Accelerating large graph algorithms (From CUDA to OpenMP)

Matteo M. Fusi

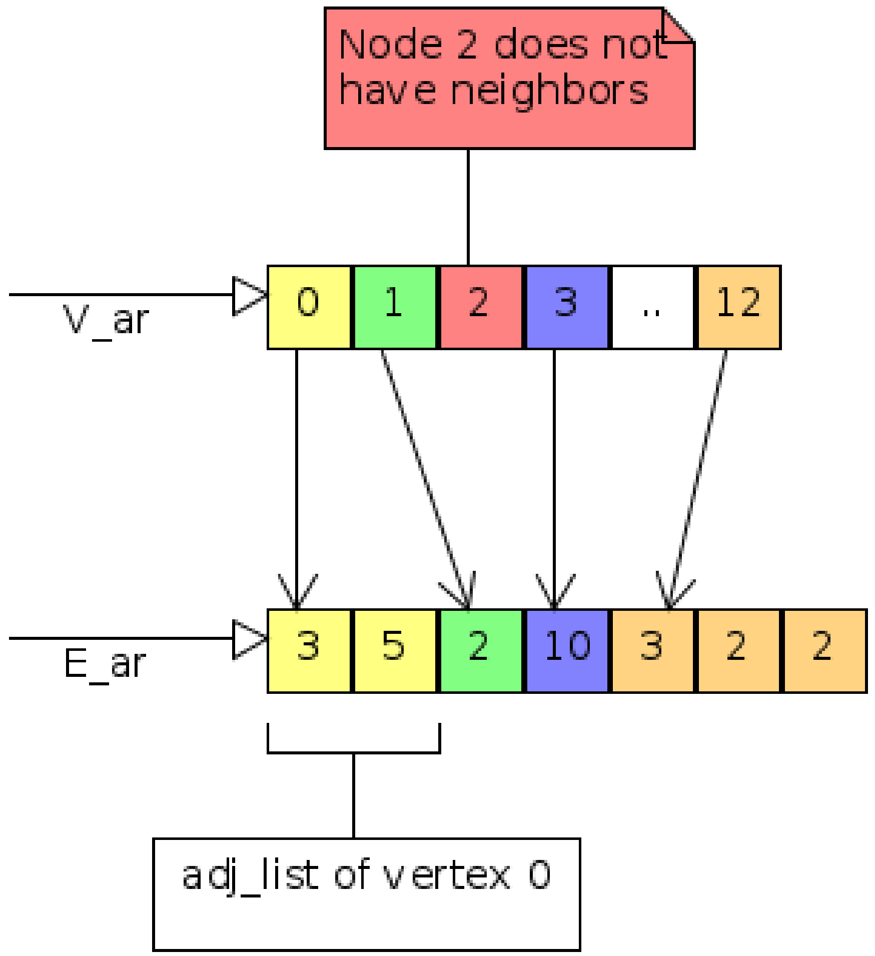
Abstract

Study the paper “*Accelerating large graph algorithms on the GPU using CUDA*” by Pawan Harish and P. J. Narayanan analyzing algorithms and presented results to obtain an efficient and equivalent solution using OpenMP.

- The authors of the article wanted to demonstrate the potential of GPUs (also cheap GPUs) for parallel algorithm implementation.
- We adapt their work to an OpenMP implementation.

The graph and its implementation

- G(V,E) where V are the vertices and E are the edges. If the weight of the edges must be taken into account the structure W of size E is considered.
- Vertices are packed in an array V_ar of size |V|. Every Vertex has a pointer to its adjacency list.
- Adjacency lists are packed in a large size array E_ar if size |E|.
- Each vertex points to the starting position of its own adjacency list in E_ar


Node 2 does not have neighbors
V_ar
0
1
2
3
..
12
E_ar
3
5
2
10
3
2
2
adj_list of vertex 0

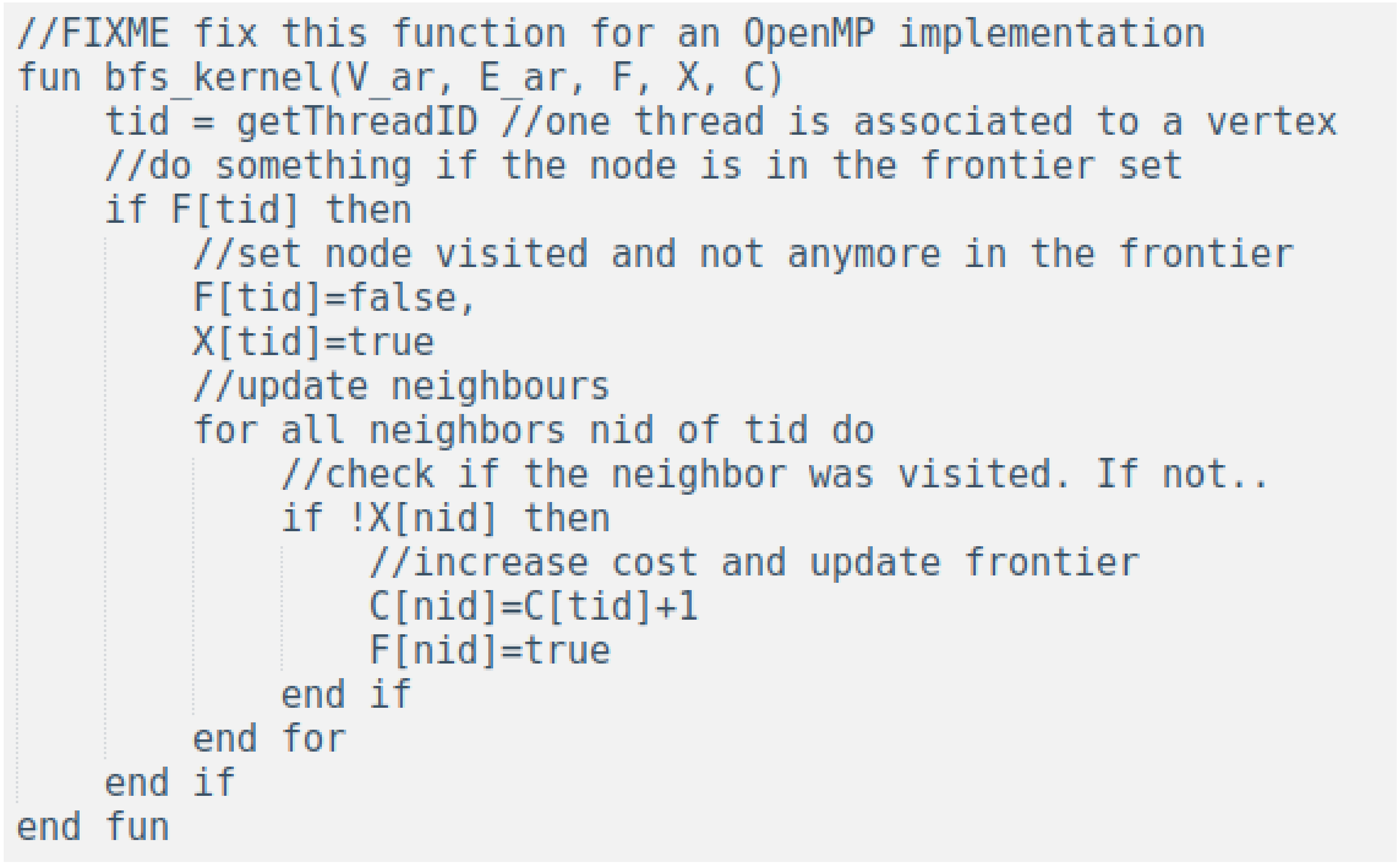
BFS – Breadth-First Search

- Visit the graph by levels. Once a level is visited it is not visited again.
- Support arrays (all of size |V|):
    - F is an array of boolean values.F[a]=true if the vertex a is in the frontier set at the moment, otherwise it is false.
    - V is an array of boolean values. V[a]=true if the node a had been visited, otherwise it is false.
    - C is an array of integer values. C[a] stores the minimal number of edges of each vertex from the source vertex S to a.
- We exploit parallelism when we update F, V and C

BFS – Main

```
fun cuda_bfs(Graph G(V,E), Source S)
    //init the structures
    Create vertex array V_ar from all vertices
        and edge Array E_ar from all edges in G(V, E)
    Create F, X and C of size V
    Initialize F and X to false and C to INF
    F[S]=true, C[S]=0
    //iterate until the frontier set is empty
    while F is not Empty do
        //update state through a parallel loop.
        for each vertex V_ar in parallel do
            //In openMp set the chunk size properly. For all vertex in the graph
            Bfs_Kernel(V_ar, E_ar, F, X, C)
        end for
    end while
end fun
```

BFS - kernel



```
//FIXME fix this function for an OpenMP implementation
fun bfs_kernel(V_ar, E_ar, F, X, C)
    tid = getThreadID //one thread is associated to a vertex
    //do something if the node is in the frontier set
    if F[tid] then
        //set node visited and not anymore in the frontier
        F[tid]=false,
        X[tid]=true
        //update neighbours
        for all neighbors nid of tid do
            //check if the neighbor was visited. If not..
            if !X[nid] then
                //increase cost and update frontier
                C[nid]=C[tid]+1
                F[nid]=true
            end if
        end for
    end if
end fun
```

SSSP - Main

```
fun sssp(Graph G(V,E,W), Source Vertex S)
    Create vertex array V_ar , edge array E_ar and weight array W a from G(V,E,W)
    Create mask array M, cost array C and Updating cost array U of size V
    Initialize mask M to false, cost array C and Updating cost array U to INF
    M[S]=true
    C[S]=0
    U[S]=0
    while M is not Empty do
        for each vertex V in parallel do
            sssp_kernel_1(V_ar, E_ar, W, M, C, U)
            sssp_kernel_2(V_ar, E_ar, W, M, C, U)
        end for
    end while
end fun
```

SSSP – Single-Source Shortes Path

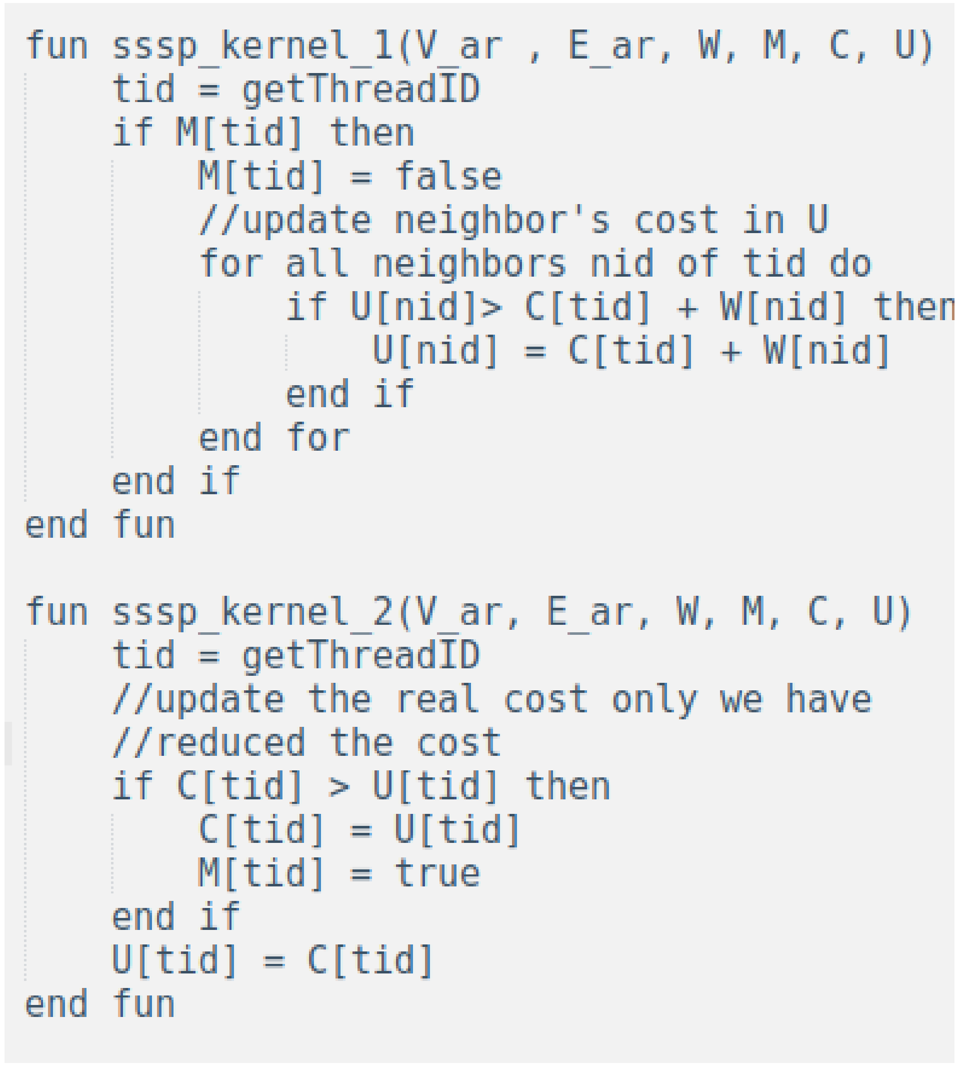
Support structures:

- **M is a boolean mask of size |V|**
- **C is the cost array from S of size |V|**
- **U is an alternate array of size |V|. It is a sort of buffer that contains the updated cost calculated in *sssp_kernel_1* function.**

SSSP - Logic

- **In each iteration each vertex checks if it is in the mask M:**
  - **If yes, it fetches its current cost from the cost array C and its neighbor’s weights from the weight array W. Update neighbor’s if this is >= than the cost of current vertex plus the edge weight to that neighbor. <u>But update the cost in U</u> (not in C)**
  - **After that, a second kernel compares cost C with updating cost U. It updates the cost C only if C[a]> U[a] and makes its own entry in the mask M[a]. The second stage of kernel execution is required as there is no synchronization between the CUDA multiprocessors (OpenMP can simplify this). Updating the cost at the time of modification itself can result in read after write inconsistencies.**

SSSP – Kernels



```
fun sssp_kernel_1(V_ar , E_ar, W, M, C, U)
    tid = getThreadID
    if M[tid] then
        M[tid] = false
        //update neighbor's cost in U
        for all neighbors nid of tid do
            if U[nid]> C[tid] + W[nid] then
                U[nid] = C[tid] + W[nid]
            end if
        end for
    end if
end fun

fun sssp_kernel_2(V_ar, E_ar, W, M, C, U)
    tid = getThreadID
    //update the real cost only we have
    //reduced the cost
    if C[tid] > U[tid] then
        C[tid] = U[tid]
        M[tid] = true
    end if
    U[tid] = C[tid]
end fun
```

APSP_FW – All Pairs Shortest Path Floyd-Warshall

- It is possible to test this algorithm with at most a thousand of vertices because the algorithm requires at $O(V^2)$ space.
- An adjacency matrix is used to implement the graph.
- It is implemented using V^2 threads running the classic CREW PRAM algorithm of the Floyd-Warshall.
- It also possible to implement it with O(V) threads that run an O(V) algorithm, but authors found this solution slower "*because of the sequential access of entire vertex array by each thread*". (But they implemented the algorithm with and adjacency matrix…)

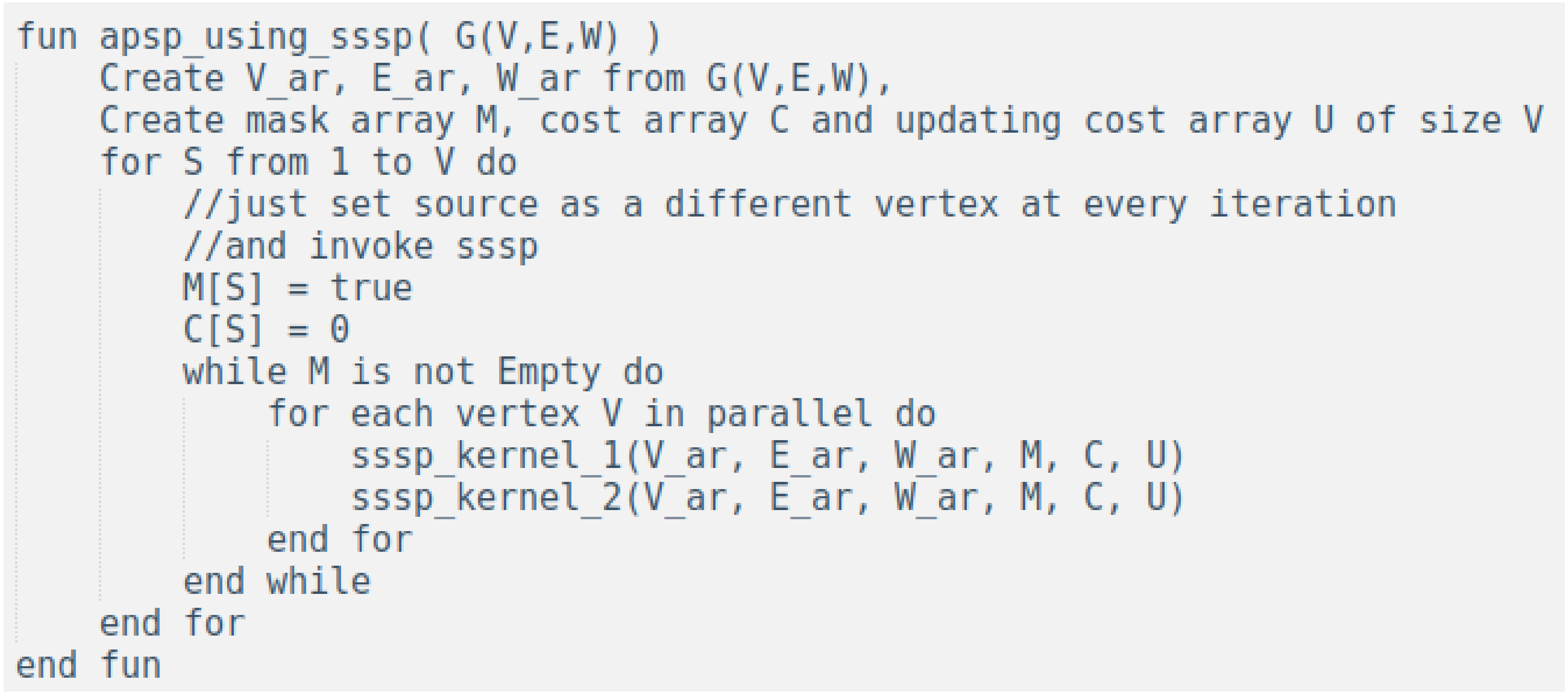
ASPS_FW - Algorithm

```
fun parallel_floyd_warshall( G(V,E,W) )
    //create the adjacency matrix
    Create adjacency Matrix A from G(V, E,W )
    for k from 1 to V do
        for all Elements in the A, where 1 ≤ i, j ≤ V in parallel do
            //build the result of the algorithm in the adjacency matrix
            A[i,j] = min( A[i,j], A[i,k]+A[k,j])
        end for
    end for
end fun
```

APSP_SSSP – APSP using SSSP

- This is faster than the Floyd-Warshal algorithm because this one requires a single O(V) operation looping over O(V^2) threads which creates extra overhead for context switching the threads on the SIMD processors.
- Authors parallelize SSSP algorithm. We could parallelize the Source selection and invoke the sequential SSSP algorithm.

APSP_SSSP - Algorithm



```
fun apsp_using_sssp( G(V,E,W) )
    Create V_ar, E_ar, W_ar from G(V,E,W),
    Create mask array M, cost array C and updating cost array U of size V
    for S from 1 to V do
        //just set source as a different vertex at every iteration
        //and invoke sssp
        M[S] = true
        C[S] = 0
        while M is not Empty do
            for each vertex V in parallel do
                sssp_kernel_1(V_ar, E_ar, W_ar, M, C, U)
                sssp_kernel_2(V_ar, E_ar, W_ar, M, C, U)
            end for
        end while
    end for
end fun
```

Experimental Results of the Paper

- PC with 2 GB RAM, Intel Core 2 Duo
- E6400 2.3GHz processor running Windows XP with one Nvidia GeForce 8800GTX.
- The graphics card has 768 MB RAM on board.
- For the CPU implementation, a PC with 3 GB RAM and an AMD Athlon 64 3200+ running 64 bit version of Fedora Core 4
- Applications were written in CUDA version 0.8.1 and C++ using Visual Studio 2005.
- Nvidia Graphics driver version 97.73 was used for CUDA compatibility.
- CPU applications were written in C++ using standard template library.

Experimental Results of the Paper

- **BFS for a 400 million vertex, 2 billion edges graph takes less than 5 seconds on a CRAY MTA-2, the 40 processor supercomputer, which costs 5–6 orders more than a CUDA hardware. They also implemented BFS on CPU, using C++ and found BFS on GPU to be 20–50 times faster than its CPU counterpart.**
- **SSSP timings are comparable to that of BFS for random graphs due to the randomness associated in these graphs. Since the degree per vertex is 6–7 and the weights vary from 1–10 in magnitude it is highly unlikely to have a less weighted edge coming back from a far away level. They compared the results with the SSSP CPU implementation and they found that the algorithm is 70 times faster than its CPU counterpart on an average.**

Experimental Results – Scale Free Graphs

- In such graphs a few vertices are of high degree while the rest are of low degree (Test done with the maximum degree of any vertex to be 1000 and average degree pervertex to be 6. A small fraction, 0.1%, of the total number of vertices were given high degrees).
- BFS and SSSP are slower for scale free graphs as compared to random graphs. Because of the large degree at some vertices, the loop inside the kernel increases, which results in more lookups to the device memory slowing down the kernel execution time. (Loops of non-uniform lengths are inefficient on a SIMD architecture).

Experimental Results - APSP

- A graph with 100K vertices, 6 degree per vertex, it takes around 22 minutes to compute sequential APSP. Authors found an average improvement of a factor of 3 for the Floyd Warshall CUDA algorithm and a factor of 17 for the all pair shortest path using SSSP CUDA implementation. As shown by the results APSP using SSSP is faster than Floyd Warshall’s APSP algorithm on the GPU. (It is also more scalable)

Experimental Results

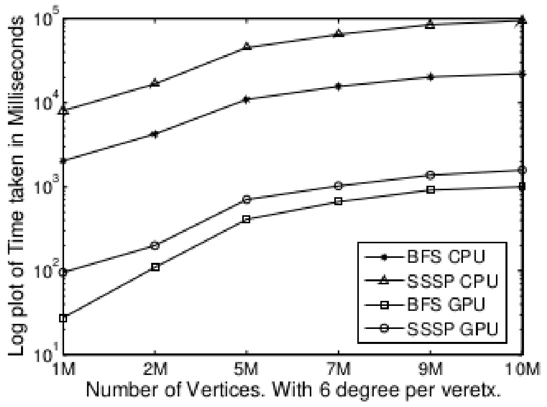


Fig. 4. BFS and SSSP times with weights ranging from 1-10

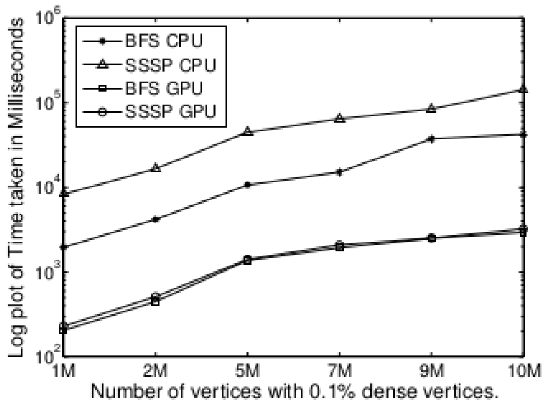


Fig. 5. BFS and SSSP times for Scale Free graphs, weights ranging from 1-10

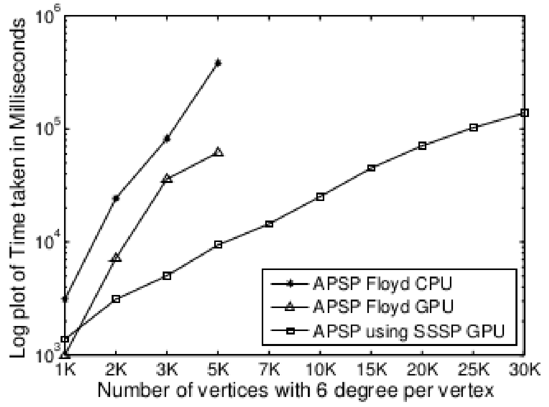


Fig. 6. APSP timings for various graphs, weights ranging from 1-10

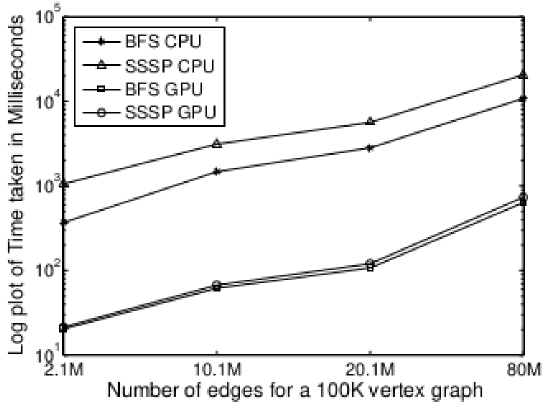


Fig. 7. Graphs with 100K vertices with varying degree per vertex, weights 1–10

Conclusions

- **It is possible to readapt algorithms from CUDA to OpenMP (improving them if possible)**
- **Performance differences between the paper implementation and the future OpenMP implementation will be evaluated (we can find out if a Laptop for the consumer’s market can do better than paper’s GPU).**

References

- **Nineth DIMACS implementation challenge (useful for datasets): http://www.dis.uniroma1.it/challenge9/download.shtml**
- **Accelerating large graph algorithms on the GPU using CUDA: https://cvit.iiit.ac.in/images/ConferencePapers/2007/Pawan07accelerating.pdf**