Graph Algorithms on GPUs using CUDA

CSE 6230 Course Project

Fall 2013

Shruti Padamata (GT id – 902915013)
Niveditha Raveendran (GT id - 903009612)

Contents

Introduction

Large graphs involving millions of vertices are common in many practical applications and are challenging to process. Algorithms for analyzing sparse relationships represented as graphs provide crucial tools in many computational fields ranging from genomics to electronic design automation to social network analysis. Practical-time implementations using high-end computers are reported but are accessible only to a few. Graphics Processing Units are parallel processors offering high FLOPS/sec for low cost. NVIDIA's CUDA framework makes it practical to take advantage of this resource for high performance computing by providing a scalable programming model and a stable and comprehensible machine model to program against. Even with the tools provided by CUDA, making full use of the inherent processing power of a GPU can be a challenge, since the speed of a computation may not be limited by the speed of the processing units, but can also be limited by memory bus bandwidth or problems with thread scheduling. Graphics Processing Units (GPUs) of today have high computation power and recent work has demonstrated the plausibility of GPU graph traversal. Through this project we attempt to port two fundamental graph algorithms - breadth first search and all-pairs shortest path to GPUs, using CUDA.

The All Pairs Shortest Path Problem

Overview

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. This is analogous to the problem of finding the shortest path between two intersections on a road map: the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment.

Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like Mapquest or Google Maps. For this application fast specialized algorithms are available.

If one represents a nondeterministic abstract machine as a graph where vertices describe states and edges describe possible transitions, shortest path algorithms can be used to find an optimal sequence of choices to reach a certain goal state, or to establish lower bounds on the time needed to reach a given state. For example, if vertices represent the states of a puzzle like a Rubik's Cube and each directed edge corresponds to a single move or turn, shortest path algorithms can be used to find a solution that uses the minimum possible number of moves.

In a networking or telecommunications mindset, this shortest path problem is sometimes called the min-delay path problem and usually tied with a widest path problem. For example, the

algorithm may seek the shortest (min-delay) widest path, or widest shortest (min-delay) path. A more lighthearted application is the games of "six degrees of separation" that try to find the shortest path in graphs like movie stars appearing in the same film. Other applications, often studied in operations research, include plant layout, robotics, transportation, and VLSI design.

Formalization of All Pairs Shortest Path Problem

Given a graph G = (V,E) where V is a set of nodes and E is a set of directed edges, every node in the graph is labeled by an integer i € [0, n−1] where n = |V | (n = |V | is the cardinality of the set V), and an edge in E is defined by a unique ordered pair of integers (i, j) with i, j € [0, n − 1]. It is assumed that there is at most one directed edge connecting two nodes and therefore, the graph has no more than |E| = m <= n^2 edges. To represent G, an adjacency matrix, A € Z^{nxn} is used. That is, an entry in row i and column j in matrix A, a_{ij} € Z, stands for the cost to reach node j from node i through the edge (i, j) € E. Also, a_{ii} = 0 for every i € [0, n − 1] i.e., there is no cost to stay in one node and if there is no direct edge from node i to node j, then a_{ij} = ∞.

Shortest Paths - Matrix Multiplication based Algorithm

An elementary path of length k from node i to j (in a graph G) is a sequence of k edges connecting i to j with no nodes repeated. An elementary path is denoted as the set of edges P_k(i, j). For the all-pairs shortest-paths problem on a graph G = (V, E), all subpaths of a shortest path are shortest paths. Suppose that the graph is represented by an adjacency matrix W = (wij). Consider a shortest path p from vertex i to vertex j, and suppose that p contains at most m edges. Assuming that there are no negative-weight cycles, m is finite. If i = j, then p has weight 0 and no edges. If vertices i and j are distinct, then we decompose path p into i -> k -> j, where i -> k is p'which contains at most m – 1 edges. Since subpaths of shortest paths are shortest paths themselves, $\mathscr{E}$(i,j) = $\mathscr{E}$(i,k) + w_{kj}.

Let l_{ij}^{m} be the minimum weight of any path from vertex i to vertex j that contains at most m edges. When m = 0, there is a shortest path from i to j with no edges if and only if i = j.

Thus, l_{ij}^{0} = 0 if i == j and ∞ otherwise. For m ≥ 1, we compute l_{ij}^{m} as the minimum of $l_{ij}^{(m-1)}$ (the weight of the shortest path from i to j consisting of at most m - 1 edges) and the minimum weight of any path from i to j consisting of at most m edges, obtained by looking at all possible predecessors k of j. Thus, we recursively define l_{ij}^{m} = min ($l_{ij}^{(m-1)}$, $\min_{1 \le k \le n}$ {$l_{ik}^{(m-1)}$ + w_{kj}})
The latter equality follows since wjj = 0 for all j.

Computing the shortest path weights bottom up

Taking the matrix W = (w_{ij}) as input, a series of matrices L(1), L(2),..., L(n-1) is computed, where for m = 1, 2,..., n - 1, L_{ij} = (l_{ij}). The final matrix L(n-1) contains the actual shortest-path weights. It is to be noted that l_{ij}^{1} = w_{ij} for all vertices i, j e V , and so L(1) = W. The heart of the algorithm is the following procedure, which, given matrices L(m-1) and W, returns the matrix L(m). That is, it extends the shortest paths computed so far by one more edge.

EXTEND-SHORTEST-PATHS (L, W)

```
1 n ← rows[L]
2 let L_ij' = ( l_ij' ) be an n × n matrix
3 for i ← 1 to n
4       do for j ← to n
5             do l_ij' ← ∞
6                   for k ← 1 to n
7                         do l_ij' ← min { l_ij', l_ik^(m-1) + w_kj}
8 return L'
```

Running time is $\Theta(n^3)$ due to the three nested for loops.

If observed carefully, this function is similar to matrix multiplication where minimum operator replaces sum and addition operator replaces multiplication.

Since we are interested in L(n-1), it could be computed by repeated squaring of L(1) to obtain the following sequence : $L_{(1)} = W$, $L_{(2)} = W^2$, $L_{(4)} = W^4$ and so on.

The following procedure computes the above sequence of matrices by using this technique of repeated squaring.

FAST-ALL-PAIRS-SHORTEST-PATHS (W)

```
1 n ← rows[W]
2 L(1) ← W
3 m ← 1
4 while m < n - 1
5       do L(2m) ← EXTEND-SHORTEST-PATHS(L(m), L(m))
6             m ← 2m
7 return L(m)
```

Running Time Analysis

The running time of FAST-ALL-PAIRS-SHORTEST-PATHS is $\Theta(n^3 \lg n)$ since each of the $\lceil \lg(n - 1) \rceil$ matrix products takes $\Theta(n^3)$ time.

Implementation

Graph Generation and Representation

Adjacency matrix representation is used to represent the input graph. A graph with n nodes is represented in the form of a n x n matrix A where $a_{i,j}$ represents the cost of moving from node i to node j. $a_{i,i} = 0$ for every i e [0, n − 1] –i.e., there is no cost to stay in one node– and if there is no direct edge from node i to node j, then ai,j = ∞. Weights are randomly generated and have a maximum value of 1000 (for ease of implementation), without loss of generality.

Computation

The EXTEND-SHORTEST-PATH function is implemented using 2 dimensional blocking similar to matrix multiplication. The maximum performance was achieved with a block size of 16 * 16 threads , with each thread processing 16 elements. The EXTEND-SHORTEST-PATH kernel is being called $\lceil \lg(n - 1) \rceil$ times.

Fllyod Warshalls Algorithm

Overview

The Floyd–Warshall algorithm is another dynamic programming based algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles) and also for finding transitive closure of a relation R. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices, though it does not return details of the paths themselves.

In the Floyd-Warshall algorithm, the characterization of the structure of a shortest path is different from the one used in the matrix-multiplication-based all-pairs algorithms. The algorithm considers the "intermediate" vertices of a shortest path, where an intermediate vertex of a simple path $p = v_1, v_2,..., v_n$ is any vertex of p other than v_1 or v_n, that is, any vertex in the set $\{ v_2, v_3,..., v_{n-1}\}$.

The Floyd-Warshall algorithm is based on the following observation. Under the assumption that the vertices of G are V = {1, 2,..., n}, consider a subset {1, 2,..., k} of vertices for some k. For any pair of vertices i, j e V, consider all paths from i to j whose intermediate vertices are all drawn from {1, 2,..., k}, and let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2,..., k - 1}. The relationship depends on whether or not k is an intermediate vertex of path p.

If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2,..., k - 1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2,..., k - 1} is also a shortest path from i to j with all intermediate vertices in the set {1, 2,..., k}.

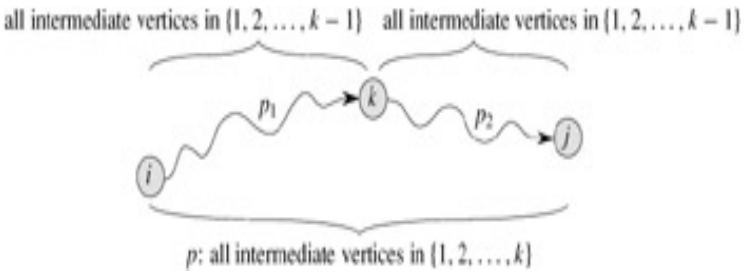


Figure 1 Structure of Path from i to j

If k is an intermediate vertex of path p, then we break p down into p_1 and p_2, where p_1 is a shortest path from i to k with all intermediate vertices in the set {1, 2,..., k}. Because vertex k is not an intermediate vertex of path p_1, we see that p_1 is a shortest path from i to k with all

intermediate vertices in the set {1, 2,..., k - 1}. Similarly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1, 2,..., k - 1}.

A recursive solution to the All Pairs Shortest Path Problem

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,..., k}. When k = 0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $dij^{0} = w_{ij}$. A recursive definition following the above discussion is given by if k = 0, $d_{ij}^{k} = w_{ij}$; $d_{ij}^{k} = \min\{ d_{ij}^{k-1}, d_{ik}^{k-1} + d_{jk}^{k-1} \}$ otherwise.
Since for any path, all intermediate vertices are in the set {1, 2,..., n}, the matrix $D_n = (d_{ij}^{n})$ gives the final answer.

Computing the shortest paths weights Bottom up

Based on recurrence defined above, the following bottom-up procedure can be used to compute the values in order of increasing values of k. Its input is an n × n matrix W. The procedure returns the matrix $D_{(n)}$ of shortest-path weights.

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) + djk^(k-1) }
7 return D(n)
```

Running Time Analysis

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Since each execution of line 6 takes O(1) time, the algorithm runs in time $\Theta(n^3)$.

Naive Implementation

The Floyd Warshall's algorithm was implemented on the CPU as described by the pseudo code above. A Naive GPU implementation was also written where the result of each value of the result matrix is computed by one thread.

Blocked Floyd Warshall Algorithm

The original Fllyod Warshall algorithm shows little scope for parallelization due to its inherent structure. However, a blocked version of the algorithm described by Venkatraman and Sahni in [6] shows potential for parallelization. A parallel implementation of this algorithm has been previously implemented by Katz and Kider in [7].
This algorithm involves partitioning the n x n matrix W that represents the graph into sub matrices of size B x B where B is the blocking factor such that now there are $(n/B)^2$ blocks.

Examining the data conflicts that exist in a distributed FW algorithm presents potential enhancements. The original FW algorithm cannot 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. Each element d, is updated through examining every other data element in matrix W, making data partitioning impossible. This 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. Initially, the matrix is partitioned into sub matrices 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

$$p_{start} = (primary\ block\ number \times matrix\ length)/number\ of\ primary\ blocks$$

$$p_{end} = p_{start} + (matrix\_length/number\ of\ primary\ blocks) - 1$$

Each stage of the algorithm is broken into three passes. In the first pass the values for the primary block are computed. The block computes FW where i, j, and k range from p_{start} to p_{end}.

For the first primary block, the sub-block can be viewed as its own matrix for which the FW algorithm is used. At the completion of the first pass, all pairs have been located between nodes p_{start} through p_{end}. In the second pass all sub-blocks that are only dependent upon the primary block and themselves are computed. By careful examination of the memory accesses, one observes 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.

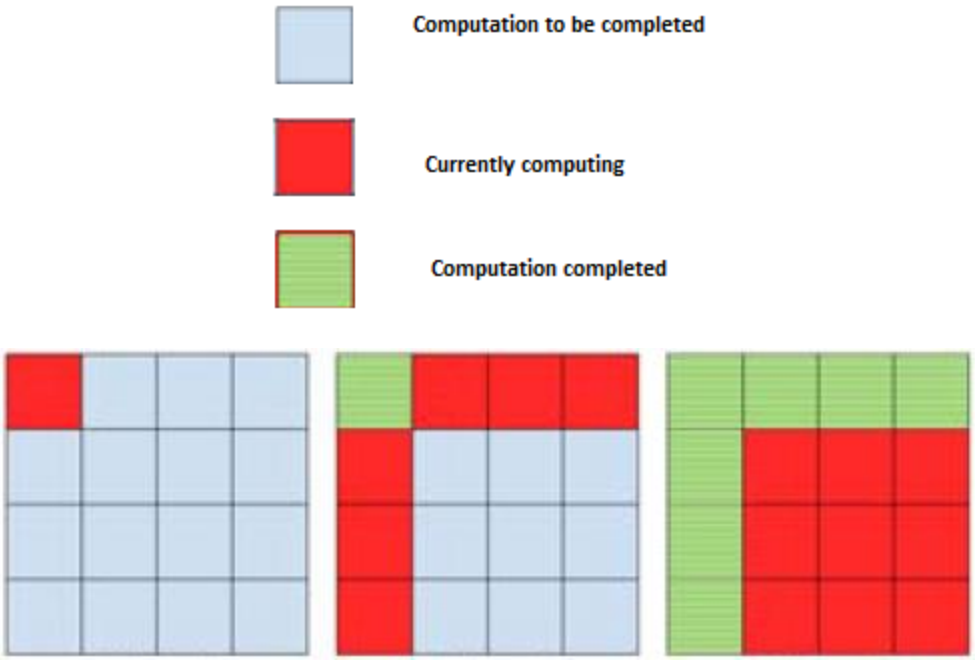


Figure 2 Phases when block (1,1) is the self dependent block

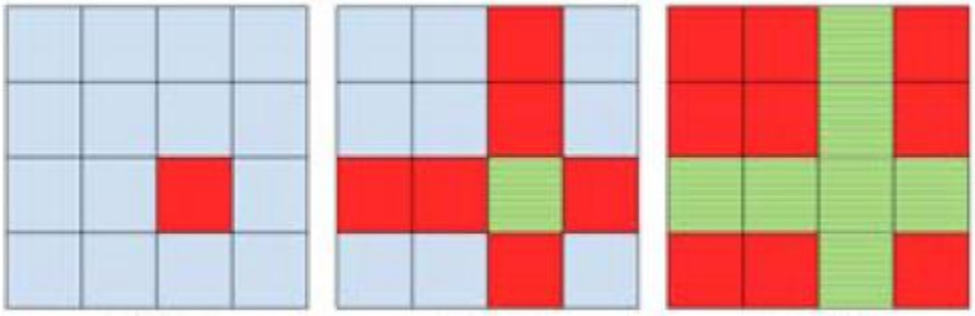


Figure 3 Phases when block (t,t) is the self dependent block

Therefore 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 $[p_{start} \rightarrow p_{end}, p_{start} \rightarrow p_{end}]$, 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 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} has been computed, with each block being computed in parallel by a separate thread block. In the third pass the values of the remaining blocks for ranges of k from p_{start} to p_{end} are computed. The same methods as in the second pass are performed by examining the memory accesses of the block. For each remaining

block, its values of d range from [$c_{start}^{i} \rightarrow c_{end}^{j}$, $c_{start}^{j} \rightarrow c_{end}^{j}$], [$c_{start}^{i} \rightarrow c_{end}^{i}$, $p_{start} \rightarrow p_{end}$], [$p_{start} \rightarrow p_{end}$, $c_{start} \rightarrow c_{end}^{j}$] for d_{ij} , d_{ik}, and 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. 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, [$p_{start} \rightarrow p_{end}$, $c_{start} \rightarrow c_{end}^{j}$] has the row of the current block and the column of the primary block. Both of these blocks were computed in pass 2.

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 all pairs solution has been found.

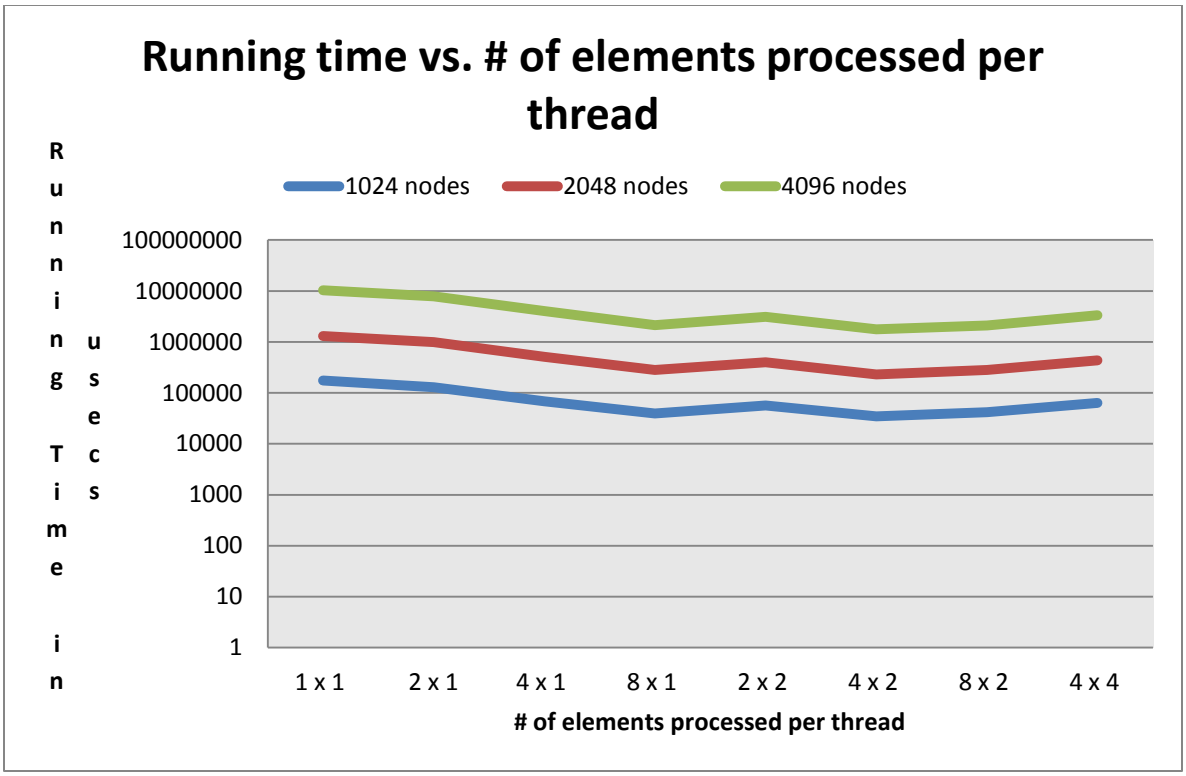
Implementation

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 its 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×(n−1) blocks processed in parallel during the second pass. These blocks are laid out 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. During the third pass, (n-1) x (n-1) thread blocks process 1 block each. It is to be noted that during the second pass, the primary block is not processed and during the third pass, the secondary blocks are not processed.

The initial implementation processed only 1 element per thread. Later, an approach similar to 2 D blocking was done for all three kernels (phase 1, phase 2 and phase 3). Processing more than 1 element seemed to improve performance. The analysis is presented below.

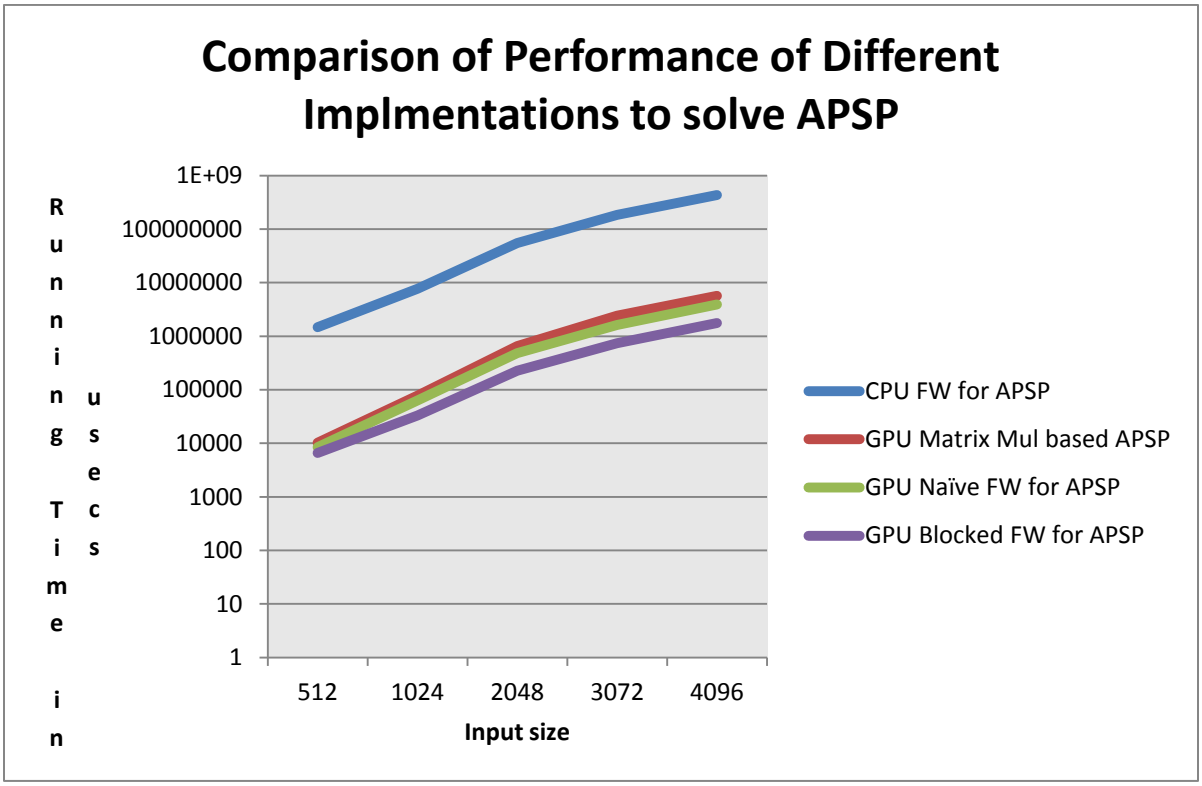
Performance Analysis

The Jinx cluster at Georgia Tech's College of Computing was used for the development and for testing performance. The CPU variant was executed on a single core of an Intel Xeon processor. For the GPU-based implementation, a single Tesla M2090 device was used.



It was observed that the configuration where each thread processed 4 (in x direction) * 2 (in y direction) elements gave the best performance. Therefore, this configuration has been used to compare with other implementations henceforth.

Results



Input Size	CPU FW for APSP Running Time in usecs	GPU Matrix Mul. based APSP Running Time in usecs	GPU Naïve FW for APSP Running Time in usecs	GPU Blocked FW for APSP Running Time in usecs
512	1473223	10199	8510	6558
1024	7735819	77701	62850	33225
2048	55289564	653967	481968	227238
3072	184187355	2404276	1618168	736571
4096	434447112	5666254	3892459	1756861

It is observed that the blocked Floyd Warshall's algorithm with each thread processing 4 x 2 elements give the best performance. It is 200x faster than the CPU version of Floyd Warshall's algorithm. The matrix multiplication based algorithm, even when implemented using the fastest version of matrix multiplication (performing above 600 GFlops/sec) is slower than the naive version of Floyd Warshalls. This result could be attributed to the higher complexity of the matrix multiplication algorithm $\Theta(n^3 \lg n)$ as compared to $\Theta(n^3)$ complexity of Fllyod Warshalls algorithm. Therefore, Floyd Warshall's algorithm scales better to solve APSP for large graphs.

The Blocked GPU implementation of Floyd Warshalls performs 2x faster when compared to the Naive GPU implementation of Floyd Warshalls. This is due to data reuse and intelligent data access patterns performed by the blocked version.

Breadth First Search

In graph theory, breadth-first search (BFS) is a technique for searching in a graph that begins at a root node and inspects all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. Breadth-first search can be used to solve many problems in graph theory like finding the shortest path between any two nodes u and v, and testing a graph for bipartiteness. Algorithms to search a graph in a breadth-first manner have been studied for over 50 years. A variety of parallel BFS algorithms have also been explored.

Given an directed, unweighted graph G(V,E) and a source vertex S, the breadth-first search(BFS) mechanism is used to search for a destination vertex V in G, by progressively exploring the neighboring nodes level-wise, starting from the source vertex S, as shown in the below diagram.

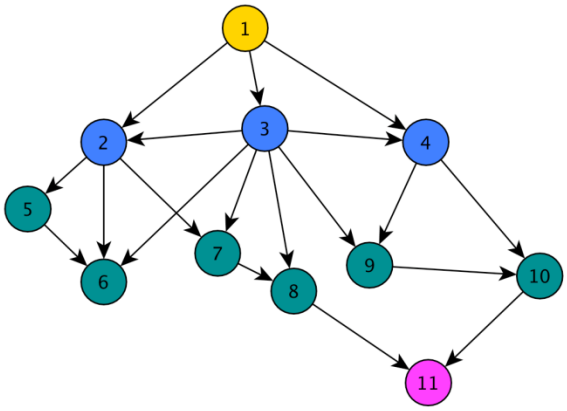


Figure 4 Example for Breadth First Search

The optimal sequential solution for this problem takes O(V +E)time. The basic parallelizations can be applied at the propagation of the search from each vertex at a level and during the search through every neighbor of those vertices. BFS is representative of a class of parallel computations whose memory accesses and work distribution are both irregular and data-dependent. Optimizing memory usage is non-trivial because memory access patterns are determined by the structure of the input graph. [2] presents a GPU implementation using a hierarchical technique to efficiently implement a queue structure on the GPU. [3] presents further fine grained optimizations to improve the memory accesses and the synchronization model. We would be studying these to develop our parallel model of BFS. All the algorithms that we consider here are level-synchronous: each level may be processed in parallel as long as the sequential ordering of levels is preserved.

Graph Generation and Representation

We use un-weighted directed graphs and for simplicity, we identify the vertices v_0 .. v_{n-1} using integer indices 0..n-1. Compressed sparse row(CSR) representation of the graphs is used for the CPU implementation and the optimized GPU implementation, while the naïve GPU implementation uses an edge-list representation.

CSR Representation

The number of edges within sparse graphs is typically only a constant factor larger than the number of vertices 'n'. So, we use the well-known compressed sparse row (CSR) sparse matrix format to store the graph in memory. Consider a graph of the form G = (V, E) containing a set V of n vertices and a set E of m directed edges. The CSR format stores vertices and edges in separate arrays, with the indices into these arrays corresponding to the identifier for the vertex or edge, respectively. The column-indices array C array (or edge array) is sorted by the source of each edge, but contains only the targets for the edges. So, it stores the neighbors for each vertex. The row-offsets R array (or vertex array) stores offsets into the edge array, providing the offset of the first edge outgoing from each vertex. C. C contains m values while R contains n+1 values. The figure below shows a simple graph with its C and R matrices.

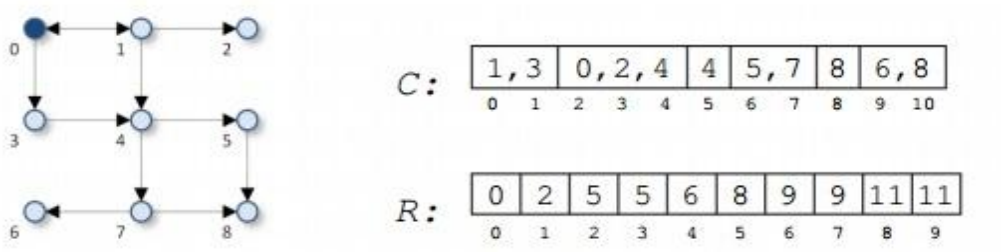


Figure 5 Example for C S R representation

For testing purposes, we generated a partially random adjacency matrix, with a given number of nodes. So, the graph generation function **initGraph** inputs the number of nodes, density of the graph and the maximum degree of any node, and outputs an adjacency matrix A, which contains 1 at (i, j) if there exists an edge from vertex v_i to v_j and 0 otherwise. Self-edges (i, i) are also set as 1 in this matrix. It ensures that the graph is connected so that the performance would not vary drastically for different inputs. The graph is generated with a specified density, but it may not be very exact because of a few extra edges that are added to make the graph connected and the culling of some edges to keep the degree of the vertices within the maximum degree. This matrix is processed in **generateEdgeList** and **generateCR** functions to obtain the edge-list and CSR representations respectively.

Single core CPU implementation

In a Breadth-first model of search, we always see a well defined frontier or boundary between all the vertices that have already been visited and those that have not been visited. The search progressively examines the neighbors of nodes or vertices from a frontier and constructs the next frontier, thus traversing all the connected nodes. The diagram in the introduction shows a simple example with the numbers representing the order of traversal of the nodes from one of the nodes taken as the root or source node.

The CPU implementation circulates the vertices of the graph in the above fashion. We use a FIFO queue that is initialized with the source vertex v_s. Vertices are de-queued iteratively and their neighbors are examined. The distances for the unvisited neighbors are updated and are enqueued for later processing. Thus, each iteration examines the neighbors of the nodes in one frontier and populates the queue with the nodes of the next frontier. The FIFO ordering of the sequential algorithm forces it to label vertices in increasing order of depth. Each depth level is fully explored before the next. Since the graphs are generated randomly, any vertex can be taken as the source node. So, without any loss of generalization, we consider the first node with index 0 as the source node. This is done for all the other implementations discussed as well.

This algorithm performs linear O(m+n) work since each vertex is labeled exactly once and each edge is traversed exactly once. Pseudo-code for this implementation is given below. Refer to the function **cpuBFS** in 'driver.c' for the code.

CPU-BFS

Input: *Vertex set V, row-offsets array R, column-indices array C, source vertex vs*
Output: *Array D[0..n-1] with D[vi] holding the distance from vs to vi*
Functions: *enqueue(val) inserts val at the end of the queue*
dequeue() returns the head element of the queue

```
1 Q := {}
2 for i in 0 .. |V|-1:
3       D[i] := ∞
4 D[s] := 0
5 Q.enqueue(s)
6 while (Q != {}) :
7       i = Q.dequeue()
8       for offset in R[i] .. R[i+1]-1 :
9               j := C[offset]
10 if (dist[j] == ∞)
11      D[j] := D[i] + 1;
12      Q.enqueue(j)
```

Naive GPU implementation

Overview

The simplest parallel BFS algorithms inspect every edge or, at a minimum, every vertex during every iteration. This is one such algorithm that uses an edge-based inspection. At a high level, we operate on the list of edges and for each edge, if one end of the edge has a depth 'd' and the other edge has not been visited, then we set the depth of the other vertex to 'd+1'. We iterate 'h' times where 'h' is the maximum depth of any node in the graph. Thus, in each iteration all the edges can be examined in a parallel fashion.

Implementation details

The graph generated as an adjacency matrix is processed to obtain an edge list representation stored as an array of node numbers. The depth array is initialized with the depth of all nodes as -1 except for the source node whose depth is set to 0. The function **cudaBFS_Edgelist** then takes the edge list and depth array and calls the kernel **BFSNaiveKernel** iteratively, each time setting the depths of all the nodes at a particular distance from the root. Each thread in the kernel is associated with one edge, which checks if the depth of the first vertex corresponds to the depth of nodes being examined in the current iteration and sets the depth of the second vertex to its depth + 1 if its not already set to a valid value. There is no synchronization required between the threads because even when multiple threads discover the same vertex concurrently, they would all set its depth to the same valid value. Finally, a flag variable is used to detect the completion, which is set in the kernel and tested on the host to stop the iterations.

Analysis

The algorithm examines all the vertices in each iteration but sets the depth for each vertex only in a single iteration. In the worst case, this algorithm has a work complexity of O(m*n). In general, the number of edges in a graph is at-least equal to the number of vertices, in which case, the complexity would be $O(n^2)$. Thus this algorithm has a quadratic work complexity and would not scale well for large graphs.

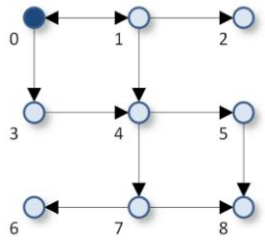
Optimized Linear Work algorithm

Overview

[1] presents a very recent work on GPU BFS parallelization that performs asymptotically optimal linear work. The algorithm does a frontier based search similar to the sequential CPU implementation. It is work-efficient because it tries to minimize the number of visits to each node and only the nodes on the frontier are subject to computation. We tried to follow [1] for getting an optimized implementation, but we do not reproduce the results exactly, as there were many details in the optimizations that were not clearly illustrated in the paper and we

interpreted them as we deemed logical. The author claims that this approach delivers high performance on a broad spectrum of structurally diverse graphs. They have also proposed a multi-GPU implementation, but we explore only the single GPU implementation in this project.

This algorithm processes each frontier in the graph iteratively. In each iteration, first the number of neighbors of each vertex in the frontier is obtained in parallel. These values are computed using R[v+1] – R[v] for any vertex 'v' and are stored in an array. A prefix sum is performed on this array in order to obtain the total size required for the next frontier and the starting indices for copying the neighbors of each vertex in the frontier. Using the output of the prefix sum, the neighboring vertices of each vertex are copied into the array allocated for the new frontier. But this list for the new frontier may contain duplicates and some vertices that have already been visited. These extra vertices need to be culled from the list. This culled list would serve as an input to the next iteration. The process is repeated until the depths of all vertices are set to a valid value. The following example shows the vertex frontiers for a simple graph and their expansion in each iteration:



Iteration	Vertex Frontier	Next Frontier
1	{0}	{1, 3}
2	{1, 3}	{0, 2, 4, 4}
3	{2, 4}	{5, 7}
4	{5, 7}	{6, 8, 8}
5	{6, 8}	{}

Implementation details

The depth array is again initialized with the value -1 for every vertex except for the source vertex whose depth is set to 0. The initial frontier is set to contain only the source node '0'. The following sub-sections explain the various steps of the algorithm and the details of their implementation.

Prefix scan

The first step in expanding the frontier is finding the allocating an array to hold the next frontier and populating it with the neighbors of each vertex in the present frontier. In order to do this in parallel, we make use of prefix sum. An array of size equal to the size of the present frontier is allocated to hold the number of neighbors of each vertex. Each index is populated using R[v+1] – R[v] for each vertex v. We then use prefix sum for implementing cooperative allocation, i.e., to compute the offsets for where each thread should start writing its output elements.

Many parallel implementations of prefix sum follow a two pass procedure with a similar model as taught in the lectures. We use a similar approach where the threads in a block cooperate to compute the prefix sum of all the elements in one block. Two for loops are used, in which the

first loop progressively compute partial sums of interleaved elements by doubling the stride in every iteration and the second weaves the remaining elements of the scan. The amount of work is cut in half at each step, resulting in an overall work complexity of O(n). The figure below shows a high-level picture of the thread accesses.

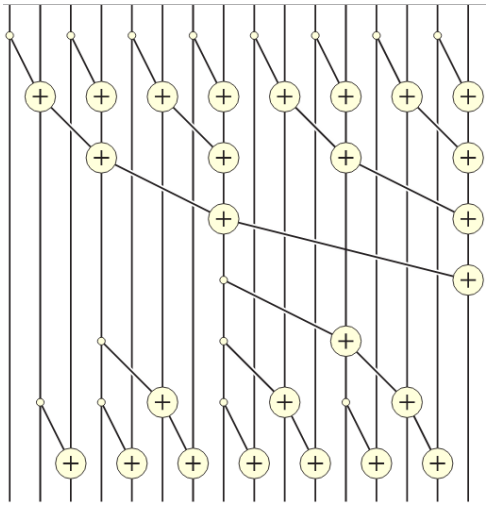


Figure 6 Working of Prefix Scan

The above algorithm does not perform very efficiently due to its memory access patterns. When multiple threads in the same warp may access the same shared memory bank, bank conflict would occur unless all threads of the warp access the same address within the same 32-bit word. We double the stride between memory accesses at each level of the tree, simultaneously doubling the number of threads that access the same bank. For deep trees, as we approach the middle levels of the tree, the degree of the bank conflicts increases, and then it decreases again near the root, where the number of active threads decreases. We avoid most bank conflicts by adding a variable amount of padding to each shared memory array index we compute. Specifically, we add to the index the value of the index divided by the number of shared memory banks. Refer the kernel **prefixscan** for the implementation.

The prefix-sum described above can handle only an input size of 1024, which is the maximum number of threads in a block, since only the threads of a block can cooperatively calculate the scan for the elements that they handle. But the input may contain more number of vertices per frontier, especially with large graphs. So, to handle this, we use another kernel **prefixscan2** to add the last element of each block to all the elements of the next block and complete the prefix-sum.

Gathering neighbors

The next step in expanding the neighbors of each vertex, is to write all the neighbors into an array. Our initial implementation uses serial gathering for this. Each thread obtains its preprocessed row-range bounds, where it needs to write, using the output of prefix sum. It

then serially acquires the corresponding neighbors from the column-indices array C. Thus each thread would expand the neighbors of one vertex in the present frontier. Refer the kernel **getNextFrontier** in 'bfs.cu'. The below picture illustrates the procedure of expansion:

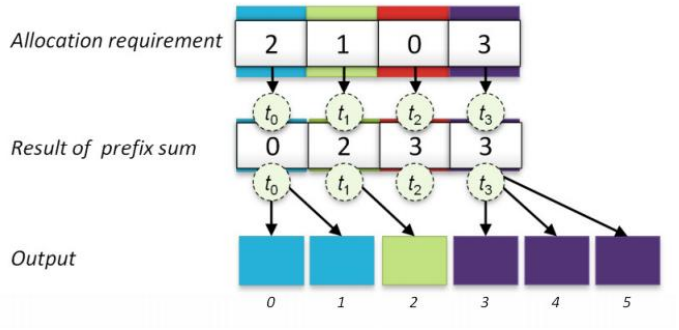


Figure 7 Serial Gathering

This method of gathering shows an imbalance in the amount of work done by each thread. The paper describes another method for fine-grained parallel adjacency list expansion. They mention CTA - wide co-operative expansion of the neighbors. Since we were not familiar with using CTAs, we tried to balance the work done by the threads belonging to a block, in a similar way. In this process, each thread tries to write one element of the output array. The block would thus expand only as many vertices whose output can be written by its threads. Each thread first obtains the vertex it needs to expand and the number of neighbors of the vertex that it needs to expand. It writes the column-indices of those neighbors into a shared memory array. Each thread then writes one element of the output using the column-indices and C. Thus the global memory accesses to C are balanced among all the threads. The following diagram shows this procedure of expansion for a similar input as in the previous example:

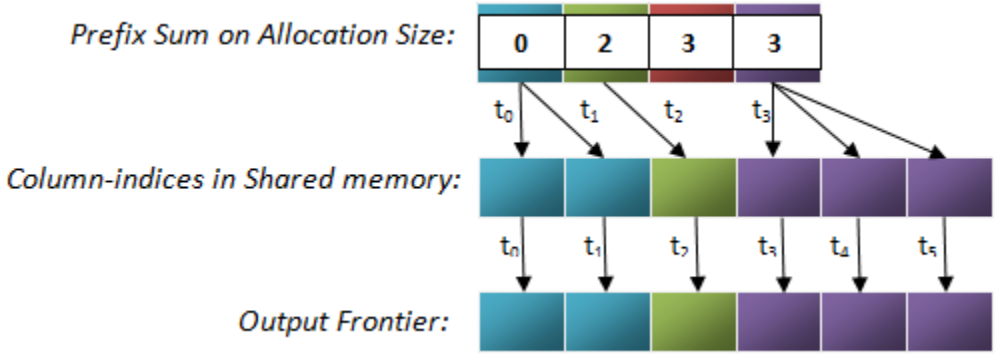


Figure 8 Fine Grained Gathering

In this method, each block needs to find a starting vertex to perform the expansion. We initially tried using binary search for this, where the first thread in each block performs a binary search to obtain the starting index and all the threads can use that value to proceed. But the search seemed to have an over-head and we saw lesser performance. We then tried to increase the work done by each thread. So, we wrote multiple output elements, defined by GATHER_TILE_SIZE, in each thread. The maximum value of GATHER_TILE_SIZE allowed by the shared memory limitations showed the maximum performance, but it was still a little slower than the serial gathering. Refer to the kernel **getNextFrontierBS** for the implementation. Next, we tried to pre-compute the starting indices for each block in a separate kernel and pass the values to avoid the binary search. But this implementation has some problems in certain cases where multiple elements of the input array need to be expanded by a single thread and we are still trying to fix it.

Filter neighbors (previously-visited & duplicates)

The duplicates and already visited vertices in the next frontier obtained after the gathering phase needs to be culled to avoid creating redundant work. The already visited vertices need to be removed because their depth should not be changed, but the duplicates do not necessarily have any implications on the correctness of the algorithm. So, our preliminary implementation did not include the removal of duplicates. But this showed a great impact on the performance, as the number of duplicates would increase hugely as the depth increases. There was a lot of redundant work generated, due to which the implementation was performing much slower than the naïve implementation. We then implemented this phase to remove the duplicates.

This phase of neighbor-gathering looks up the status of the vertices gathered; it entails checking depth array to determine which neighbors within the frontier have already been visited and the removal of duplicates. For this phase, the paper mentions different techniques called *Warp-culling*, *History-culling,* etc. But we explore a simple scan based duplicate removal technique for our implementation.

A 'mask' array where each index corresponds to one vertex in the graph, is used to store the status of each vertex. The status indicates whether the vertex in the next frontier is a newly discovered vertex or not. To obtain the mask, we initially reference the depth array and set the mask to 1 for all the vertices which are present in the next frontier and have an invalid depth of -1. This process implicitly removes all the duplicates as each vertex corresponds to only one index in the mask. To gather all these vertices, we again use a prefix-sum, in a similar way as described previously and generate an array called 'mask_scan'. Using the mask and the mask_scan, we allocate and populate a compact array for the next frontier. Refer to the kernels **generateMask**, **setDepth** and **compact** for the implementation. The figure below shows all the steps described above, for an example input:

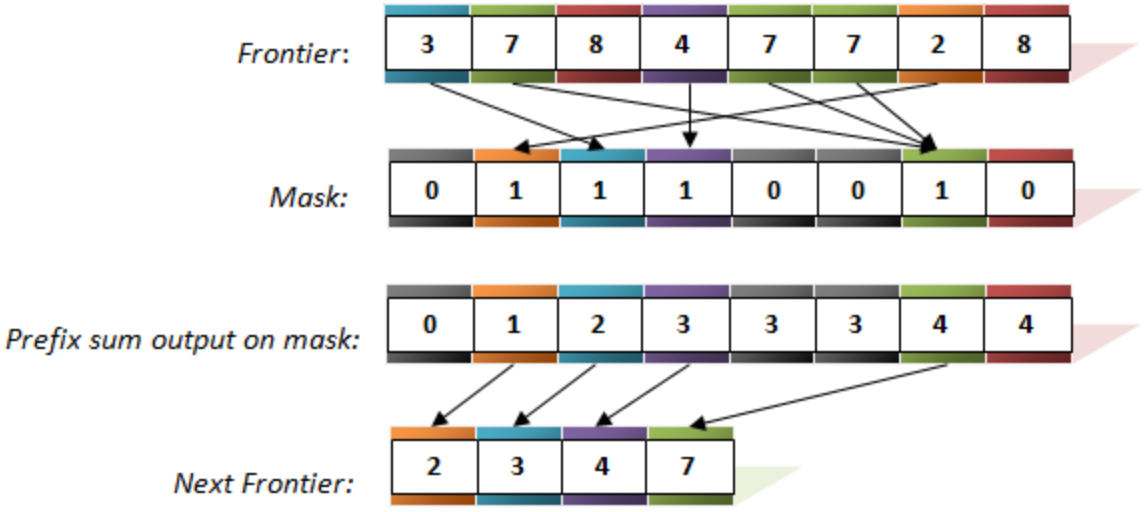


Figure 9 Duplicate Removal

The prefix scan might prove costly for small inputs or short frontiers. So, to optimize this further, we employ a threshold on the number of vertices in the frontier, below which we perform a sequential scan of the mask to populate the final frontier, instead of using the prefix sum. The threshold is set to a value of 512, based on some experiments.

Depth setting and completion

The depth for all the vertices is set using the mask and accessing depth array for each vertex. A variable *current_depth* is incremented in every iteration and the kernel **setDepth** sets the depth for all the vertices that have the mask set to 1, to the *current_depth*. The same function also checks if the depth for any vertex is invalid in order to check for the completion of traversal. A flag variable is set to indicate completion, which is tested on every iteration.

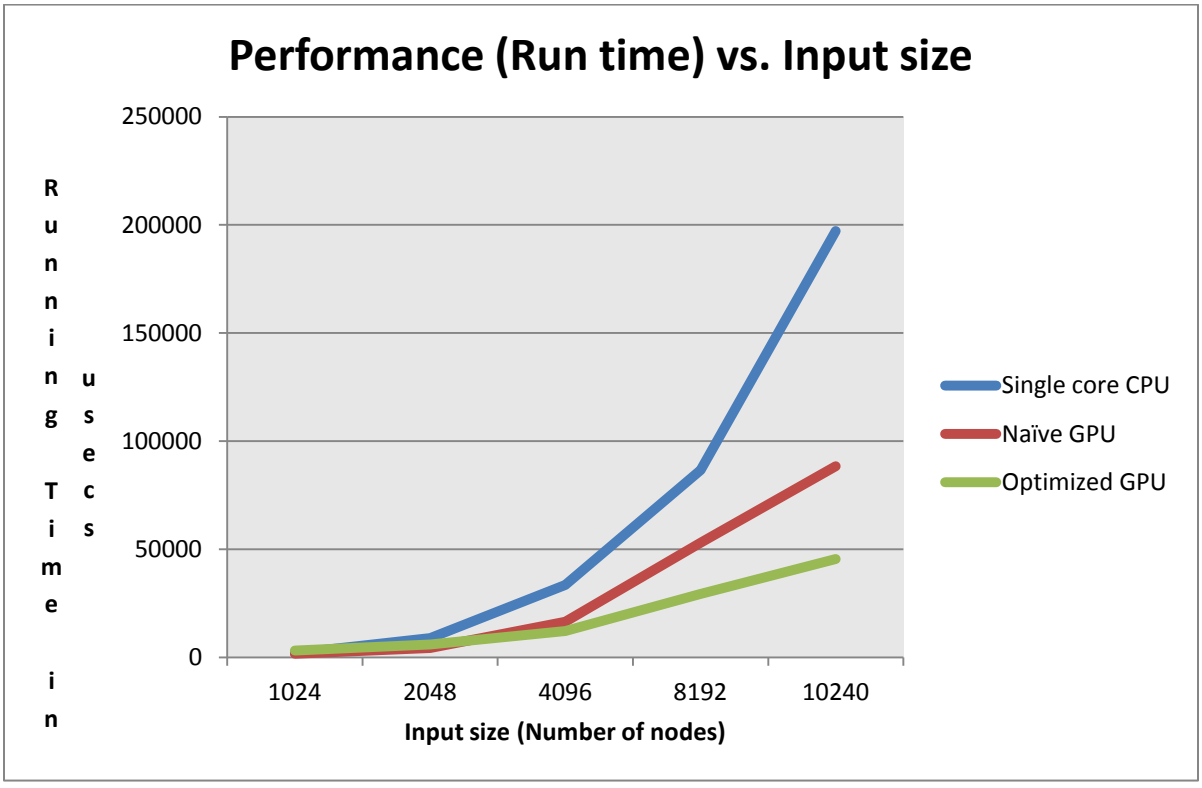
Experiments

The Jinx cluster located at Georgia Tech's College of Computing was used for the development and for testing performance. The CPU variant was executed on a single core of an Intel Xeon processor. For the GPU-based implementation, a single Tesla M2090 device was used. The following table shows that average run time in usecs for each of the benchmarks for the different implementations. Since the algorithms perform best for sparse graphs, the density is kept 25% and the maximum degree is set to 100%.

Input Graph size (Number of nodes)	Single Core CPU	Naive GPU	Optimized GPU variant
1024	1997	1653	3060
2048	8931	4327	5879
4096	33472	16485	12142
8192	86553	53052	29270
10240	197114	88398	45412
16384	299541	211666	95539
20480	467524	338202	143536
32768	1211611	898731	374758
40960	1892354	1368697	574792

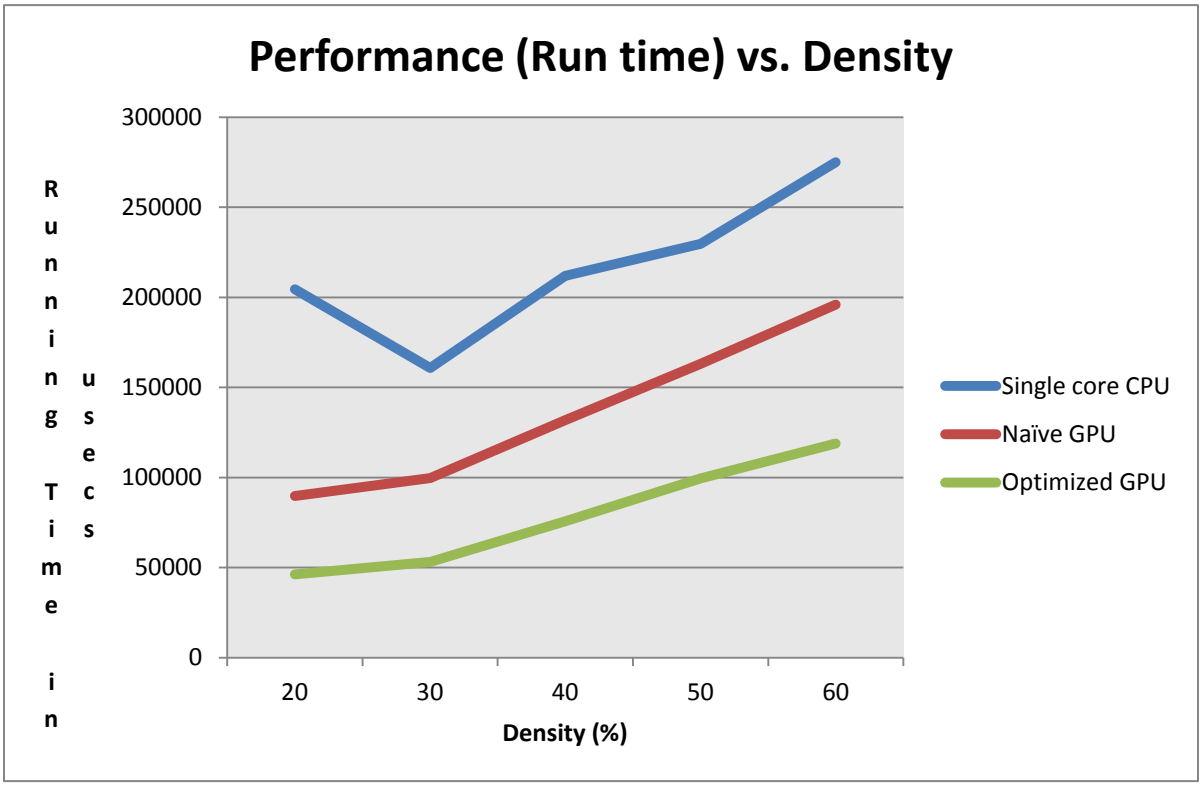
From the results, we see that the CUDA naïve implementation shows an average speed-up of 2x while the optimized implementation is much faster with an average speed-up of 4x. Also, the naïve GPU implementation degrades progressively as the size of the graph increases while the optimized version continues to give a good speedup.

Below is a graphical representation of this data:



Next, we analyze the impact of density variations for each of the implementations, to observe the scaling of the implementations as the density of the graph varies. The following table and graph show the performance variation for the three versions on a fixed-size input of 10240 nodes, but varying densities of the graph:

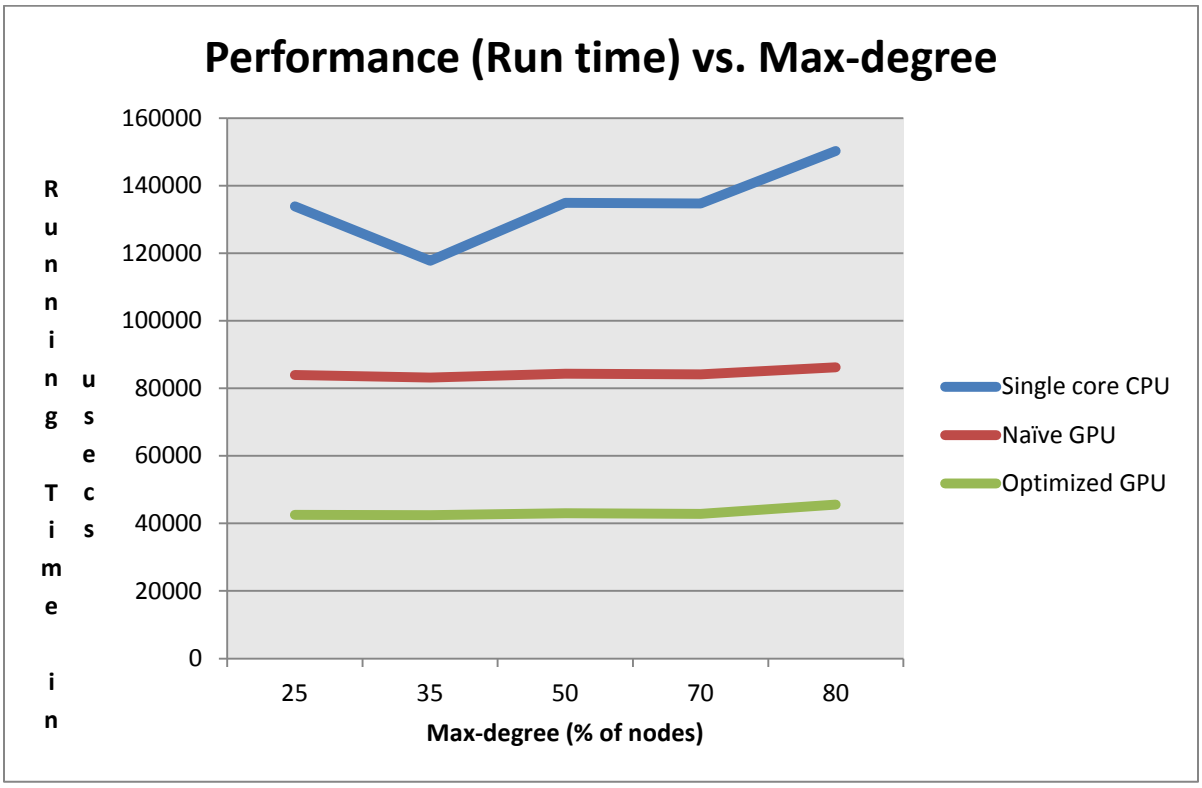
Denisty	Single Core CPU	Naive GPU	Optimized GPU variant
20	204521	89628	46254
30	160688	99605	53136
40	211950	131760	75666
50	229761	163117	99436
60	274914	195936	118878
70	345689	229205	139649
80	385518	262268	159321



This data shows that the performances of all three implementations are impacted by the changes in density. While the GPU implementations show a continuous degradation, the CPU variant shows more irregular run times.

We also study the performance of the different implementations, on graphs with varying max-degree, i.e., the maximum degree of any vertex in the graph. The below graph shows the variation on an input size of 10240 with a fixed density of 25%:

Degree	Single Core CPU	Naive GPU	Optimized GPU variant
25	133909	83904	42541
35	117804	83155	42427
50	134931	84341	42992
70	134758	84127	42850
80	150290	86250	45598
90	192377	87083	45158
100	213684	89488	46324



This data also shows an irregular variation in the performance of CPU implementation while the GPU variants show a smooth variation. The performance of the GPU variants is almost constant even for extreme values of the max-degree while the CPU version seems to show significant degradation overall.

Conclusion

The results from the experiments demonstrate that GPUs are well-suited for both sparse graph traversal as well as Floyd Warshall Algorithm. The two parallel versions of BFS algorithm implemented perform well for sparse graphs. While the naive implementation has a quadratic work complexity, the optimized implementation performs an asymptotically optimal amount of work and is expected to scale very well for large graphs.

Though initially the Floyd Warshall algorithm did not reveal much potential for parallelization, analysis has demonstrated that the All Pairs Shortest Path Problem can be solved on GPUs efficiently. Future work could involve porting these graph algorithms for computation across multiple GPUs.

References

[1] “Scalable GPU Graph Traversal” Duane Merrill (NVIDIA), Michael Garland (NVIDIA), Andrew Grimshaw (University of Virginia), in *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, Feb 2012.

[2] P. Harish and P.J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In Proc. 14th Int'l. Conf. on High-Performance Computing (HiPC), pages 197{208, dec 2007.

[3] L. Luo, M. Wong, and W m. Hwu. An effective GPU implementation of breadth-first search. In Proc. 47th Design Automation Conference (DAC '10), pages 52-55, June 2010.

[4] Bondhugula, U.; Devulapalli, A.; Fernando, J.; Wyckoff, P.; Sadayappan, P., "Parallel FPGA-based all-pairs shortest-paths in a directed graph," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, vol., no., pp.10 pp.,, 25-29 April 2006

[5] Matsumoto, K.; Nakasato, N.; Sedukhin, S.G., "Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System," *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on* , vol., no., pp.145,152, 2-4 Sept. 2011

[6] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. 2003. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics* 8, Article 2.2 (December 2003). DOI=10.1145/996546.996553 http://doi.acm.org/10.1145/996546.996553

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