Graph Traversal: All-Pairs Shortest Path

Optimizing the Floyd-Warshall Algorithm

EC527 Spring 2016

Final Project

Isaac Goldman

Petar Ojdrovic

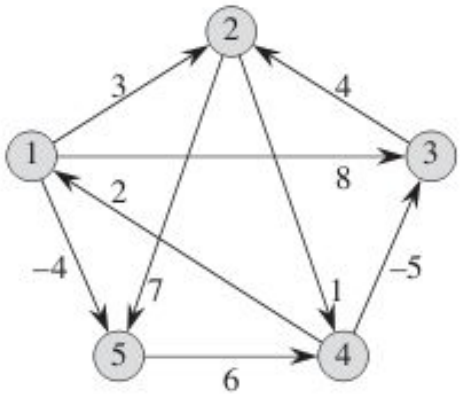
Introduction

From social networking to VLSI design, graphs and graph theory are integral to daily life, and the all-pairs shortest path problem is one of the most computationally intensive and algorithmically challenging common applications. A solution to the all-pairs shortest path problem finds the the shortest (least costly) path between each vertex in a graph and every other vertex. Finding an efficient solution is very important for a diverse range of real-world applications, such as computer clustering to high-frequency trading. With the rapid increase of mobile applications relying on real-time internet-based mapping systems (i.e. Google Maps), finding the shortest paths between all vertices in a graph as quickly as possibly becomes more important by the day. Given the high-performance necessity of the problem and its plethora of real-world applications, we thought it would be a productive topic to investigate further.

Although there are quite a few established algorithms for solving the all-pairs shortest path problem, we chose to implement and optimize Floyd-Warshall due to it's easy-to-follow reference code and its potential for a variety of optimizations. Additionally, as we began to implement the serial version, we noticed the structural similarity of the algorithm to matrix multiply. Although there are some significant differences from MMM which make it more complex to optimize and achieve markedly better performance (as discussed in the next section), we thought it would be interesting to be able to apply the techniques learned from optimizing MMM and seeing how they improved another real-world application as a benchmark for optimization.

Serial Implementation and Basic Optimizations

The standard implementation of Floyd-Warshall operates on a graph represented by an adjacency matrix, where any index A[i][j] represents the weight of an edge between vertex i and vertex j. If there is no edge connecting them, the distance is infinite. To find the shortest path for all pairs of vertices, Floyd-Warshall breaks the problem down dynamically, and iterates through all the vertices V times (V = number of vertices) for each vertex K, and checks to see if the shortest path between two vertices is actually shorter going through k than a direct connection. Every time a shorter path between i and j is found, the matrix is updated and the connecting node K is stored in a predecessor matrix at the index of the two vertices. The algorithm results in a finished distance matrix storing the values of the shortest distance between each vertex from all other vertices as well as the predecessor matrix storing the preceding vertices in the path, and runs in O(V^3) time and O(V^2) space (if not implemented recursively). The diagram below displays an example graph and the resulting matrices produced for each iteration of the algorithm:



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

The standardized pseudo-code for the algorithm:

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

(From Cormen et al, *Introduction to Algorithms)*

Using the pseudo-code above as a reference, we created a simple serial version of the code as seen here, not meant to optimize, but to serve as a baseline reference for our attempts at optimization:

```
for(k = 0; k < vertices; k++){
  for(i = 0; i < vertices; i++){
    for(j = 0; j < vertices; j++){
      // if d(i,k) + d(k,j) < d(i,j)
      if(distanceMatrix[i*vertices+k] + distanceMatrix[k*vertices+j] < distanceMatrix[i*vertices+j]){
        // then d(i,j) = d(i,k) + d(k,j)
        distanceMatrix[i*vertices+j] =  distanceMatrix[i*vertices+k] + distanceMatrix[k*vertices+j];
        // and set predecessor matrix (i,j) = k
        predMatrix[[i*vertices+j] = k;
      }
    }
  }
```

To generate graphs, we implemented two adjacency matrix generators, one for complete graphs (all vertices are connected to each other by an edge of random weight), and one that was parameterized to allow testing a range of densities (the percentage of connected vertices), both displayed below:

```
// generates a 2D distance matrix for a complete directed graph
void init_distance_matrix(matrix_ptr G, long int len, long int distance){
  printf("\nInitialize matrix: %d vertices\n", len);
  //srand(time(0));
  srand(18); // same seed for validation
  long int i, j;
  G->len = len;
  G->edges = len * len;
  for(i = 0; i < len; i++){
    for(j = 0; j < len; j++){
      if(i == j){
        G->data[ i * len + j] = (data_t)0;
        continue;
      }
      long int edgeWeight = rand() % distance;
      G->data[i*len+j] = edgeWeight == 0 ? 1 : edgeWeight;
    }
  }
  //print_graph(G);
  printf("\n\nNew Distance Matrix!\n");
}
```

```
// intialize an incomplete graph with parameterized density
void init_distance_matrix(matrix_ptr G, long int len, long int distance, long int edges, long int maxedges){
  srand(18);
  long int edgeRange = (maxedges/edges);
  long int i, j;
  G->len = len;
  G->edges = edges;
  for(i = 0; i < len; i++){
    for(j = 0; j < len; j++){
      if(i == j){
        G->data[ i * len + j] = (data_t)0;
        continue;
      }
      long int edgeWeight = rand() % edgeRange;
      G->data[i*len+j] = edgeWeight == 0 ? (rand()%distance)+1)  : INFINITE;//set edge random edge weight
  }
}
//print_graph(G);
}
```

We used complete graphs to generate most of our data, as that provided consistent computation intensity, however, we ran tests over varying densities. For the adjacency matrix structure, we saw almost no difference in performance, even for very sparse graphs. This is to be expected based on the structure of the algorithm, as it will have to iterate over every vertex three times whether or not the values of the distance matrix will

change. We were curious to see if pre-processing a sparse graph into an alternative structure would show an increase in performance, however as we researched common structures for sparse graphs, it became apparent that implementing such a case would change the structure of the algorithm so drastically that we would have to create another serial code and the scope might become too large for our time constraints. However, if this project were to be taken further, such implementations would be an important direction to investigate.

For validation, we first validated the the output of our serial implementation and made sure we were getting consistent results. We then tested each of our next implementations againsts the results of the serial, testing over the same matrices, and compared the values of the final outputs, incrementing an error counter for any inconsistency. For all of the final implementations of our optimizations, we were able to achieve 100 percent accuracy, thouh with some cost to performance, as discussed below.

Although similar in structure to a matrix multiply, Floyd-Warshall has data dependencies that makes optimization and parallelization not as straightforward. Due its dynamic nature, correct evaluation of the shortest path between vertices i and j is critically dependant on the distance matrix created in the k-1th iteration. Therefore, loop interchange can only be done with the two inner loops, and provided trivial differences in performance.

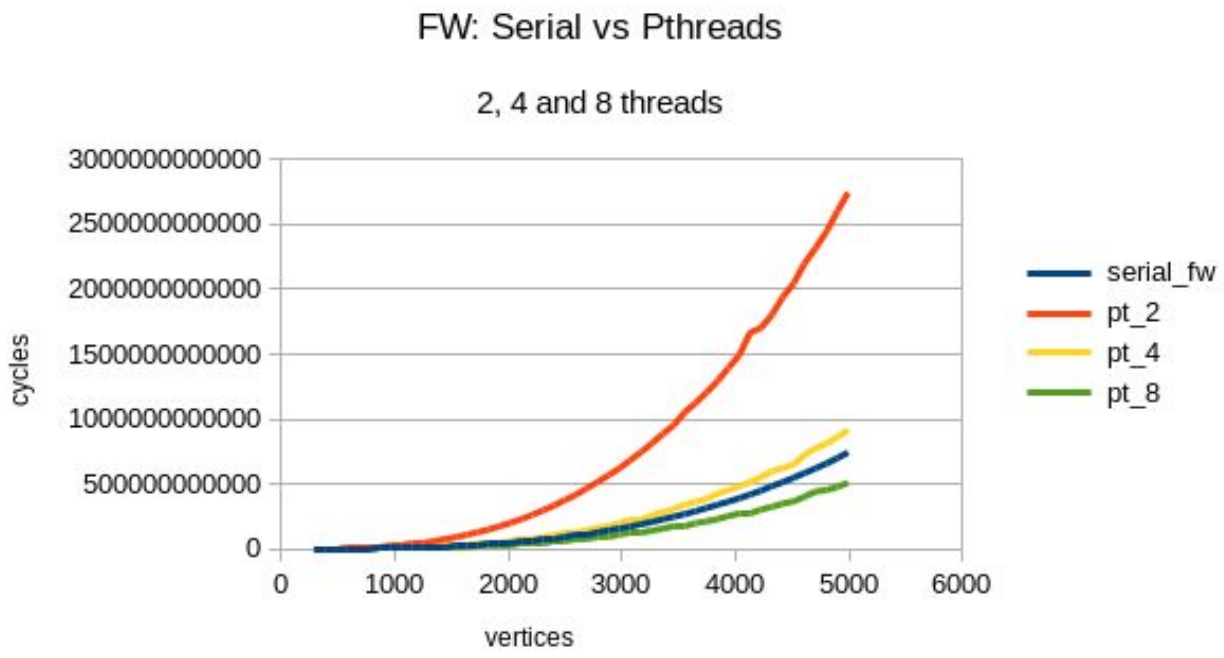
The dependency on the k loop also makes blocking difficult. Any blocking that successfully increased performance decreased accuracy to an unacceptable threshold.

Pthreads

Due to the dependencies discussed above, threading Floyd-Warshall in parallel proved to be difficult for getting any marked performance increase, however, we were finally able to achieve a successful 8x speedup (with 8 threads) over the serial implementation by threading the inner i and j loops and having each thread take a different strip of the matrix to iterate over and update(data parallelization), using a barrier after each incrementation of the outer loop to synchronize the threads:

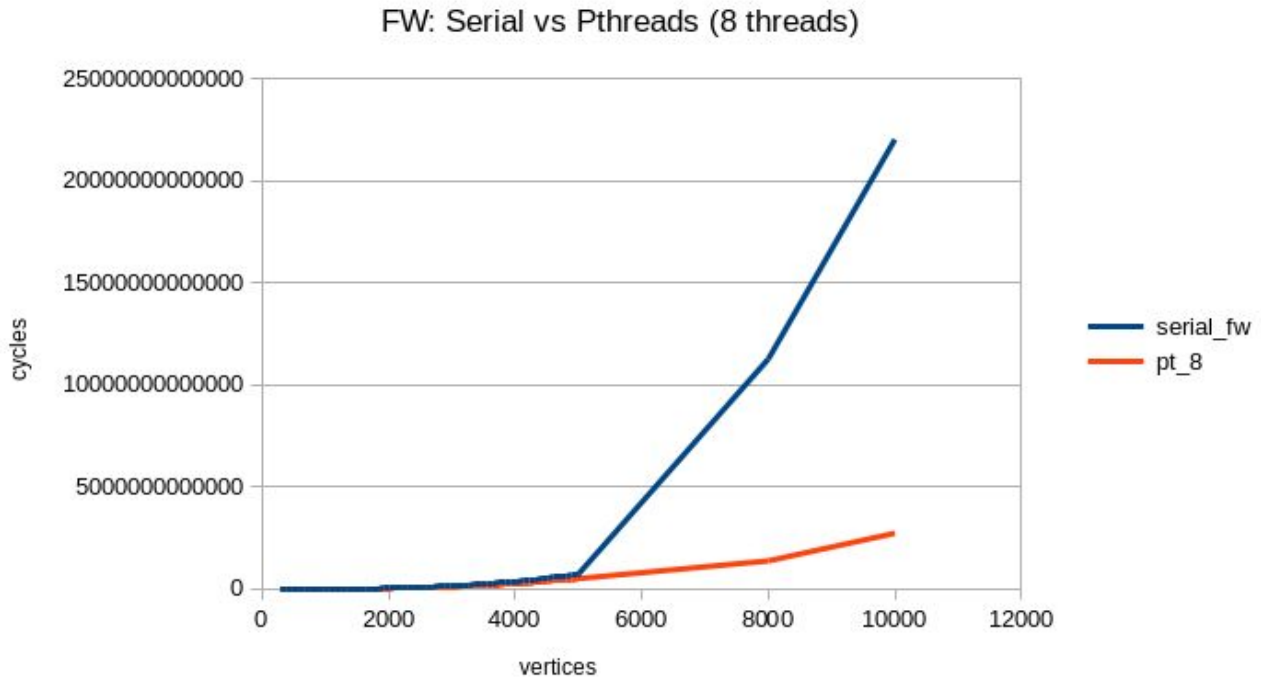
```
low = (taskid * vertices)/NUM_THREADS;
high = ((taskid+1)* vertices)/NUM_THREADS;

for(k = 0; k < vertices; k++){

  //  printf("\nwaiting for barrier\n");

  pthread_barrier_wait(&barrier);

  for(i = low; i < high; i++){
    for(j = 0; j < vertices; j++){
      currentIndex = i*vertices+j;
      currentPath = distanceMatrix[currentIndex];
      nextPath = distanceMatrix[i*vertices+k] + distanceMatrix[k*vertices+j];
      if(nextPath < currentPath){
        distanceMatrix[currentIndex] = nextPath;
        predMatrix[currentIndex] = k;
      }
    }
  }
```

Testing over an increasing number of threads (on the ME grid with 8 cores), we produced the following results:



For the lower number of threads, especially in the case of 2, it was apparent that the threads were competing with each other and resulting in false sharing. However, as the number of vertices increase, we saw a increase in performance for the 8 threaded version. Testing it further over larger array sizes we were able to produce a remarkable 8x speedup:

	8000 vertices	10000 vertices
speedup	8.16	8.06



Although the data dependencies presented initial difficulties for parallelization, it is clearly evident that multithreading does pay off for larger, real-world-sized graphs.

Cuda

As expected, the CUDA implementations of the Floyd Warshall algorithm offered a substantial improvement in speed - to the extent that implementing it in a non-cuda way seems to be obsolete, especially knowing that at any given moment, almost none of the processing power on a GPU is being used.

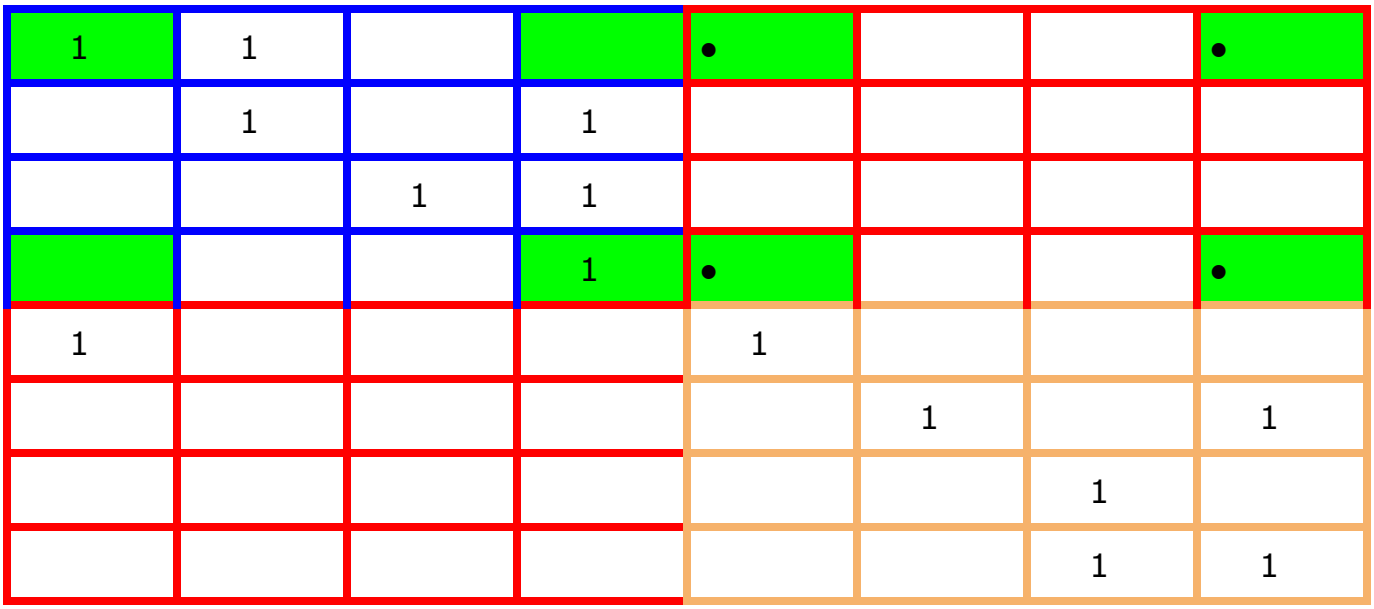
The reason it works so well on the GPU is that the structure of the algorithm essentially breaks the data up into "blocks", and operates on independent portions of the data. This is true for most programs written in a dynamic manner, and makes programs of this type highly parallelizable.

A simple PseudoCode implementation of Floyd Warshall looks as follows:

```
void Floyd_Warshall(Graph W){
 int n = NumOfRows(W);
 for(int k = 1; k < n; k++){
      Parallel_Floyd_Warshall[i = 1:n, j = 1:n](w);
 }
}

void Parallel_Floyd_Warshall(Graph W){
 W[i,j] = W[i,j] | (W[i,k] && W[k,j]);
}
```

Using this algorithm, we can block-process the data to achieve the desired speed.

The following is an example adjacency matrix which we can use to illustrate this point.



Using this part of the algorithm: `W[i,j] = W[i,j] | (W[i,k] && W[k,j]);` we will show what happens when K is equal to [1-4]

Pass 1:

i = [1-4], j = [1,4]

Pass 2:

i = [5-8], j = [1-4]

i = [1-4], j = [5-8]

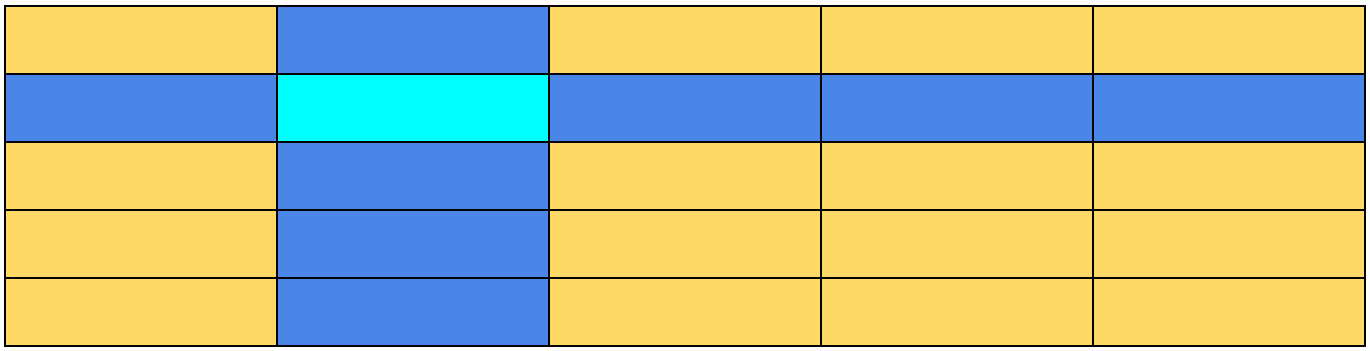
Pass 3:

i = [5-8], j = [5-8]

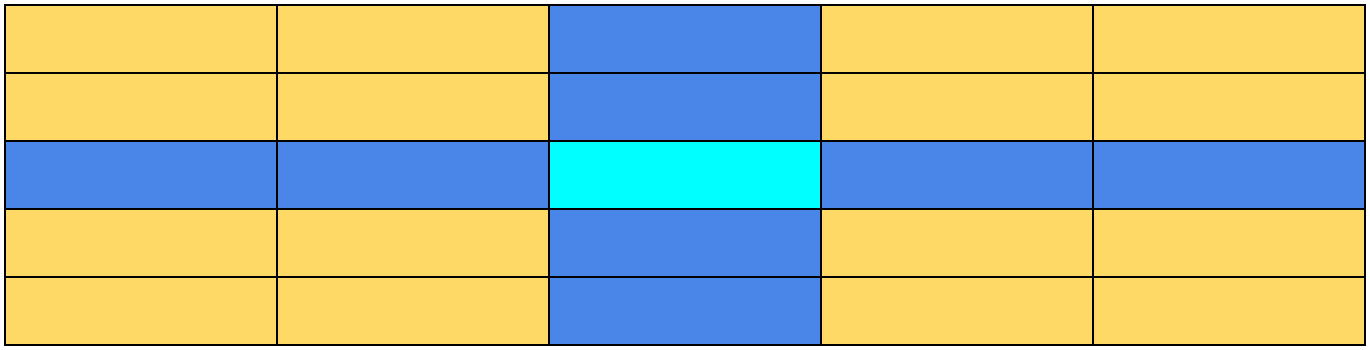
Note that for each iteration of K, the cells retrieved must be processed to at least k-1.

The following illustrates this process on a larger scale. Primary blocks are along the diagonal, Secondary blocks are the rows and columns of the primary block, and the Tertiary blocks are all remaining blocks.

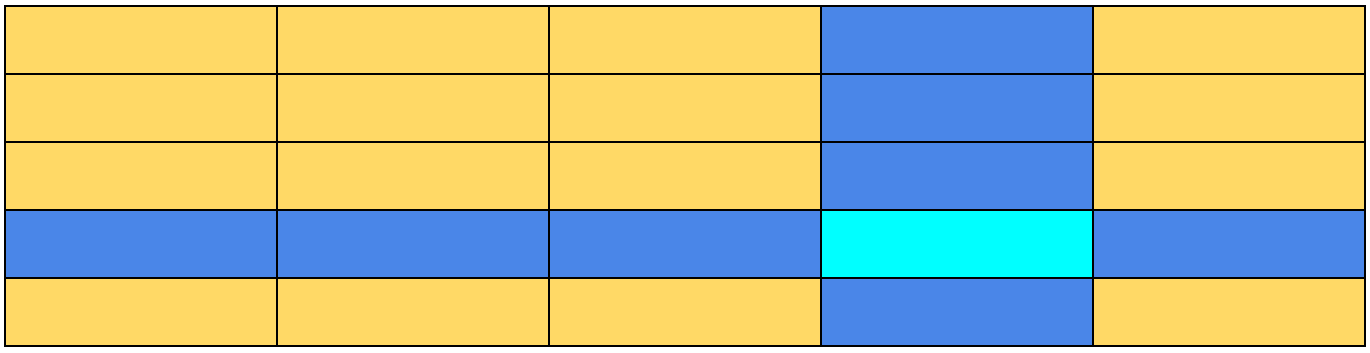
Pass 2



Pass 3



Pass 4

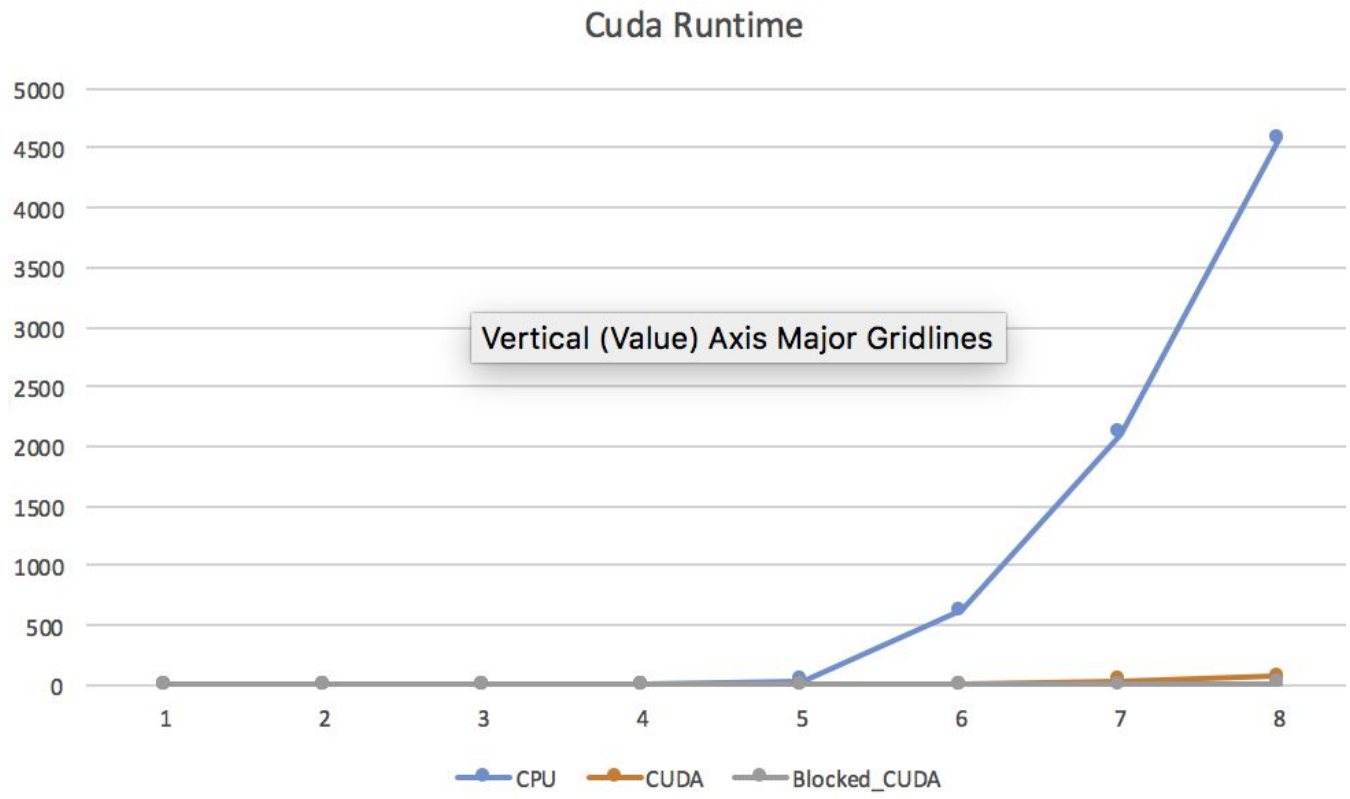


The results, as expected, were astounding.

The x-axis represents the sizes of the test matrices:

1	200
2	400
3	800
4	1000
5	2000
6	5000
7	7500
8	10000

The Y-axis is the runtime in seconds.



Once again, it is important to note that the increase in runtime is astounding. All-Pairs Shortest path is not an algorithm often used in layman computing, and those writing algorithms to be run on massively parallel machines already know what they're doing and how to write code to take maximum advantage of hardware. That being said, a future where the computer takes more advantage of the idle time on the GPU could be a future in which much more performance could be squeezed out of unchanging hardware.

Cuda, too, has its limits - these being related to the number of CUDA cores on the GPU. The processor we used had 1596 cores, so the theoretical maximum speedup would have yielded a runtime one-and-a-half thousand times better than the serial runtime. On SLI-ed systems, or supercomputers with tens or even hundreds of GPUs, the increase in performance becomes even greater. Unfortunately, we weren't able to write code and test it on a system with multiple independent GPU's.

The only realistic downside to writing APSP for the GPU is that writing even marginal code requires a strong knowledge of GPU architecture and how to use its features effectively.

Conclusion

Although Floyd-Warshall does not lend itself easily to efficient parallelization due to the structural and data dependencies of the algorithm, we were able to successfully achieve significant performance increases using both parallel threading and a GPU. Moving forward, our next steps would be to test our optimizations with some real world data. We found some interesting financial data from high-speed trading in addition to paths of computer clustering implementations, however, due to time constraints, we were unable to convert the information into our chosen graph structure to produce substantial and verifiable results. This would be our next step if we were to take the project further.

A further implementation we would like to explore is data and information movement around a wireless sensor network. As factories and hospitals are becoming more and more automated, the number of wireless sensors is growing exponentially. However, existing networks in these locations will soon run out of the bandwidth necessary to support all the sensors, and re-building the network is an extremely expensive undertaking. By moving the data around in a more efficient manner, and getting sensor data to where it needs to be in a more efficient manner, one could increase the number of devices without fundamentally altering or upgrading the network architecture, saving a large amount of money.

Works Cited

Cormen, Thomas et al, *Introduction to Algorithms*

https://hal.inria.fr/hal-00905738/document

http://research.ijcaonline.org/volume110/number16/pxc3900920.pdf

http://arxiv.org/pdf/1001.4108.pdf

http://www.shodor.org/media/content/petascale/materials/UPModules/dynamicProgrammingPartI/dynProgPt1ModuleDoc.pdf

https://github.com/d1vyank/LeastCostPath

https://github.com/OlegKonings/CUDA_Floyd_Warshall_

http://www.cs.virginia.edu/~pact2006/program/pact2006/pact139_han4.pdf