for graphs with large vertex sizes.

Warshall [War62] formulated how to compute the transitive closure of Boolean matrices. Later, Floyd [Flo62] developed an algorithm, the Floyd-Warshall algorithm (FW), based on Warshall’s theorem to solve the all-pairs shortest-path problem. Early on, Foster described simple techniques to parallelize the FW algorithm in his book [Fos95]. Diamond and Ferencz [DF] talked about comparing straightforward parallel implementations, one of which used blocking to speedup calculations. Venkataraman et al. [VSM03] proposed a more complex blocked version of Floyd’s all-pairs shortest-path algorithm to better utilize the cache for large graphs and provided a detailed performance analysis. They achieved a speedup of 1.6 and 1.9 over the unblocked basic implementation’s counterpart.

Penner and Pranna [PP06] proposed cache-efficient implementations of transitive closure and the Floyd-Warshall algorithm showing a 2x improvement in performance over the best compiler optimized implementation on three different architectures. Han et al. [HFP06] created a program generator to derive the variant of the FW algorithm which is

most efficient for a CPU. The generator uses tiling, loop unrolling, and SIMD vectorization to produce the best solution for a given CPU by tuning and testing a variety of parameters.

Furthermore, Subramanian et Al. [STV93] showed a theoretical foundation for the prospect of efficient parallelization of APSP algorithm and devises an approach that yields a runtime complexity of $O(\log^3 n)$, with n the number of processors in planar layered digraphs. Bondhugula et al. [BDF*06] proposed a tiled parallel Field Programmable Gate Arrays (FPGAs)-based implementation of the APSP problem. Han and Kang [HK05] presented a vectorized version of the FW algorithm that improved performance by 2.3 and 5.2 times of speed-up over a blocking version.

Micikevicius [Mic04] proposed using graphics hardware to solve APSP. Each pixel corresponded to a unique distance matrix entry, so the fragment shader was used to perform the FW algorithm. Additionally, using the 4 component RGBA vector capability on the legacy hardware (NVIDIA 5900 ULTRA), they were able to produce further performance gains. Harish and Narayanan [HN07] proposed using CUDA to accelerate large graph algorithms (including APSP) on the GPU, however they implemented only a basic version of Floyd-Warshall algorithm. Therefore, they are limited by the device memory limits and cannot handle graphs larger than the GPU’s DRAM. Additionally, Harish and Narayanan do not perform any improvements to handle the unique structure of the G80’s on-chip shared memory cache layout. Utilizing a shared memory coherent blocking method, we display a 5x-6.5x increase in performance over Harish and Narayanan’s proposed CUDA GPU implementation.

Both the relatively small cache size and the significant latency issue are inherent in stream computing on the GPU. The G80 has only 16 kB of fast on-chip shared memory. Govindaraju et al. [GLGM06] analyzed scientific algorithms on older NVIDIA 7900 GTX GPUs (that did not contain shared memory) and presented models to help performance. Furthermore, they updated the cache-efficient algorithms for scientific computations using graphics processing units and hardware. They showed performance gains of 2-10x for GPU-based sorting, fast Fourier transform and dense matrix multiplication algorithms [GM07].

Early GPU work centered on developing cache-efficient algorithms for matrix-matrix multiplication using a variety of blocking techniques [LM01, HCH, FSH04, NVI07]. Additionally, Lefohn et al. proposed a template library for defining complex, random-access graphics processor (GPU) data structures [LSK*06] to help increase performance of a variety of algorithms. More recently, Deschizaux and Blanc used a simple tiling method to store the frequencies for their plane in their wave propagation kernel [DB07] in CUDA.

3. Overview

In this section, we give a short overview of modern GPUs, a review of the transitive closure and the all-pairs shortest-path algorithms, and describe a memory layout of the graph using GPU memory.

3.1. G80 Architecture and CUDA

In recent years, many scientific and numeric GPGPU applications found success due to graphics hardware's streaming data-parallel organizational model. With the introduction of NVIDIA G80 GPU architecture, the graphics pipeline now features a single unified set of processors that function as vertex, geometry, and fragment processors. Additionally, the release of the Compute Unified Device Architecture (CUDA) API [NVI07] on the G80 architecture allows developers to easily develop and manage general purpose scientific and numerical algorithms without formulating solutions in terms of nontrivial shaders and graphic primitives.

The GPU serves, to an extent, as a coprocessor to the CPU programmed through the CUDA API. A single program known as a kernel is compiled to operate on the GPU device to exploit the massive data parallelism inherent in Single Instruction, Multiple Data (SIMD) architecture. Groups of threads then execute the kernel on the GPU. Threads are organized into blocks which allow efficient sharing of data through a high-speed shared memory region (16 kB in size) on the G80 architecture accessible to the programmer directly through CUDA. Shared memory is shared among threads in a block, facilitating higher bandwidth and overall performance gains. Therefore, algorithms must intelligently manage this ultra fast shared memory cache effectively. This will fully utilize the data parallelism capabilities of graphics hardware and alleviate any memory latency that data intensive algorithms suffer from on the GPU.

3.2. Warshall's Transitive Closure

Let $G = (V, E)$ be a directed graph with vertex set $|V|$, the *transitive closure* (or *reachability*) problem endeavors to locate whether a directed path between any two given vertices in G exists. A vertex i is *reachable* by vertex j iff there exists some directed path from i to j in G . The *transitive closure* of G is defined by the directed graph, $G^* = (V, E^*)$ which contains the same vertex set $|V|$ as G however has an edge $(i, j) \in E^*$ iff there is a path from vertex i to vertex j in G . The *transitive closure* of a graph provides the list of edges at any vertex showing paths reaching other vertices thereby answering reachability questions. Algorithm 1 [CLRS01] solves for the *transitive closure* of G .

3.3. Floyd-Warshall APSP Algorithm

Again, let $G=(V,E)$ be a directed graph with $|V|$ vertices. Now, we solve APSP by the Floyd-Warshall algorithm us-

```

TRANSITIVE CLOSURE (G)
1:  n ← |V[G]|
2:  for i ← 1 to n
3:    do for j ← 1 to n
4:      do if i = j or (i, j) ∈ E[G]
5:        then tij(0) ← 1
6:        else tij(0) ← 0
7:    for k ← 1 to n
8:      do for i ← 1 to n
9:        do for j ← 1 to n
10:       do tijk ← tijk-1 ∨ (tikk-1 ∨ tkjk-1)
11:  return T(n)
    
```

Algorithm 1: Pseudocode for Transitive Closure.

ing a dynamic programming approach on a directed graph [CLRS01]. (Note, the Floyd-Warshall algorithm has a running time of $\Theta(|V|^3)$).

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{weight of edge}(i, j) & \text{if } i \neq j \text{ and } (i, j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E. \end{cases} \quad (1)$$

Let d_{ij}^k be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \dots, k\}$. Where $k=0$, we have $d_{ij}^0 = w_{ij}$. A recursive definition from the above formulation is given by:

$$d_{ij}^k = \begin{cases} w_{ij} & \text{if } k = 0; \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases} \quad (2)$$

We can now solve the Floyd-Warshall Algorithm [CLRS01]:

```

FLOYD-WARSHALL (W)
1:  n ← rows[W]
2:  D(0) ← W
3:  for k ← 1 to n
4:    do for i ← 1 to n
5:      do for j ← 1 to n
6:        do dij(k) ← min(dij(k-1), dik(k-1) + dkj(k-1))
7:  return D(n)
    
```

Algorithm 2: Pseudocode of the Floyd-Warshall Algorithm.

3.4. Memory overview of the graphs

We store the nodes and edges of the graph in a matrix form. The vertices of the graph are represented by the unique indices along $n \times n$ matrix (n represents the number of nodes in our graph). An edge between 2 vertices is specified by a 1 connecting the x axis index of the matrix to the y axis index.

We partition the $n \times n$ matrix W that holds our graph G into

sub-matrices of size $B \times B$, where B is the *blocking factor* such that there are now $(n/B)^2$ blocks. Now our blocked version of FW will operate B iterations of the outer-loop in the simple FW Algorithm (Algorithm 2) via the GPU algorithm described in Section 4 below. We load the necessary blocks (described below in section 4) into GPU memory, synchronize the threads, perform the FW algorithm, then return result to global memory.

4. Implementation

In this section, we discuss the design and implementation of our tiled FW algorithm on NVIDIA GPUs utilizing the CUDA programming API.

4.1. Algorithm

Our goal in this section is to revise the original straightforward FW algorithm into a hierarchically parallel method that can be distributed, in parallel, across multiple multi-processors on the GPU, and furthermore across multiple GPUs. Our method implements a unique extension of Venkataraman et al. [VSM03] adapting for efficiency on GPUs using CUDA. The method begins by examining the data conflicts that exist in a distributed FW algorithm. The original FW algorithm can not be broken into distinct sub-matrices that are processed on individual multi-processors because each sub-matrix needs to have terms across the entire dataset. If we examine the FW algorithm presented (Algorithm 2) we see that each element d , is updated through examining every other data element in matrix W , making data partitioning impossible. The algorithm negates this problem by carefully choosing a sequential ordering of groups of sub-matrices that use previously processed sub-matrices to determine the values of the current sub-matrices being processed.

To begin the algorithm, we partition the matrix into sub-blocks of equal size. In each stage of the algorithm, a primary block is set. The primary block for each stage is along the diagonal of the matrix, starting with the block holding the matrix value $(0,0)$.

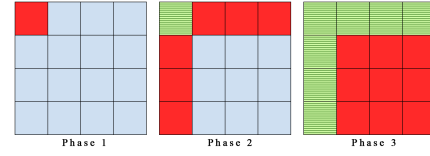
The primary block holds the sub-matrix holding values from (p_{start}, p_{start}) to (p_{end}, p_{end}) , where

$$p_{start} = \frac{\text{primary block number} \times \text{matrix length}}{\text{number of primary blocks}} \quad (3)$$

$$p_{end} = p_{start} + \left(\frac{\text{matrix length}}{\text{number of primary blocks}} \right) - 1 \quad (4)$$

Each stage of the algorithm is broken into three passes. In the first pass we only compute values for the primary block. The computations are completed in one CUDA block, and therefore only one multi-processor is active during this pass. The block computes FW where i , j , and k range from p_{start} to p_{end} .

Phases when block $(1,1)$ is the self-dependent block



Phases when block (t,t) is the self-dependent block

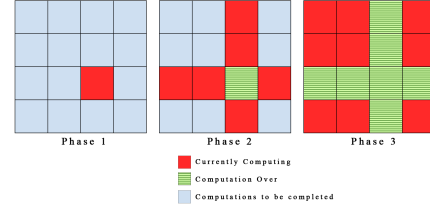


Figure 1: A tiled FW algorithm proposed in section 4.

For the first primary block, we can view the sub-block as its own matrix for which we compute the FW algorithm. At the completion of the first pass, all pairs have been located between nodes p_{start} through p_{end} . In the second pass we would like to compute all sub-blocks that are only dependent upon the primary block and themselves. By careful examination of the memory accesses, we can see that all sub-blocks that share the same row or the same column as the primary block are only dependent upon their own block's values and the primary block. This can be shown by noticing that for current blocks that share the row of the primary block, k ranges from p_{start} to p_{end} , j ranges from p_{start} to p_{end} , and i ranges from c_{start} to c_{end} , where c_{start} and c_{end} are the start and end indices for the current block in the x direction.

We can therefore show that the indices of d range from $[c_{start} \rightarrow c_{end}, p_{start} \rightarrow p_{end}]$, $[c_{start} \rightarrow c_{end}, p_{start} \rightarrow p_{end}]$, $[p_{start} \rightarrow p_{end}, p_{start} \rightarrow p_{end}]$ for d_{ij} , d_{ik} , and d_{kj} respectively. Since the indices of the current block range from $[c_{start} \rightarrow c_{end}, p_{start} \rightarrow p_{end}]$, and the indices of the primary block range from $[p_{start} \rightarrow p_{end}, p_{start} \rightarrow p_{end}]$, we can clearly see that the second pass only needs to load the current block and the primary block into memory for updates to the current block. A similar proof can be used to show that the memory usage for the column blocks only need to load the primary block and the current column block. In the second pass we have computed the all pairs solution for all blocks sharing the same row or column as the primary block for values of k from p_{start} to p_{end} , with each block being computed in parallel on a separate multi-processor.

In the third pass we would like to compute the values of the remaining blocks for ranges of k from p_{start} to p_{end} . We perform the same methods as in the second pass by examining the memory accesses of the block. For each remaining block, we note that its values of d range from $[c_{start}^i \rightarrow c_{end}^j, c_{start}^j \rightarrow c_{end}^j]$, $[c_{start}^i \rightarrow c_{end}^i, p_{start} \rightarrow p_{end}]$,

$[p_{start} \rightarrow p_{end}, c_{start}^i \rightarrow c_{end}^j]$ for d_{ij} , d_{ik} , d_{kj} respectively, where c_{start}^i , c_{end}^i , c_{start}^j , c_{end}^j are the current i start, current i end, current j start and current j end, respectively. We note that $[c_{start}^i \rightarrow c_{end}^i, p_{start} \rightarrow p_{end}]$ is the block with the column of the current block and the row of the primary block, while $[p_{start} \rightarrow p_{end}, c_{start}^j \rightarrow c_{end}^j]$ has the row of the current block and the column of the primary block. Both of these blocks were computed in pass 2. Figure 2 shows a visualization of this data access.

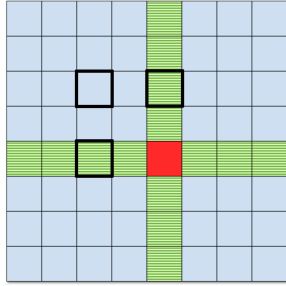


Figure 2: Visualization of data access.

With the completion of pass 3, all cells of the matrix have their all-pairs solution calculated for k values ranging from p_{start} to p_{end} . The primary block is then moved down the diagonal of the matrix and the process is repeated. Once the three passes have been repeated for each primary block, the full all pairs solution has been found.

4.2. Memory and Grid Layout

In each pass of the algorithm the primary block number is sent as an input to the kernel. The individual thread blocks must determine which sub-matrix they are currently processing and determine what data to load into its shared memory. In the first pass, this is a straightforward task. The primary block is the current block and its data is the only memory being loaded. Each thread can load the data point corresponding to their own thread id, and save that value back to global memory at the end of the pass.

In the second pass, the primary block is loaded with the current block, with each thread loading one cell from each block. At the end of the algorithm, each thread saves its cell from the current block back to global memory.

The grid lay out in the second pass determines the ease of processing. For a data set with n blocks per side, there are $2 \times (n - 1)$ blocks processed in parallel during the second pass. We layout these blocks into a grid of size $n - 1$ by 2. The first row in the grid processes the blocks in the same row as the primary block while the second row in the grid processes the blocks in the same column as the primary block. The block y value of 1 or 0 specifies the row or column attribute and can be used as a conditional to specify how data is indexed.

We also need to make sure the blocks are correctly indexed by the x index. The blocks need to make sure to 'skip' over the primary block. This can be accomplished by the following equation.

$$Skip\ center\ block = \min\left(\left(\frac{blockIDx.x + 1}{center_block_ID + 1}\right), 1\right) \quad (5)$$

This value is added to the $blockIdx.x$ when referencing the current block to load. Skip center block will round to 0 when the center block index is greater than the $blockIdx.x$ and be set to 1 when the center block is less than or equal to the $blockIdx.x$. In shared memory, the two blocks are laid out side-by-side as shown in figure 3.

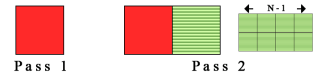


Figure 3: Visualization of memory configuration of Pass 1 and Pass 2.

The third pass contains a grid size of $n - 1$ by $n - 1$. As in the second pass, we must skip over the primary block and any row or column blocks computed in the second pass. This is accomplished by the following equations that share the same format as skip center block above.

$$Skip\ center\ block(x) = \min\left(\left(\frac{blockIDx.x + 1}{center_block_ID + 1}\right), 1\right) \quad (6)$$

$$Skip\ center\ block(y) = \min\left(\left(\frac{blockIDx.y + 1}{center_block_ID + 1}\right), 1\right) \quad (7)$$

These values are added to the $blockIdx$ in the x and y directions. The shared memory layout in the third pass must contain three blocks, the current block, the column block and the row block as shown in figure 4.



Figure 4: Visualization of memory configuration of Pass 3.

The maximum size of the sub-block is limited to the minimum of 1/3 the total shared memory size (16 kB for the 8800) and the maximum threads per block (512 for the 8800).

4.3. Adapting transitive closure to APSP

The FW algorithm can be used to find the transitive closure of the graph and to find the all-pairs-shortest-path. The transitive closure of a graph is a Boolean result describing whether a path exists between two nodes. The all-pairs shortest-path provides the shortest path between those two nodes for every pair of nodes in the graph. The transitive closure begins with a matrix of size n by n . As an input, a 1 is placed in all cells where two nodes are connected by an edge and a zero is placed in all other cells. Transitive closure replaces line 6 of Algorithm 2 with the following:
 $d_{ij} = d_{ij} \parallel (d_{ik} \& d_{kj})$

The matrix can be represented by a variety of different data types, int, bool, char, etc. Our implementation used ints to remove bank conflicts although a less memory intensive data type could be used necessitating each thread to process multiple cells at a time to reduce conflicts.

To determine the all pairs shortest path we must update our data types and change our processing equation. Each cell in the matrix must now hold two values $[c_1, c_2]$. The cell represents the shortest path as the combination of two sub paths. These sub paths are represented as i to c_1 and c_1 to j , where (i, j) is the index of the matrix cell. At the end of the algorithm, the shortest path between any two nodes a, b can be found by accessing the cell $[a, b]$ and then recursively gathering the sub paths between $[a, c_1]$ and $[c_1, b]$. A cell with no sub paths has a value of -1 for c_1 . The value of c_2 contains the number of edges between a and b . The algorithm is initialized by placing $[-1, 1]$ in every cell (i, j) where there is a connection, and $[0, 0]$ for all other cells. Our implementation uses float2s to hold the cell values to ease indexing, but a vector of two ints could also be used.

<pre> 1 : if((distance(i,k) ≥ 1 && distance(k,j) ≥ 1) && ((distance(i,k) + distance(k,j) < distance(i,j) (distance(i,j) == 0) 2 : (matrix[i, j] = (k, distance(i,k) + distance(k,j))) </pre>

Algorithm 3: Pseudocode for path checking.

The processing equation for all pairs shortest path must first check if a path exists between d_{jk} and d_{kj} . Next, it must check if the new path is shorter than the existing path d_{ij} , if one exists. We illustrate this process in pseudo-code above in Algorithm 3. The primary methods of the algorithm remain the same across transitive closure and all pairs shortest path. The increased memory size for all pairs shortest path reduces the number of nodes that can be processed on a single GPU and changes the optimal block size to 18×18 .

4.4. Adapting the algorithm across multiple GPUs

Our algorithm can be easily adapted across multiple GPU's. CUDA's API allows a programmer to specify programming multiple GPU's in parallel by spawning different threads on

the CPU. To ease processing, the dataset is broken into pages of data, similar to the blocks of data used in shared memory. The word page will refer to a square of data residing in global memory, while a block will refer to a square of memory residing in shared memory. Each page is equal sized, with equal numbers of pages in the x and y directions of the graph. The maximum page size is equal to memory on the graphics card. The page size varies according to the size of the dataset, the number of GPUs, and the GPU memory size.

The previously described tiled all-pairs algorithm is performed on each page in the same manner that the Floyd-Warshall algorithm is performed on each block in shared memory. The first pass is performed in parallel on each GPU, ensuring the necessary information is available to all cards for pass 2 without having to perform additional memory transfers. The pages processed during the second pass are split among the available GPUs. Once all the blocks in the second pass have completed and their results have been returned, the third pass is performed, being evenly split among the GPUs. As described in the initial algorithm, the primary page is then advanced and the process is repeated. The page processing uses the same tiled FW algorithm as performed on each multiprocessor shown in figure 1.

In the shared memory algorithm, each pass requires a different number of blocks to process the algorithm. In the multi-GPU algorithm the same memory layout occurs. The first pass of the algorithm only requires the primary page. The second pass requires the primary page along with the page being processed. The third pass requires the row and column pages associated with the page being processed.

The first, second and third sub-passes performed in shared memory must correctly load data from the right page into shared memory. In the first major pass the tiled algorithm remains the same because only the primary page is being loaded. In the second major pass, for each minor pass, d_{ij} and d_{ik} from equation 2 perform their lookups from the current page being processed, while d_{kj} is loaded from the primary page. In the third major pass, d_{ij} accesses the page being processed, while d_{ik} is loaded from the column page and d_{kj} is loaded from the row page.

The multi-GPU algorithm described above naively distributes pages to be processed among the GPUs. It does not schedule the GPUs according to the information already loaded on the GPU. The algorithm may also have idle GPUs while some GPUs wait for others to finish processing their blocks in the second and third passes. This results in reduced occupancy for the GPUs along with non-optimal memory transfers. A more efficient method would be to have a separate thread that schedules the GPU threads to process the most optimal page.

We tested our multi-GPU algorithm using NVIDIA's CUDA API on a Windows PC with a 2.8 GHz Xeon(TM) CPU and 2 GB of RAM using 2 NVIDIA GeForce 8800 GT graphics cards.

5. Experimental results

This section analyzes the performance of our algorithm on both synthetic and real-world datasets. To demonstrate the strength of our approach, we provide 2 comparisons. First, we analyze the performance of our SM cache efficient GPU algorithm on various large graph sizes between a standard CPU, cache-efficient CPU implementation, and highly tuned CPU implementation generated by the SPIRAL. Furthermore, we compare our technique to the latest GPU implementation and demonstrate a significant performance gain.

5.1. Comparison of Synthetic Dataset with Prior CPU-based and GPU based FW Algorithms

Figure 5 compares our method to a standard CPU implementation of FW and a shared memory cache-efficient CPU implementation proposed by Venkataraman et al. [VSM03]. We simply generated a synthetic dataset by randomly specifying direct paths between vertices in our initial matrix to illustrate the arbitrary cases our method handles. We see that the CPU versions grow at a rate of $O(n^3)$. For large graphs these calculations take an unreasonable amount of processing time. The cache-efficient CPU implementation has 1.6-1.9x increase in performance as reported by Venkataraman et al. [VSM03]. However, the curve still grows at a rate that makes larger graphs impractical to run on the CPU still. Our method greatly outperforms both CPU versions, thereby making processing large graphs more practical and efficient.

We tested our single-GPU algorithm using NVIDIA's CUDA API on a Windows PC with a 2.8 GHz Xeon(TM) CPU and 2 GB of RAM using 1 NVIDIA QUADRO FX 5600 graphics card.

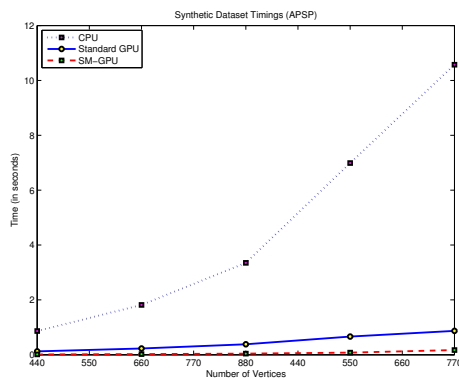


Figure 5: Performance of our SM cache-efficient GPU implementation compared to standard CPU implementation and a cache-efficient CPU implementation proposed by Venkataraman et al. [VSM03] for APSP.

Figure 6 compares our method to a standard GPU implementation proposed by Harish and Narayanan [HN07]. Our

method has a 5.0-6.5x increase in performance due to a better shared memory cache-efficient algorithm. Additionally, since we block the initial graph into parts, our algorithm generalizes to work on arbitrary sized graphs, unlike Bondhugula et al. [BDF*06] which is limited by the size of on-board memory on the GPU.

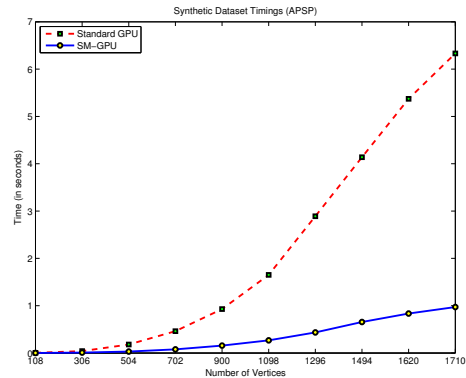


Figure 6: Performance of our SM cache-efficient GPU implementation compared to standard GPU implementation proposed by Harish and Narayanan [HN07] for APSP.

5.2. Comparison of Synthetic Dataset with Prior GPU based Transitive Closure Algorithm

Figure 7 compares our method to a standard GPU implementation of the Transitive Closure algorithm. Our method has a 7.5-8.5x increase in performance.

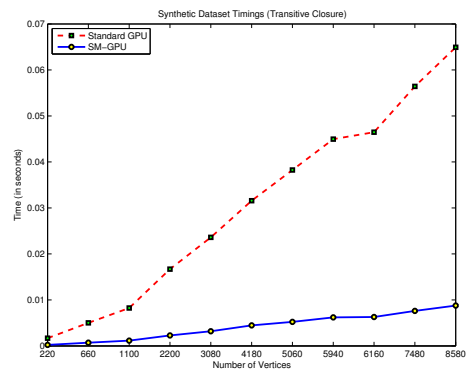


Figure 7: Performance of our SM cache-efficient GPU implementation compared to standard GPU implementation of the Transitive Closure algorithm.

5.3. Comparison of AS Relationships Dataset with Prior CPU-based and GPU based Algorithms

To demonstrate the strength and the applicability of our method, we tested our solution on a real world dataset.

We used the CAIDA Route Views BGP autonomous system (AS) links annotated with inferred relationships dataset [CAI08]. We preprocessed the data once on the CPU to generalize all the indexing to our straightforward node indexing scheme on the GPU by mapping the node indices to a simpler data type rather than encode all the information CAIDA provides. For example we only created directed edges for 1 relationship (if AS1 is a customer of AS2) between the nodes to generate our test graph. We do this since, inherently, the number of nodes, not edges, affect the overall running time. However to formulate other relationships requires a trivial text processing step. For the connections between the first 1000 AS nodes the standard GPU implementation time was 1371.95 ms. Our shared memory cache efficient implementation ran in 189.29 ms showing a 7.26x speedup.

Additionally, we compared our method to the best auto-generated highly tuned CPU code generated by Han et al. [HFP06]. Figure 8 compares our multi-GPU results for large graphs. Han et al.'s auto-generator analyzes various variants of the APSP algorithm, tuning for the best blocking size, loop unrolling setting, tiling, and SIMD vectorization combined with hill climbing search to produce the best code for a given platform [HFP06]. We implemented our tests using NVIDIA's CUDA API on a Windows PC with a 2.8 GHz Xeon(TM) CPU and 2 GB of RAM. We received slightly different timings than the authors since Xeon processors have only a 512 kB cache and the paper used a 3.6 GHz Pentium 4 which has a 1 MB cache. We note Han et al.'s times in table 1, in addition to the times we produced.

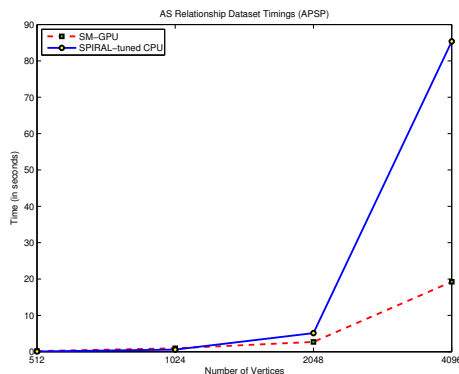


Figure 8: Performance of our SM cache-efficient GPU implementation compared to the best variant of Han et al.'s [HFP06] tuned code.

6. Conclusions and Future Work

In this paper, we described a parallel blocked version of the transitive closure and Floyd's all-pairs shortest-paths algorithms that handle two key issues that data dependant algorithms suffer from on the GPU. Namely, our algorithm is both shared memory cache efficient and generalizes to

graph sizes larger than the GPU's on-board memory effectively. Using the GPU to find a solution to these problems increases performance over previously proposed solutions and does not require exotic distributed computing hardware. Our method adapts graphics hardware which is more common among standard users and cheaper, thus providing a wider market of usage of our implementation.

Having a more efficient APSP implementation that is practical for the common user has potential for a variety of interesting practical applications. To list a few: geo-location algorithms, internet node routing, and social network applications all can potentially benefit from our proposed method since APSP and transitive closure are fundamental graph algorithms that function as building blocks of numerous applications.

One extension to our implementation would be to construct a function to print the vertices on the shortest path. CLRS [CLRS01] suggests either computing the predecessor matrix Π from the resultant matrix, or computing a predecessor matrix Π "online" just as the Floyd-Warshall algorithm computes the output matrix.

7. Acknowledgments

The authors would like to thank NVIDIA for providing graphics hardware to help test our implementation. The authors would like to thank Norm Badler, Alla Safonova, and Amy Calhoun for all their insightful help.

References

- [BDF^{*}06] BONDHUGULA U., DEVULAPALLI A., FERNANDO J., WYCKOFF P., SADAYAPPAN P.: Parallel FPGA - Based All-Pairs Shortest-Paths in a Directed Graph. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium* (apr 2006).
- [CAI08] CAIDA: *The CAIDA AS Relationships Dataset, 20080107 - 20080317*. 2008.
- [CLRS01] CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C.: *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
- [DB07] DESCHIZEAUX B., BLANC J.-Y.: *Imaging Earth's Subsurface Using CUDA*. In *GPU Gems 3*. Hubert Nguyen, (Ed.). Addison-Wesley, 2007, ch. 38, pp. 831–850.
- [DF] DIAMENT B., FERENCZ A.: Comparison of parallel apsp algorithms.
- [Flo62] FLOYD R. W.: Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345.
- [Fos95] FOSTER I.: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Nodes	Memory Size	(single) SM-GPU	(Dual) SM-GPUs	Our Best tuned CPU	Han et al.'s best CPU
64	32 kb	0.0014 s	0.087 s	0.00029 s	0.00021 s
128	131 kb	0.0025 s	0.088 s	0.0018 s	0.0015 s
256	524 kb	0.0077 s	0.096 s	0.0097 s	0.0098 s
512	2.09 MB	0.0349 s	0.185 s	0.0644 s	0.052 s
1024	83.8 KB	0.2301 s	0.950 s	0.578 s	0.419 s
2048	3.35 MB	1.7356 s	2.689 s	5.114 s	3.51 s
4096	134 MB	13.72 s	19.244 s	85.373 s	29.97 s
9216	679 MB	158.69 s	149.74 s	na	na
10240	838 MB	216.4 s	151.92 s	na	na
11264	1.015 GB	1015.7 s	353.78 s	na	na

Table 1: Detailed performance of our SM cache-efficient GPU implementations compared to the best variant of CPU tuned code. We tested our single SM-GPU algorithm using NVIDIA's CUDA API on a Windows PC with a 2.8 GHz Xeon(TM) CPU (512 kB cache) and 2 GB of RAM with 1 NVIDIA QUADRO FX 5600 card (1.56 GB GDDR3 SDRAM). We tested our multi-GPU variant using 2 GeForce 8800 GT (512 MB GDDR3 SDRAM) graphics cards to demonstrate how our code scales to graphs larger than 1 GPU's on-board memory. Han et al. [HFP06] published results ran on a 3.6 GHz Pentium 4 with 1 MB of cache and 2 GB of RAM and produced different timings since the CPU architecture was slightly different then ours.

- [FSH04] FATAHALIAN K., SUGERMAN J., HANRAHAN P.: Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Graphics Hardware 2004* (Aug. 2004), pp. 133–138.
- [GLGM06] GOVINDARAJU N. K., LARSEN S., GRAY J., MANOCHA D.: A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM, p. 89.
- [GM07] GOVINDARAJU N. K., MANOCHA D.: Cache-efficient numerical algorithms using graphics hardware. *Parallel Computing* 33, 10–11 (2007), 663–684.
- [HCH] HALL J., CARR N., HART J.: Cache and bandwidth aware matrix multiplication on the GPU. UIUC Technical Report UIUCDCS-R-20032328.
- [HFP06] HAN S.-C., FRANCHETTI F., PÜSCHEL M.: Program generation for the all-pairs shortest path problem. In *Parallel Architectures and Compilation Techniques (PACT)* (2006), pp. 222–232.
- [HK05] HAN S.-C., KANG S.-C.: Optimizing all-pairs shortest-path algorithm using vector instructions. <http://www.ece.cmu.edu/~pueschel/teaching/18-799B-CMU-spring05/material/sungchul-sukchan.pdf>, 2005.
- [HN07] HARISH P., NARAYANAN P. J.: Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing* (2007), vol. 4873 of *Lecture Notes in Computer Science*, Springer, pp. 197–208.
- [LM01] LARSEN E., MCALLISTER D.: Fast matrix multiplies using graphics hardware. *Supercomputing, ACM/IEEE 2001 Conference* (10–16 Nov. 2001), 43–43.
- [LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics* 25, 1 (Jan. 2006), 60–99.
- [Mic04] MICIKEVICIUS P.: General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '04, June 21–24, 2004, Las Vegas, Nevada, USA, Volume 3* (2004), CSREA Press, pp. 1359–1365.
- [NVI07] NVIDIA CORPORATION: NVIDIA CUDA compute unified device architecture programming guide. <http://developer.nvidia.com/cuda>, Jan. 2007.
- [OLG*07] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26, 1 (Mar. 2007), 80–113.
- [PP06] PENNER M., PRASANNA V. K.: Cache-friendly implementations of transitive closure. *ACM Journal of Experimental Algorithms* 11 (2006), 185–196.
- [SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan Primitives for GPU Computing. In *Graphics Hardware 2007* (Aug. 2007), ACM, pp. 97–106.
- [STV93] SUBRAMANIAN S., TAMASSIA R., VITTER S. J.: *An Efficient Parallel Algorithm for Shortest Paths in Planar Layered Digraphs*. Tech. rep., Durham, NC, USA, 1993.
- [VSM03] VENKATARAMAN G., SAHNI S., MUKHOPADHYAYA S.: A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics* 8 (2003), 2.2.
- [War62] WARSHALL S.: A theorem on boolean matrices. *J. ACM* 9, 1 (1962), 11–12.