Parallelization of Floyd-Warshall Algorithm

Picchirallo Francesco Alessandro VR432225 Avogaro Andrea VR422017

***Abstract*—The Floyd-Warshall Algorithm solves the all-pairs shortest path problem on direct graphs. In this work this known algorithm will be parallelized using two different parallel programming: CUDA and OpenMP.**

I. Introduction

The shortest path problem refers to the problem of finding the shortest path or minimal cost route from a specific source to a particular destination. Normally, graphs are most widely used for representation of such problems. Shortest path problem basically deals with graphs and with problems belonging to weighted directed graph category. It finds applications in a large number of domains such as, in robotics and intelligent systems, routing protocols, bioinformatics etc. A graph is a finite set of vertices and edges represented as G(V, E). Vertices are connected using edges. A directed graph is a graph in which each edge could be traversed only in a given direction. To calculate the distance between vertices (nodes), edges are given some values called cost (weights). These costs measure the distance between any two nodes via different possible edge sequence. Shortest path is not restricted to the shortest distance between the source and destination, but also refers to other measurement units such as time, cost and capacity of the link.

Graphics Processing Units are parallel processors offering high FLOPS/sec for low cost. CUDA is a parallel computing platform and programming model that makes using a GPU for general purpose computing created by NVIDIA, small extension of C/C++. With CUDA it is possible to accelerate your code by moving computationally intensive portions of your code to an NVIDIA GPU.

OpenMP is an Application Programming Interface (API) that supports multi-platform shared memory parallel programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms.

II. Background

Floyd-Warshall's Algorithm is a different approach to solving the all pairs shortest paths problem. Rather than running Dijkstra's Algorithm on every vertex, Floyd-Warshall's Algorithm uses dynamic programming to construct the solution. The general idea is to - across n iterations, where n is the width and height of the adjacency matrix graph input - pick all of the vertices as intermediates in the shortest paths. We then test if the picked vertex is in the shortest path. If it is an intermediate in the path, the distance will be updated, else the original distance is kept. Floyd-Warshall's Algorithm takes in an adjacency matrix representation of a graph, and returns a matrix with the shortest distances between any two vertices. The work for this algorithm is O(V^3), where V is the number of vertices in the graph.

Algorithm 1 Floyd-Warshall Algorithm

```
Result: Matrix with shortest path
for k = 1 → n do
    for i = 1 → n do
        for j = 1 → n do
            if M[i][k]! = ∞ ∧ M[k][j]! = ∞ then
                if M[i][k] + M[k][j] < M[i][j] then
                    M[i][j] ← M[i][k] + M[k][j]
                end
            end
        end
    end
end
```

III. Parallelized Versions

The project goal was to parallelize the Floyd-Warshall algorithm using CUDA and OpenMP. Starting from the sequential version, we have implemented three differents version of parallel Floyd-Warshall version's in CUDA:

- Simple parallel kernel configuration
- Blocked Version using GPU Global Memory
- Blocked Memory using GPU Shared Memory

The first implementation is a simple parallelization of the basic Floyd-Warshall algorithm, working on the same mode of the sequential version, introducing thread, blocks and CUDA kernel. The Blocked version instead works on a different way, splitting into three phases the normal Floyd-Warshall.

A. Simple parallel kernel configuration

The first CUDA approach took a simple way to get a basic benchmark, not much faster than the sequential one. The host function launch as many BLOCKDIM x BLOCKDIM blocks as necessary to cover the graph and then each block's threads work on single values before returning. This requires n kernel launches, where every thread in each block has the task to update a particular element of the matrix. In this implementation, threads of the same warp are assigned to adjacent memory locations to assure memory coalescing.
There are many disadvantages with this approach, all memory

accesses are to global memory, which is extremely slow. This implementation makes no use of the shared memory of the blocks and, since Floyd-Warshall is a memory bound algorithm, this approach doesn't yet reach CUDA's full potential.

Algorithm 2 Simple Parallel Floyd-Warshall Algorithm

Input: Matrix $M[]$, number of vertices n, k vertice;
Output: Matrix with the shortest path

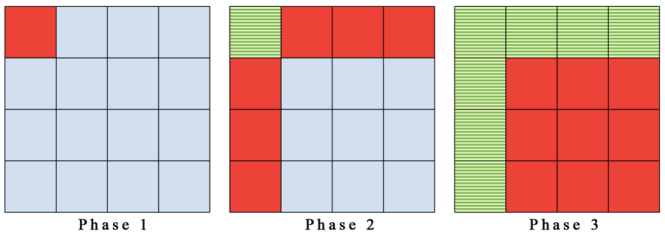
$i \leftarrow Global\,Thread\,Index\,in\,y$;
$j \leftarrow Global\,Thread\,Index\,in\,x$;
$kj \leftarrow k * n + j$;
$ij \leftarrow i * n + j$;
$ik \leftarrow i * n + k$;
if $i < n \wedge j < n$ **then**
  if $M[ik]! = \infty \wedge M[kj]! = \infty \wedge$
  $M[ik] + M[kj] < M[ij]$ **then**
    $M[ij] \leftarrow M[ik] + M[kj]$
  end
end

B. Blocked Version using GPU Global Memory

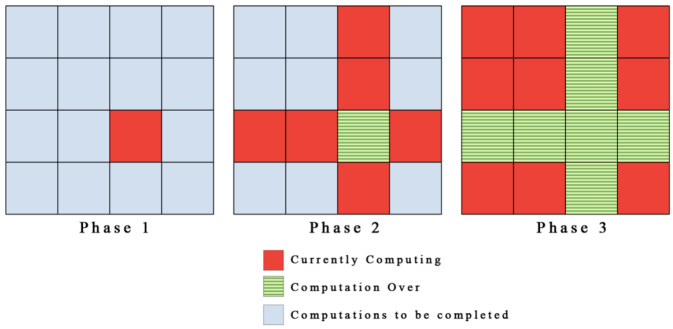
The original FW algorithm can not be broken into distinct sub-matrices that are processed into individual multiprocessors because each sub-matrix needs to have terms across the entire data-set. The blocked version of Floyd-Warshall consists in partitioning the original matrix into sub-matrices of equal size, will be used a subroutine that processes each of its given block. However, the blocks are not all independent and they must be processed in three separate phases. First, in the dependent phase, the kth diagonal block need to be processed. Then, in the partially dependent phase, the kth row and the kth column of blocks have to be computed. Lastly, the indipendent phase will produce the remainings blocks.
Since this blocked version was done using only Global Memory of the GPU, we expect not the best performance, since the access cost of Global Memory is high. For each phase, will not be allocated the same number of blocks. In the first phase only one block will be assigned because need to compute only the independent block, in the second one will be allocated 2 * numBlock to compute the dependent blocks which share the same row and column with primary block. BLOCK_SIZE constant defines the size of the block.

Each phase is composed by a kernel, which will be execute to find the shortest path solution. Each phase call the block_calc function, which task is to update one element in the global memory of the matrix. Here for each k in $BLOCK\_SIZE$ if exists a shortest path from vertex i to vertex j passing trough vertex k then the current value is updated by summing up the values from i to k and k to j, otherwise, the element remains unchanged.

- Phases when the block is in (1,1) position



- Phases when the block is in (k,k) position



Algorithm 3 Block Calc

Input: Matrix $M[]$ 1D, number of vertices n, block index k, memory index idx, idy;
Output: Matrix updated

$index \leftarrow idy * n + idx$
$old \leftarrow M[\text{index}]$

for $j = 0 \rightarrow BLOCK\_SIZE$ **do**
  $ku \leftarrow BLOCK\_SIZE * k + j$
  index1 $\leftarrow idy * n + ku$
  $index2 \leftarrow ku * n + idx$
  $sum \leftarrow M[index1] + M[index2]$
  if $sum < old$ **then**
    $M[index] \leftarrow sum$
  end
  $synchthreads()$
end

Algorithm 4 Phase 1

Input: Matrix $M[]$ 1D, number of vertices n, block index k
Output: Matrix updated

$idx = BLOCK\_SIZE * k + threadIdx.x$
$idy = BLOCK\_SIZE * k + threadIdx.y$
$block\_calc(M, k, n, idy, idx)$

Algorithm 5 Phase 2

Input: Matrix $M[]$ 1D, number of vertices n, block index k
Output: Matrix updated

if $blockIdx.x = k$ **then**
  $return$
end
$idx = BLOCK\_SIZE * k + threadIdx.x$
$idy = BLOCK\_SIZE * k + threadIdx.y$
if $blockIdx.y = 0$ **then**
  $idx \leftarrow BLOCK\_SIZE * blockIdx.x + threadIdx.x$
else
  $idy \leftarrow BLOCK\_SIZE * blockIdx.x + threadIdx.y$
end
$block\_calc(M, k, n, idy, idx)$

Algorithm 6 Phase 3

Input: Matrix $M[]$ 1D, number of vertices n, block index k
Output: Matrix with the shortest path

if $blockIdx.x = k \vee blockIdx.y = k$ **then**
  $return$
end
$idx \leftarrow BLOCK\_SIZE * blockIdx.x + threadIdx.x$
$idy \leftarrow BLOCK\_SIZE * blockIdx.x + threadIdx.y$
$block\_calc(M, k, n, idy, idx)$

C. Blocked Version using GPU Shared Memory

This implementation is similar to the Global Memory version. The big difference is moving the sub-matrices needed by every block from the Global Memory to the Shared Memory, where the *block_calc* function will do the computations. Since Shared Memory is much faster than Global Memory we expect a performance improvements. In all three kernel a matrix whose size is $BLOCK\_SIZE$ * $BLOCK\_SIZE$ is going to be allocated in Shared Memory, which containing the data of the all blocks to work on and the related blocks to compute each phase. Like the Global Memory version, all threads of the blocks invokes the *block_calc* function, modified from the Global Memory version, to compute the results working on the Shared Memory. Once the compute is over, the data will be copied to the Global Memory.

Algorithm 7 Block Calc Shared

Input: Matrix M,$old1$,$old2$ 2D,
memory index idx,idy
Output: Matrix updated

for $k = 0 \rightarrow BLOCK\_SIZE$ **do**
  $sum \leftarrow old1[idy][k] + old2[k][idx]$
  if $sum < M[idy][idx]$ **then**
    $M[idy][idx] \leftarrow sum$
  end
  $synchthreads()$
end

D. OPENMP

OpenMP, as already mentioned, is an API that supports multi-platform parallel programming. We have studied some OMP directives to parallelize the sequential code and to understand which one is the best for the Floyd-Warshall. The directive used is $ompparallelfor$. $Ompparallel$ pragma starts a parallel region, it creates a team of N threads (where N is determined at run-time, usually from the number of CPU cores), each of which execute the next statement. After the statement, the threads join back into one. In our implementation has been used $ompparallelfor$. The for directive split the for loop, so that each thread in the current team handles a different portion of the loop. The private clause indicates that each variable in its list is private, i.e. each thread has its own copy of it. It has been used two $ompparallelfor$ because in the sequential code are used two main loop, one for the i (row) and one for the j (column).

IV. Results

We used the CUDA 04 server to test our CUDA and OPENMP implementations. This server mount an AMD Phenom (tm) II x6 1055T Six-Core Processor with a GeForce GTX 980 4GB.

A. CUDA Results

For each CUDA implementations we took in exam three different $BLOCK\_SIZE$: 8 x 8, 16 x 16, 32 x 32.
This configuration ensure that the number of the threads is a multiple of the warp size, implying 100% theoretical occupancy of the SMs. The fastest configuration is the Blocked version using Shared Memory, followed by the Blocked version using Global Memory and the simple configuration as the last. We expect these results since the Shared Memory is faster than Global memory (shared memory ~1.7TB/s, while global memory ~150GB/s) and since the Blocked Floyd-Warshall is more accurate than the normal one. For each configuration the best $BLOCK\_SIZE$ is the 16 x 16, it guarantees the best balance between blocks available in order to hide memory latencies and having small blocks.
Using NVIDIA Visual Profiler we found that all of our implementation achieve 100% theoretical occupancy. The tables below show sequential, parallel timing and speed up between them.

1) Simple Parallel: The simple parallel implementations has 80% of real occupancy. This is due this implementation uses the base algorithm that doesn't broke the matrix into sub-matrix. Since this implementation doesn't use shared memory, the most of the threads wait for resources to be available. Table 1, 2 and 3 show the timing regarding this implementation, Figure 1 shows the occupancy.

2) Blocked Version using GPU Global Memory: This Blocked implementation has 98,4% of real occupancy, in fact we can see a clear performance improvements. As we can see in the Figure 1 from the Visual Profiler, this improvement is caused by a more usage of Unified and L2 cache and a less usage of the device memory.

3) Blocked Version using GPU Shared Memory: This Blocked implementation has 97,6% of real occupancy, less than the Global Memory version but since it uses shared memory, performance still increases. In fact, as we can see from the Figure 1, there is a massive use of shared memory followed by a less usage of Unified and L2 cache than the Global Memory Version.

B. OpenMP Results

OpenMP results are stable respect the size of the matrix but not the best, infact the speed up ranging up from 4.23 to 4.84. The worst result obtained is with the smallest dataset, the best is with the bigger one. These results were predictable since OpenMP works on CPU and not with GPU.

V. Conclusions

In this work we showed some parallel implementations for the Floyd-Warshall algorithm. As we've seen the Simple implementation brings good result in terms of speeding up, but the Blocked Version brings even better ones, with a triple speed up in some graph, however it is necessary develop custom algorithms. Instead with OpenMP it is very easy develop a CPU parallel version of the algorithm but the results are not comparable with the CUDA ones.

Figure 1: From the top to bottom: Memory utilization of Simple, Global and Shared implementations

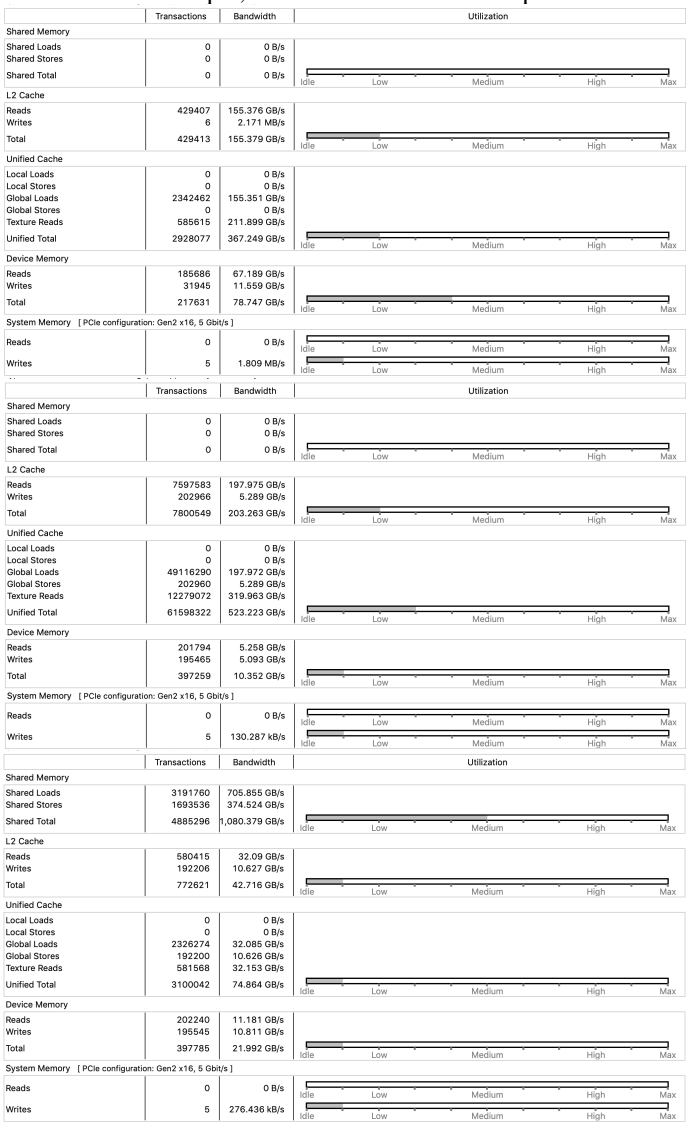


Table 1: Simple parallel implementation BlockSize 32x32

Matrix	Sequential	Parallel	Max Speedup	AVG Speedup
nos4	0.007928	0.000336192	23.5818	22.8988
bcsstk08	2.7394	0.0533424	51.355	51.005
spaceStation10	3.38391	0.069818	48.4676	47.9051
c-44	1840.76	30.0778	61.2	60.72
net50	6291.64	95.6252	66.8406	66.2511

Table 2: Simple parallel implementation BlockSize 16x16

Matrix	Sequential	Parallel	Max Speedup	AVG Speedup
nos4	0.007529	0.000328544	22.9163	22.4874
bcsstk08	2.73	0.0456372	59.8196	59.0951
spaceStation10	3.35725	0.061633	54.4716	53.8068
c-44	1840.76	26.2355	70.1629	69.7163
net50	6291.64	79.9736	78.6715	77.9895

Table 3: Simple parallel implementation BlockSize 8x8

Matrix	Sequential	Parallel	Max Speedup	AVG Speedup
nos4	0.007529	0.00033872	22.2278	21.8437
bcsstk08	2.76352	0.0501491	55.106	55.0274
spaceStation10	3.35169	0.0735277	45.584	45.0656
c-44	1840.76	43.6826	42.1394	42.0677
net50	6291.64	137.099	45.8912	45.6539

Table 4: Blocked Global & Shared version BlockSize 32x32

Matrix	Sequential	Blocked Global	Max Speedup	AVG Speedup	Blocked Shared	Max Speedup	AVG Speedup
nos4	0.007928	0.000255936	30.9765	30.7309	0.000117088	67.7098	62.0087
bcsstk08	2.83916	0.0307485	94.0120	94.04035	0.0135416	209.662	207.742
spaceStation10	3,36333	0,0412125	81,6094	80,0588	0.0215598	155.798	155.745
c-44	1840.76	21.3844	86.0796	86.0218	12.4649	147.676	147.414
net50	6291.64	70.3405	89.4455	89.07	43.7888	143.681	143.608

Table 5: Blocked Global & Shared version BlockSize 16x16

Matrix	Sequential	Blocked Global	Max Speedup	AVG Speedup	Blocked Shared	Max Speedup	AVG Speedup
nos4	0.007936	0.000211584	37.5076	34.4216	0.000106912	74.2293	73.1424
bcsstk08	2.83916	0.0259712	109.319	109.004	0.013104	216.663	216.204
spaceStation10	3.34951	0.0365363	91.6764	91.6536	0.0208938	160.312	160.274
c-44	1840.76	19.8244	92.853	92.8032	11.9308	154.286	154.237
net50	6291.64	67.6831	92.9573	92.5417	41.9981	149.808	149.776

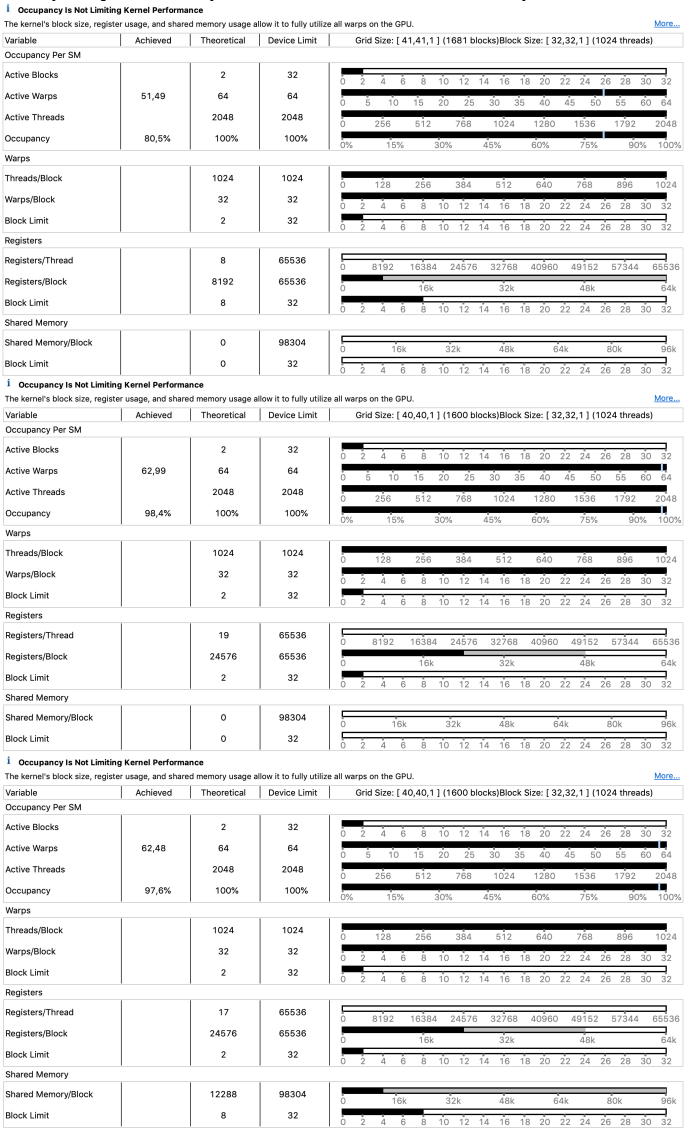
Table 6: Blocked Global & Shared version BlockSize 8x8

Matrix	Sequential	Blocked Global	Max Speedup	AVG Speedup	Blocked Shared	Max Speedup	AVG Speedup
nos4	0.007529	0.000274496	27.4285	27.3399	0.000161152	46.7199	46.1128
bcsstk08	2.76352	0.0359999	76.7646	75.6066	0.0157403	175.569	175.111
spaceStation10	3.35169	0.0509373	65.8003	65.7845	0.024135	138.872	138.844
c-44	1840.76	45.1378	40.7809	40.7794	13.6293	135.059	132.804
net50	6291.64	134.395	46.8144	46.7248	48.4357	129.897	128.486

Table 7: OpenMP implementation

Matrix	Sequential	OpenMP	Speedup
nos4	0.00421	0.000995	4.23115
bcsstk08	2.43498	0.53629	4.54042
spaceStation10	3.40011	0.712224	4.77394
c-44	1840.76	416.448	4.42014
net50	6291.64	1299.65	4.84102

Figure 2: From the top to bottom: Memory occupancy of Simple, Global and Shared implementations



Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [41,41,1] (1681 blocks)Block Size: [32,32,1] (1024 threads)
Occupancy Per SM				
Active Blocks		2	32	
Active Warps	51,49	64	64	
Active Threads		2048	2048	
Occupancy	80,5%	100%	100%	
Warps				
Threads/Block		1024	1024	
Warps/Block		32	32	
Block Limit		2	32	
Registers				
Registers/Thread		8	65536	
Registers/Block		8192	65536	
Block Limit		8	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	

Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [40,40,1] (1600 blocks)Block Size: [32,32,1] (1024 threads)
Occupancy Per SM				
Active Blocks		2	32	
Active Warps	62,99	64	64	
Active Threads		2048	2048	
Occupancy	98,4%	100%	100%	
Warps				
Threads/Block		1024	1024	
Warps/Block		32	32	
Block Limit		2	32	
Registers				
Registers/Thread		19	65536	
Registers/Block		24576	65536	
Block Limit		2	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	

Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [40,40,1] (1600 blocks)Block Size: [32,32,1] (1024 threads)
Occupancy Per SM				
Active Blocks		2	32	
Active Warps	62,48	64	64	
Active Threads		2048	2048	
Occupancy	97,6%	100%	100%	
Warps				
Threads/Block		1024	1024	
Warps/Block		32	32	
Block Limit		2	32	
Registers				
Registers/Thread		17	65536	
Registers/Block		24576	65536	
Block Limit		2	32	
Shared Memory				
Shared Memory/Block		12288	98304	
Block Limit		8	32	

Figure 3: From the top to bottom: Kernel Stall Analysis of Simple, Global and Shared implementations