International Journal of Computer Applications (0975 – 8887)
Volume 111 – No 15, February 2015

OpenCL Parallel Blocked Approach for Solving All Pairs Shortest Path Problem on GPU

Manish Pandey
Department of Computer Science Engineering
Maulana Azad National Institute of Technology
Bhopal, India

Sanjay Sharma
Department of Mathematics and Computer Applications
Maulana Azad National Institute of Technology
Bhopal, India

ABSTRACT

All-Pairs Shortest Path Problem (APSP) finds a large number of practical applications in real world. This paper presents a blocked parallel approach for APSP using an open standard framework OpenCL, which provides development environment for utilizing heterogeneous computing elements of computer system and to take advantage of massive parallel capabilities of multi-core processors such as graphics processing unit (GPU) and CPU. This blocked parallel approach exploits the local shared memory of GPU, thereby enhancing the overall performance. The proposed solution is for directed and dense graphs with no negative cycles and is based on blocked Floyd Warshall (FW) and Kleene's algorithm. Like Floyd Warshall this approach is also in-place and therefore requires no extra memory.

General Terms

Heterogeneous Computing, Many core processing, GPU Computing, High Performance Computing.

Keywords

OpenCL, Graphics processing Unit, All Pairs Shortest Path, Floyd Warshall

1. INTRODUCTION

The all-pairs shortest path (APSP) targets to find the shortest path between every pair of vertices in a directed/undirected weighted graph, where cost is simply the sum of weights of edges composing the path. APSP may be solved by running a single source shortest path (SSSP) algorithm for all n vertices or by using APSP algorithms like Johnson algorithm or Floyd-Warshall (FW). APSP finds applications in various areas like geographical information system, intelligent transportation systems, IP routing [7], VLSI design etc.

A comparison of time complexities of different algorithms for SSSP and APSP is compared below. . In most of the cases an instance of the problem is represented in the form of directed weighted graph stored in the form of cost adjacency matrix of size $[n \times n]$

Consider a weighted graph G (V, E) stored in the form of weight adjacency matrix represented by a W where $w_{ij} \epsilon W$ for all $\langle i, j \rangle \epsilon E$. Each edge has an associated weight. Negative weigh cycles are not allowed

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of edge } \langle i, j \rangle & \text{if } i \neq j \text{ and } \langle i, j \rangle \epsilon E \\ \text{infinity} & \text{if } i \neq j \text{ and } \langle i, j \rangle \epsilon E \end{cases}$$

Algorithm	**Problem**	**Time Complexity**
Dijkstra	SSSP	$O(n^2)$
Floyd–Warshall	APSP	$O(n^3)$
Johnson	APSP	$O(n^2 \log n + ne)$

Although the theoretical time complexity of these well known algorithms is bounded by polynomial time, yet some applications require size of input data to be very large and therefore computational complexity for these well known algorithms also grow beyond practical limits.

GPU is not only a graphics engine to perform graphics acceleration tasks like gaming, rendering, image processing etc. In recent years GPUs spawned some new areas of research and programmability thus it is now referred to as general purpose GPU (GPGPU)[20][21]. Now various GPU implementations are proposed and GPU has become a cost effective platform for high performance computing (HPC). OpenCL provides a development environment for utilizing heterogeneous platforms and to take advantage of graphics processing unit (GPU).

Contribution of this paper: In this paper we have proposed an OpenCL blocked parallel approach for APSP based on Kleene's algorithm and compared the results with our previous implementation which involve parallel implementation of FW and parallel implementation of R-kleene. R-kleene works by recursively partitioning the matrix into sub-matrices and applies the computation on those sub-matrices. And in blocked approach aim is to utilize the GPU cache and shared memory. We have also used vectorization technique to improve blocked approach as it involves matrices.

Organization of the paper: Section 2 describes related work in the field of APSP. Section 3 comprises of OpenCL framework and different OpenCL models. Different parallel approaches to APSP based on Kleene's and Floyd-Warshall algorithm are explained in section 4 also an OpenCL parallel algorithms and tiled approach using Matrix-Multiply kernel is also explained. Experimental results are demonstrated in section 5. Section 6 presents conclusion and future work.

2. RELATED WORK

Parallel approaches for solving APSP using single source shortest path algorithm is discussed in [1][14] or by using parallel versions of APSP algorithms [4][18] or Johnson's Algorithm[21] The algorithms presented in these papers are in-place and are also capable of providing high level of parallelism but these algorithms cannot fully exploit architectural capabilities of GPU due to absence of high data reuse. Many algorithms have been proposed for solving APSP problem using Floyd-Warshall (FW) yet there is large scope in enhancing its performance. A divide and conquer approach using R-Kleene's algorithm have been proposed for dense graphs for APSP in [12]. This approach is in-place and recursive in nature. Challanges in parallel graph processing is discussed in[3]

Our computations involve matrices and therefore some fast matrix multiplication algorithms such as [19][6] are also our area of concern. As CPU implementations have several limitations of performance so some cache optimization techniques and cache friendly implementations are given in [2]and [5] using recursion for dense graphs. In [10] to reduce TLB misses blocked data layout and mortan layout are given for FW. In [2] block size is adjusted according to the cache parameters and matrix size to improve performance and to reduce cache misses. Our work is similar to the work by Venkataraman et. al [17] but unlike their work we have proposed OpenCL based implementation involving high level of parallelism, data reuse that fully exploits architectural benefits of GPU as a low cost computational resource.

GPU implementation of FW for smaller graphs is given in [8] and for larger graphs shared memory and cache efficient GPU implementations for APSP using FW are given in [16][9].To further enhance the performance some optimization techniques like tiling, loop unrolling and SIMD vectorization can be used.

3. OPENCL FRAMEWORK

OpenCL is an open standard framework for parallel programming composed of several computational resources (CPU, GPU and other processors). Thus one can achieve considerable acceleration in parallel processing as it utilizes all the computational resources. The main advantage with OpenCL is its portability as it provides cross vendor software portability [25].

OpenCL framework [15][23][24] comprises of following models:

3.1 OpenCL Platform Model

High level representation of heterogeneous system is demonstrated by Platform model as shown in Fig. 1. It consists of a host and OpenCL device. A host is any computer with a CPU and standard operating system. OpenCL device can be GPU, DSP or a multi-core CPU [25]. OpenCL device is collection of compute units which is further composed of one or more processing elements.

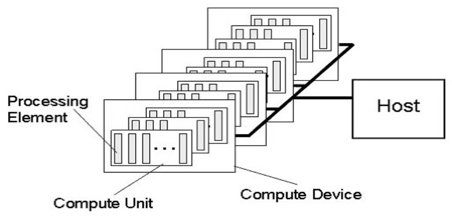

Fig.1 OpenCL Platform Model [23]

Processing elements within a compute unit will execute same instruction sequence while compute units can execute independently. Different GPU vendors follow different architectures but all follow a similar design pattern which is illustrated in Fig. 2.

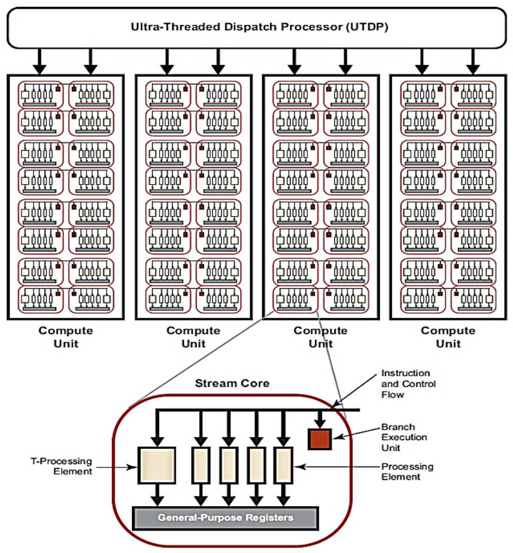

Fig.2 AMD GPU Compute Device [23]

3.2 OpenCL Execution Model

OpenCL execution model define how the kernel execution takes place and how kernel interact with host and with other kernels and it comprises of two components: kernels and host program. Kernels are further categorized into two types: OpenCL kernels and Native Kernels. Kernels execute on OpenCL devices and host execute on CPU (host system)

Workgroups evenly divide the index space of NDRange in each dimension. And the index space within a workgroup is referred as local index space which is defined for each work item. Size of index space in each dimension is indicated with uppercase and ID is indicated using lowercase. See figure 3

A work-item can be uniquely identified by its global ID(g_x, g_y) or by the combination of its local ID(l_x, l_y) and work group ID(w_x, w_y) as shown in relation below:

$$g_x = w_x * L_x + l_x \qquad (1)$$

$$g_y = w_y * L_y + l_y \qquad (2)$$

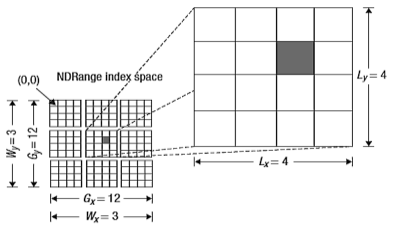

Fig.3 Relation between global ID and local ID, work-group ID in 2-D index space [23]

In fig.3, NDRange index space of size G_x by G_y $(12 * 12)$ is divided into 9 work-groups, each having size $(3 * 3)$. The shaded block has a global ID of $(g_x, g_y) = (6,5)$ and a work-group plus local ID of $(w_x, w_y) = (1,1)$ and$(l_x, l_y) = (2,1)$.

3.3 OpenCL Memory Model

OpenCL memory model defines different regions of memory and how they are related to platform and different execution model. This is shown in Fig. 4. There are generally five different regions of memory:

Host memory: This memory is limited to host only and OpenCL only defines the interaction of host memory with OpenCL objects.

Global memory: All work items in all work groups have read/write access to this region of memory and can be allocated only by the host during the runtime.

Constant memory: Region of memory which stays constant throughout the execution of kernel. Work-items have read only access to this region.

Local memory: Region of memory is local to work group. It can be implemented dedicatedly on OpenCL device or may be mapped on to regions of Global memory.

Private memory: Region that is private for work-item.

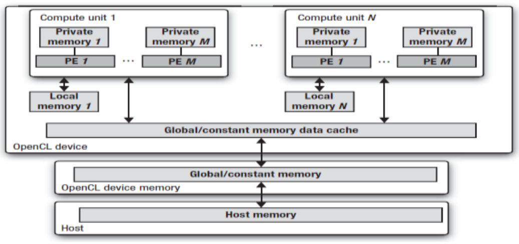

Fig.4 OpenCL Memory Model [23]

3.4 OpenCL Programming Model

A programmer can freely combine any of the programming models available in OpenCL. OpenCL is basically defined with two programming models: data parallel and task parallel model. However hybrid model can also be used.

4. OPENCL IMPLEMENTATION FOR SOLVING APSP PROBLEM

In the following sub-sections we owe to present an OpenCL blocked approach based on Kleene's algorithm [11], that was originally used for finding transitive closure and can be extended to shortest path problems also. The work embodied in the following sections is compared with OpenCL implementation of Floyd Warshall and Recursive Kleene's algorithms [26] and therefore OpenCL Floyd Warshall is our starting point

4.1 APSP Problem

APSP is the most fundamental problem in graph theory and our solution will follow a well-known algorithm called Floyd-Warshall (FW)[21]. FW sequential implementation uses three nested loops (fig.5).

Consider a weighted graph G (V, E) stored using adjacency matrix representation by a weight matrix W where $w_{ij} \in W$ for all $< i, j > \in E$.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of edge } \langle i,j \rangle & \text{if } i \neq j \text{ and } \langle i,j \rangle \in E \\ \text{infinity} & \text{if } i \neq j \text{ and } \langle i,j \rangle \in E \end{cases}$$

ALGORITHM FLOYD-WARSHALL(W)

```
1   n ← rows (W)
2   $D^{(0)}$ ← W
3   for k = 1 to n do
4       for i = 1 to n do
5           for j = 1 to n do
6               $d_{ij}^{(k)} \leftarrow min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7           end for
8       end for
9   end for
```

Fig.5 FW Algorithm pseudo code

4.2 OpenCL Parallel Implementation of Floyd-Warshall

Parallel implementation of FW algorithm requires N^2 work items to be created, where N is the total number of nodes. So, each work item (i, j) finds the shortest path between nodes i and j. In parallel implementation a 2-D kernel FW_KERNEL (A, k) is designed as shown in Fig. 7

Pseudo code for OpenCL parallel FW OpenCL_Parallel_FW (A, n) is shown in Fig.6, which calls kernel FW_KERNEL (A, k)

ALGORITHM OPENCL_PARALLEL_FW(A, N)

```
1   for k = 1 to n do
2       for all elements in matrix A, where 1≤ i , j ≤ n in parallel do
3           call FW_KERNEL (A, k)
4       end for
5   end for
```

Fig.6 Pseudo code for OpenCL parallel FW algorithm

KERNEL FW_KERNEL(A, K)

```
1   (i, j) ← getThreadID
2   A[i, j] ← min(A[i, j], A[i, k] + A[k, j])
```

Fig.7 Pseudo code for FW kernel in OpenCL

In each kth iteration of outermost for loop, n^2 work-items (threads) invokes kernel that computes shortest path between every possible pair of vertices $\langle i, j \rangle$ going through no vertex higher than vertex k , each using its thread_id (i, j), in parallel, where $1 \leq i, j \leq n$. In final iteration when k = n is completed, output matrix A will hold shortest path between every possible pair of vertices $\langle i, j \rangle$ going through no more vertex higher than vertex n that is the shortest distance between all-pairs of nodes.

4.3 Blocked approach to APSP problem using kleene's algorithm

The blocked approach for solving APSP is inspired by Kleene's algorithm [11] that was originally used for finding transitive closure for computing the existence of path between every possible pair of vertices $\langle i, j \rangle$ and is also applicable for shortest path problem in a closed semi ring. Algorithm is as shown in Fig.8

Kleene's algorithm divides the nodes of the graph into $n/\sqrt{s}$ zones as shown in Fig. 9. Nodes 1 to $\sqrt{s}$ will be in zone 1, nodes $\sqrt{s}$ +1 through $2\sqrt{s}$ will be in zone 2, and so on. Thus adjacency matrix corresponding to the graph is divided into n^2/s sub matrices each having size $\sqrt{s} \times \sqrt{s}$. A sub matrix M_{ij} refers all the edge from nodes in zone i to nodes in zone j. For √s = 4, nodes 1to 4 will be in zone 1, nodes 5 to 6 will be in zone 2 and so on nodes n - √s + 1 to n will be in zone $n/\sqrt{s}$ (Fig. 9)

ALGORITHM KLEENE'S_APSP (M, N, S)

```
1  /* Divide the graph 'M' into $n/\sqrt{s}$ zones */
2    for k = 1 to $n/\sqrt{s}$ do
3  /* Compute $M^*_{k,k}$, APSP solution using FW for $M_{k,k}$ */
4          $M_{k,k} = M^*_{k,k}$
5          for i = 1 to $n/\sqrt{s}$ do
6          for j = 1 to $n/\sqrt{s}$ do
7                          $M_{i,j} = M_{i,j} + M_{i,k} \times M_{k,k} \times M_{k,j}$
8                  end for
9          end for
10   end for
```

Fig.8 Kleene's Algorithm

		Zone 1	Zone 2	Zone 3	Zone $n/\sqrt{s}$
		1 2 3 4	5 6 7 8	9 . . .	. . n
Zone 1	1				
	2	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{1,n/\sqrt{s}}$
	3				
	4				
Zone 2	5				
	6	$M_{2,1}$	M_{22}	$M_{2,3}$	$M_{2,n/\sqrt{s}}$
	7				
	8				
Zone 3	9				
	.	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$	$M_{3,n/\sqrt{s}}$
	.				
	.				
Zone $n/\sqrt{s}$					
		$M_{n/\sqrt{s},1}$	$M_{n/\sqrt{s},2}$	$M_{n/\sqrt{s},3}$	$M_{n/\sqrt{s},n/\sqrt{s}}$
	.				
	n				

Fig.9 A n×n matrix having $n/\sqrt{s}$ Zones, where each zone is size √s × √s

Each entry $e_{ij} \in M_{ij}$ refers to the shortest path from every possible pair of vertex from zone 'i' to zone 'j' possibly going through some vertex in zone 'k' and is computed using $e_{ij} \mathrel{+}= \sum_{k=1}^{n} e_{ik} * e_{kj}$. Here operator '+' refers to 'min' and operator '*' refers to '+'

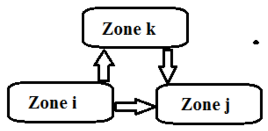

Fig.10 Shortest path from vertex 'x' to vertex 'y' through vertex 'z' such that x ∈ Zone I, vertex y ∈ Zone j, z ∈ Zone k

For n = 16 and √s = 4 Kleene's algorithms in fig.8 unrolls to following steps in the first iteration of outermost loop for k = 1.

Step 1	$M_{11} = M^*_{11}$
Step 2	$M_{11} \mathrel{+}= M_{11} * M_{11} * M_{11}$
Step 3	$M_{12} \mathrel{+}= M_{11} * M_{11} * M_{12}$
Step 4	$M_{13} \mathrel{+}= M_{11} * M_{11} * M_{13}$
Step 5	$M_{14} \mathrel{+}= M_{11} * M_{11} * M_{14}$
Step 6	$M_{21} \mathrel{+}= M_{21} * M_{11} * M_{11}$
Step 7	$M_{22} \mathrel{+}= M_{21} * M_{11} * M_{12}$
Step 8	$M_{23} \mathrel{+}= M_{21} * M_{11} * M_{13}$
Step 9	$M_{24} \mathrel{+}= M_{21} * M_{11} * M_{14}$
Step 10	$M_{31} \mathrel{+}= M_{31} * M_{11} * M_{11}$
Step 11	$M_{32} \mathrel{+}= M_{31} * M_{11} * M_{12}$
Step 12	$M_{33} \mathrel{+}= M_{31} * M_{11} * M_{13}$
Step 13	$M_{34} \mathrel{+}= M_{31} * M_{11} * M_{14}$
Step 14	$M_{41} \mathrel{+}= M_{41} * M_{11} * M_{11}$
Step 15	$M_{42} \mathrel{+}= M_{41} * M_{11} * M_{12}$
Step 16	$M_{43} \mathrel{+}= M_{41} * M_{11} * M_{13}$
Step 17	$M_{44} \mathrel{+}= M_{41} * M_{11} * M_{14}$

Fig.11 Loop unrolling for k = 1, n = 16 and √s = 4 in Kleene's Algorithm

Following simplifications can be performed on Kleene's Algorithm of fig. 8

I. For iterations when (i == j == k), the Line 7 of Kleene's algorithm (fig.8) is equivalent to $M_{k,k} = M_{k,k} + M_{k,k} \times M_{k,k} \times M_{k,k}$ (eq. Step 2 in fig.11). This step calculates the shortest path from every possible vertex from zone 'k' to zone 'k' (itself) going through zone 'k' (fig.10). Because shortest path between every possible pair of vertex in zone 'k' going through no zone higher than zone 'k' has already been computed in previous step using $M_{k,k} = M^*_{k,k}$ (eq. Step 1 in fig. 11), this statement is redundant and therefore can be eliminated.

II. For iterations when(i == k), Line 7 of Kleene's algorithm is equivalent to $M_{i,j}=M_{i,j} + M_{k,k} \times M_{k,k} \times M_{k,j}$. This step can be simplified to $M_{i,j}=M_{i,j} + M_{k,k} \times M_{k,j}$ using $M_{k,k} \times M_{k,k} = M_{k,k}$ (eq. Step 3 to 5 in Fig. 11)

III. For iterations when(j == k), Line 7 of Kleene's algorithm is equivalent to $M_{i,j}=M_{i,j} + M_{i,k} \times M_{k,k} \times M_{k,k}$. This step can be simplified to $M_{i,j}=M_{i,j} + M_{i,k} \times M_{k,k}$ using $M_{k,k} \times M_{k,k} = M_{k,k}$ (eq. Step 6, 10 and 14 in Fig. 11)

IV For iterations when $(i \neq j \neq k)$, the Line 7 of Kleene's algorithm (fig.8) is equivalent to $M_{i,j}=M_{i,j} + M_{i,k} \times M_{k,j}$ (eq. Steps 7, 8, 9, 11, 12 and 13) using either $M_{i,k} = M_{i,k} \times M_{k,k}$ or $M_{k,j} = M_{k,k} \times M_{k,j}$ that are already computed (eq. Step 6 or Step 3)

Performing above simplifications we yield the following modified Kleene's algorithm as shown in fig. 12

ALGORITHM MODIFIED_KLEENE'S_APSP_VER1(M, N, S)

```
1  /* Divide the graph 'M' into $n/\sqrt{s}$ zones */
2    for k = 1 to $n/\sqrt{s}$ do
3  /* Compute $M^*_{k,k}$, APSP solution using FW for $M_{k,k}$ */
4        $M_{k,k} = M^*_{k,k}$
5        for i = 1 to $n/\sqrt{s}$ do
6        for j = 1 to $n/\sqrt{s}$ do
7              if((i == j) &&(j ≠ k))
8                    $M_{i,j} = M_{i,j} + M_{k,k} \times M_{k,j}$
9              else
10               if((i ≠ j) &&( j== k ))
11                   $M_{i,j} = M_{i,j} + M_{i,k} \times M_{k,k}$
12               else
13                   $M_{i,j} = M_{i,j} + M_{i,k} \times M_{k,j}$
14             end for
15       end for
16   end for
```

Fig.12 Modified Kleene's Algorithm

For n = 16 and √s = 4 modified Kleene's algorithms in fig.12 unrolls to following steps (fig. 13) in the first iteration of outermost loop for k = 1.

Step 1	$M_{11} = M^*_{11}$
Step 2	$M_{12} \mathrel{+}= M_{11} * M_{12}$
Step 3	$M_{13} \mathrel{+}= M_{11} * M_{13}$
Step 4	$M_{14} \mathrel{+}= M_{11} * M_{14}$
Step 5	$M_{21} \mathrel{+}= M_{21} * M_{11}$
Step 6	$M_{22} \mathrel{+}= M_{21} * M_{12}$
Step 7	$M_{23} \mathrel{+}= M_{21} * M_{13}$
Step 8	$M_{24} \mathrel{+}= M_{21} * M_{14}$
Step 9	$M_{31} \mathrel{+}= M_{31} * M_{11}$
Step 10	$M_{32} \mathrel{+}= M_{31} * M_{12}$
Step 11	$M_{33} \mathrel{+}= M_{31} * M_{13}$
Step 12	$M_{34} \mathrel{+}= M_{31} * M_{14}$
Step 13	$M_{41} \mathrel{+}= M_{41} * M_{11}$
Step 14	$M_{42} \mathrel{+}= M_{41} * M_{12}$
Step 15	$M_{43} \mathrel{+}= M_{41} * M_{13}$
Step 16	$M_{44} \mathrel{+}= M_{41} * M_{14}$

Fig.13 Loop unrolling for k = 1, n = 16 and √s = 4 in modified Kleene's Algorithm of Fig.12

Following is the precedence graph for the steps in Fig 13 illustrating potential parallelism in modified Kleene's Algorithm of fig.12

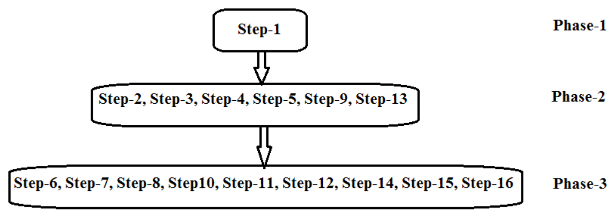
Step-1
Phase-1
Step-2, Step-3, Step-4, Step-5, Step-9, Step-13
Phase-2
Step-6, Step-7, Step-8, Step10, Step-11, Step-12, Step-14, Step-15, Step-16
Phase-3

Fig.14 Precedence graph showing parallelism in the steps of modified kleene's algorithm for k = 1, n = 16, √s=4

Each iteration of the outermost for loop consists of three phases that has to be executed sequentially but the set of steps within each phase are independent and therefore can be executed in parallel. Following is the algorithm that incorporates above phases.

ALGORITHM MODIFIED_KLEENE'S_APSP_VER2(M, N, S)

```
1 /* Divide the graph 'M' into $n/\sqrt{s}$ zones */
2   for k = 1 to $n/\sqrt{s}$ do
3        /* Phase I*/
4         $M_{k,k} = M^*_{k,k}$

5         /* Phase II */
6         for i = 1 to $n/\sqrt{s}$ such that (i ≠ k) do
7             $M_{k,i} = M_{k,i} + M_{k,k} \times M_{k,i}$
8             $M_{i,k} = M_{i,k} + M_{i,k} \times M_{k,k}$
9         end for

10       /*Phase III */
11         for i = 1 to $n/\sqrt{s}$ such that (i ≠ k) do
```

```
12              for j = 1 to  $n/\sqrt{s}$ such that (j ≠ k) do
13                      $M_{i,j} = M_{i,j} + M_{i,k} \times M_{k,j}$
14              end for
15          end for
16  end for
```

Fig.15 Three Phase Modified Kleene's Algorithm

4.4 OpenCL Parallel Implementation of three phase blocked algorithm

OpenCL implementation of blocked algorithm requires a host program to be designed that will execute over CPU and OpenCL kernels that will execute over OpenCL devices such as CPU and or GPU and other compute devices as specified in the host program. Each instance of the kernel can be executed concurrently by all the work-item that belongs to the same work group over single compute unit. All such work groups can be executed concurrently over compute units.

The algorithm outlined in fig. 15 requires $n/\sqrt{s}$ sequential iterations of the outermost for loop. The outermost for loop contains three phases that are required to be executed sequentially, within each iteration. The set of matrix operations within each phase can be executed concurrently. Thus for solving a two dimensional [N $\times$ N] graph problem a 2-dimensional NDRange is required to be created .Block size in a graph is chosen as $(\sqrt{s} \times \sqrt{s})$, where $'s'$ is the size of shared local memory of GPU. Thus if and when required the blocks in heavy reuse can be brought from global memory to local shared memory of the GPU to reduce memory latency. Thus for solving a [N $\times$ N] graph problem a two dimensional NDRange is formed where there are $(\sqrt{s} \times \sqrt{s})$ work-items per work-group. The number of work-groups in each dimension depends on the number of sub-matrices involved in a particular phase. Fig.16 outlines the algorithm for host program. Fig.17, Fig.18 and Fig.19 outlines the algorithm for Phase-1, Phase-2 and Phase-3 kernels respectively.

ALGORITHM FOR HOST PROGRAM IMPLEMENTING PARALLEL BLOCKED APPROACH

```
1   /* Divide the graph 'M' into  $N/\sqrt{s}$ zones having
    size $(\sqrt{s} \times \sqrt{s})$
2           BLOCKSIZE = $\sqrt{s}$;
3     for k = 1 to  $n/\sqrt{s}$ do
4                       // Phase1
5       //Set a two Dimensional Global Work group size$(\sqrt{s} \times \sqrt{s})$
6           globalWorkSize_p1[2] = {BLOCKSIZE , BLOCKSIZE};
7       //Set a two Dimensional Local Work group size$(\sqrt{s} \times \sqrt{s})$
8           localWorkSize_p1[2] = {BLOCKSIZE, BLOCKSIZE};
9           /* Invoke Phase1 kernel "Kernel_PH1"with given global and
10      local work group sizes and argument value 'k' that decides the
11      portion of the matrix in global/shared local memory where
12      instance of this kernel is executed by  work items concurrently*
13          Phase1_Kernel (kernel_PH1,globalWorkSize_p1,
14          localWorkSize_p1, &k);
                        //Phase2
15     /* Set a two Dimensional Global Work group
16    size $(2\sqrt{s} \times ((N/\sqrt{s}) - 1)\sqrt{s})$ */
17          globalWorkSize_p2[2];
18          globalWorkSize_p2[0] = $(N - BLOCKSIZE)$
19          globalWorkSize_p2[1] = 2*BLOCKSIZE;
20    //Set a two Dimensional Local Work group size$(\sqrt{s} \times \sqrt{s})$
21          localWorkSize_p2[2] = {BLOCKSIZE, BLOCKSIZE};
22          Phase2_Kernel (kernel_PH2, globalWorkSize_p2,
23          localWorkSize_p2, &k);
                        //Phase3
24     /* Set a two Dimensional Global Work group
25    size $((\times ((N/\sqrt{s}) - 1)\sqrt{s}) \times ((N/\sqrt{s}) - 1)\sqrt{s})$ */
26          globalWorkSize_p3[2];
27          globalWorkSize_p3[0] = $(n - BLOCKSIZE)$;
28          globalWorkSize_p3[1] = $(n - BLOCKSIZE)$;
29    //Set a two Dimensional Local Work group size$(\sqrt{s} \times \sqrt{s})$
30          localWorkSize_p3[2] = {BLOCKSIZE, BLOCKSIZE};
31          Phase3_Kernel (kernel_PH3, globalWorkSize_p2,
32          localWorkSize_p2, &k);
33     end for
```

Fig.16 Pseudocode for host program implementing OpenCL Blocked Parallel Approach.

Phase1_Kernel (globalWorkSize_p1, localWorkSize_p1, pblock)

```
1   /* kernel function for phase1, uses FW to solve APSP
2       "adjMbuff "stores adjacency matrix in device global memory*/
3   kernel_PH1( adjMbuff,  pblock)
4       {
5       //local thread id
6       int lxid = get_local_id(0);
```

```
    int lyid = get_local_id(1);
    // Calculates the offset of elements in Global memory
    int offset =pblock*BLOCKSIZE*N + pblock *BLOCKSIZE;
    // Reserve (√s × √s)space in local shared memory
    __local float Ms[BLOCKSIZE][BLOCKSIZE];
    // Transfer submatrix into local shared memory
        Ms[lyid][lxid] = adjMbuff[offset + lyid*N + lxid];
        barrier( );
        for(int k=0; k<BLOCKSIZE; k++)
        {
        float tempweight = combine(Ms[lyid][k], Ms[k][lxid]);
                if(tempweight < Ms[lyid][lxid])
                {
                        Ms[lyid][lxid] = tempweight;
                }
                barrier( );
        }
    /* Transferring sub-matrix from local shared memory back to
global memory*/
        adjMbuff[offset + lyid*N + lxid] = Ms[lyid][lxid];
        barrier( );
    }
```

Fig.17 Kernel for Phase 1

Phase2_Kernel (globalWorkSize_p1, localWorkSize_p1, pblock)

```
        //kerenl function for phase2, "adjMbuff "stores adjacency matrix
        in device global memory*/
  Kernel_PH2(  adjMbuff, pblock)
        {
        // global thread_ID
        int bxid = get_group_id(0);
        int byid = get_group_id(1);
        //local thraed id
        int lxid = get_local_id(0);
        int lyid = get_local_id(1);
        /* Reserve local shared memory for primary and current
           blocks*/
        __local float Ps[BLOCKSIZE][BLOCKSIZE];
        __local float Cs[BLOCK_SIZE][BLOCK_SIZE];
        // variable to skip primary block
        skip = (bxid < pblock) ? 0 : 1;
        if(byid == 0)
        {
        // Transferring blocks to local shared memory
        Ps[lyid][lxid] = adjMbuff [pblock*BLOCKSIZE*N +
                pblock*BLOCKSIZE + lyid*N + lxid];
        Cs[lyid][lxid] = adjMbuff[pblock*BLOCKSIZE*N +
                (bxid+skip)*BLOCKSIZE + lyid*N + lxid];
        barrier( );
        for(int k=0; k<BLOCKSIZE; k++)
        {
        float tempweight = combine(Ps[lyid][k], Cs[k][lxid]);
        if(tempweight < Cs[lyid][lxid])
        Cs[lyid][lxid] = tempweight;
        barrier( );
        }
        /* Transferring current matrix back from local shared
                memory to global memory*/
        adjMbuff [ pblock*BLOCK_SIZE*N +
       (bxid+skip)*BLOCK_SIZE + lyid*N + lxid] =
        Cs[lyid][lxid];
        barrier( );
        }
        else
        {
        // Transferring blocks to local shared memory
        Ps[lyid][lxid] = adjMbuff [pblock*BLOCKSIZE*N +
        pblock*BLOCK_SIZE + lyid*N + lxid];
        Cs[lyid][lxid] = adjMbuff[(bxid+skip)*BLOCKSIZE*N +
        pblock*BLOCK_SIZE + lyid*N + lxid];
        barrier( );
        for(int k=0; k<BLOCKSIZE; k++)
        {
        float tempweight = combine(Cs[lyid][k], Ps[k][lxid]);
        if(tempweight < Cs[lyid][lxid])
        Cs[lyid][lxid] = tempweight;
        barrier(CLK_LOCAL_MEM_FENCE);
        }
        /* Transferring current matrix back from local shared
         memory to global memory*/
        adjMbuff[(bxid+skip)*BLOCK_SIZE*N +
        pblock*BLOCK_SIZE + lyid*N + lxid] = Cs[lyid][lxid];
        barrier( );
        }
        }
```

Fig.18 Kernel for Phase 2

Phase3_Kernel (globalWorkSize_p1, localWorkSize_p1, pblock)

```
//kerenl function for phase2, “adjMbuff ”stores adjacency
matrix      in device global memory*/
Kernel_PH3( adjMbuff, pblock)
{
//block id
int bxid = get_group_id(0);
int byid = get_group_id(1);
//local thraed id
int lxid = get_local_id(0);
int lyid = get_local_id(1);
// Reserve space in local shared memory
//current block
__local float Cs[BLOCKSIZE][BLOCKSIZE];
//row block
__local float ROWs[BLOCKSIZE][BLOCKSIZE];
//column block
__local float COLs[BLOCKSIZE][BLOCKSIZE];
// variable to skip primary block in x dimension
 skipx = (bxid < pblock) ? 0 : 1;
// variable to skip primary block in y dimension
skipy = (byid < pblock) ? 0 : 1;
Cs[lyid][lxid] = adjMbuff[(byid+skipy)*BLOCKSIZE*N + (bxid+skipx)*BLOCK_SIZE + lyid*N + lxid];
COLs[lyid][lxid] = adjMbuff[pblock*BLOCK_SIZE*N + (bxid+skipx)*BLOCK_SIZE + lyid*N + lxid];
ROWs[lyid][lxid]=adjMbuff[(byid+skipy)*BLOCKSIZE*N + pblock*BLOCK_SIZE + lyid*N + lxid];
barrier( );
for(int k=0; k<BLOCK_SIZE; k++)
{
float tempweight = combine(ROWs[lyid][k],
COLs[k][lxid]);
if(tempweight < Cs[lyid][lxid])
Cs[lyid][lxid] = tempweight;
barrier( );
}
adjMbuff[(byid+skipy)*BLOCK_SIZE*N +
(bxid+skipx)*BLOCK_SIZE + lyid*N + lxid] =
Cs[lyid][lxid];
barrier( );
}
```

Fig.19 Kernel for Phase 3

5. EXPERIMENTAL RESULTS

We have tested OpenCL parallel blocked implementation on various GPUs and Intel CPU. Details of devices on which tests are performed, are given as follows:

- AMD Radeon HD 6450(GPU): 2 Compute units, 625 MHz clock, 2048MB Global Mem., 32KB Local Mem., 256 work group size on a system having Intel Core i5 CPU 650 @ 3.2 GHz and 2048MB RAM with AMD APP SDK v2.8.
- NVIDIA GeForce GT 630M (GPU): 2 Compute units, 950 MHz clock, 1023MB Global Mem., 48 KB Local Mem., 1024 work group size on a system having Intel Core i5 CPU-3210M @ 2.5GHz and 4096MB RAM with NVIDIA GPU computing SDK 4.2.
- AMD Radeon HD 6850 (GPU): 12 Compute units, 860 MHz clock, 1024MB Global Mem., 32KB Local Mem., 256 work group size on a system having Intel Core i3 CPU 530 @ 2.93 GHz and 4096MB RAM with AMD APP SDK v 2.8.
- Intel Core i3-2310M (CPU): 4 Compute units, 2095 MHz clock, 2048MB Global Mem., 32KB Local Mem., 1024 work group size with AMD APP SDK v2.8.

Results of the OpenCL parallel blocked implementation is compared with sequential Floyd Warshall on Intel Core i3-2310M (CPU): 2095 MHz clock, 2 GB RAM. We have also compared the results of OpenCL parallel blocked implementation with OpenCL parallel Floyd Warshall and OpenCL parallel R-Kleene [26] on various GPUs.

We have tested our results on various randomly generated dense graphs having edges of the order of $O(n^2)$. Random weight values between 1 to 10 are assigned to edges of graph. All results of parallel implementation for APSP problem are verified with FW sequential implementation on host CPU. For measuring time, we have considered total kernel execution time.

5.1 Results for OpenCL Parallel Blocked APSP Vs FW Iterative Sequential

Figure 20 shows, log plot of execution time in milliseconds and no. of nodes in a graph for OpenCL parallel blocked APSP implementation on various devices and also for FW sequential implementation. Parallel blocked APSP (BAPSP OCL DEVICE) out performs iterative sequential FW on all devices.

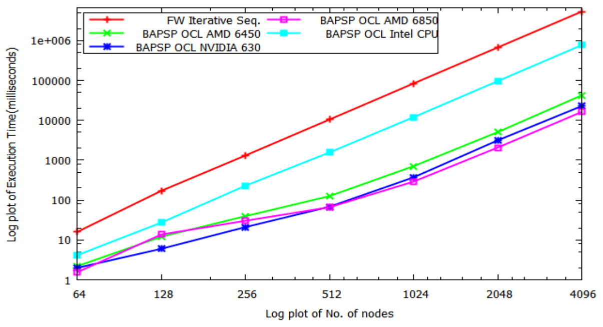

Fig.20 Execution time of OpenCL Parallel Blocked APSP and FW Iterative Sequential.

Figure 21 shows speedup for OpenCL parallel blocked APSP (OCL DEVICE) with respect to OpenCL parallel FW. It is evident from the figure that parallel blocked approach using OpenCL is up to 320x faster over AMD 6850 GPU, up to 200x faster over NVIDIA 630 GPU, up to 120x faster over AMD 6450 GPU and approx 10x faster over Intel CPU.

In figure 22 a comparison between OpenCL parallel blocked implementation Vs OpenCL parallel RKleene implementation over NVIDIA 630 GPU is presented. In figure 23 a comparison between OpenCL parallel blocked implementation Vs OpenCL parallel RKleene implementation over AMD 6850 GPU is presented. OpenCL parallel blocked APSP outperforms OpenCL recursive approach in both the cases due to explicit data reuse in phase-2 and phase-3. The sub matrices in heavy use can be explicitly moved to local shared memory of GPU and thereby improving memory latency. Also high level of parallelism is also involved in each phase. We have implemented OpenCL approach for parallel FW and parallel R-Kleene as outlined in [26].

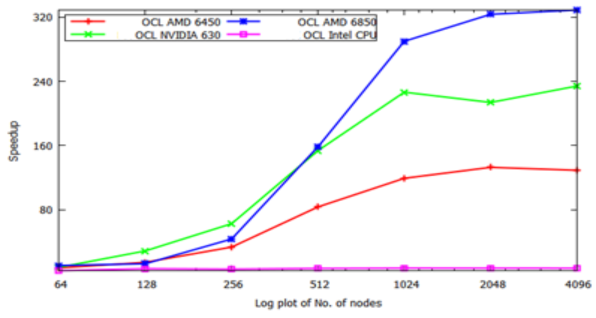

Fig.21. Speedup for OpeCL Parallel Blocked APSP w.r.t. OpenCL Parallel FW Approach[26]

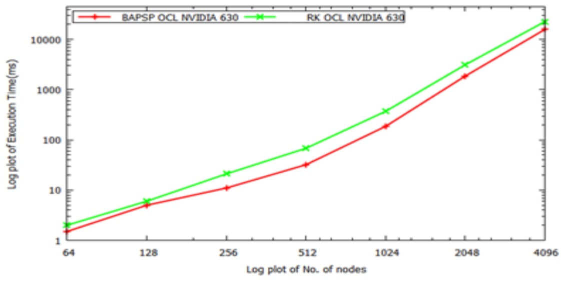

Fig.22. Execution time for OpenCL parallel blocked approach against OpenCL Parallel RKlene's approach on NVIDIA 630 GPU.

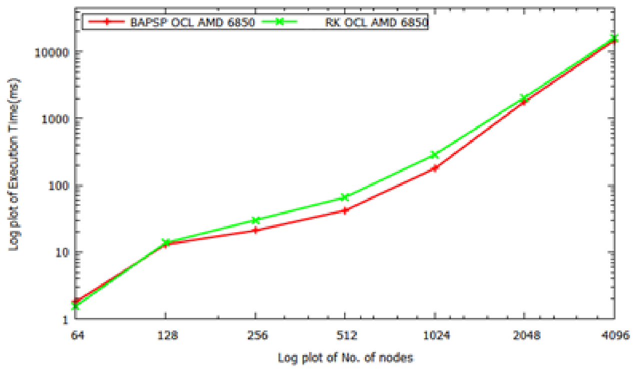

Fig.23. Execution time for OpenCL parallel blocked approach against OpenCL Parallel RKlene's approach on AMD 6850 GPU.

6. CONCLUSIONS AND FUTURE WORK

OpenCL blocked parallel implementation showed a significant speedup up to 320x on AMD 6850 GPU and up to 200x on NVIDIA 630 GPU, as compared to OpenCL parallel FW. This speedup is attributed to the fact that although high level of parallelism is involved in FW yet it is poor in data reuse. Blocked parallel approach outperforms OpenCL parallel recursive Kleene's approach as evident from fig. 22 and fig.23. In Recursive Kleene's approach (RKleene) there is high level of parallelism and data reuse but this data reuse is due to intrinsic characteristic of the recursive program. It is attributed to the program and not explicitly controlled by programmer. Only until the matrix is divided into sufficient smallest size (base case), shared memory cannot be used explicitly at each recursive call, but in case of parallel blocked approach there is high level of parallelism and high data reuse during phase-2 and phase-3. Block size is programmer dependent and can be chosen in a way so that sub-matrix can be accommodated in local shared memory. Thus sub matrices in heavy use can be moved to local shared memory and therefore considerable speedup can be gained.

In all our implementations so far only single OpenCL device CPU or GPU (but not both) is used for massive parallelism. OpenCL as a programming language can exploit the architectural benefits of heterogeneous and vendor independent computational devices. It would be interesting to develop an OpenCL approach that utilizes all such components by offloading an appropriate share of workload to these computational components.

7. REFERENCES

[1] P. Mateti, Cleveland, Ohio, N.Deo, "Parallel Algorithms for Single Source Shortest Path Problems" Computing 29 Springer, pp 31-49

[2] Rothberg, M.L.E., Wolfe, M., "The cache performance and optimizations of blocked algorithms". In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating System, pp. 63–74, 1991.

[3] Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," Parallel Processing Letters, vol. 17, no. 1, pp. 5–20, 2007

[4] A. Frieze and L. Rudolph, "A parallel algorithm for all pairs shortest paths in a random graph", Technical

Report, Dept. of Com. Sci., Carnegie-Mellon Univ. (1982).

[5] Park, J., Penner, M., Prasanna, V., “Optimizing graph algorithms for improved cache performance”. In: Proc. of International Parallel and Distributed Processing Symposium, 2002.

[6] K. Fatahalian, J. Sugerman, P. Hanrahan, “Understanding the efficiency of GPU algorithms for matrix–matrix multiplication”, in: HWWS ’04: Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS Conference, ACM, New York, 2004, pp. 133–137.

[7] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang. “IP routing processing with graphic processors”. In DATE ’10, March 2010.

[8] P. Harish, P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA”, in: Proc. of 14th Int’l Conf. High Performance Computing (HiPC’07), Dec. 2007.

[9] David A. Badar, K. Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2”, in: ICPP, pages 523-530, 2006.

[10] Penner, M., Prasanna, V., “Cache-friendly implementations of transitive closure”, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2001.

[11] Ullman, J., Yannakakis, M.: The input/output complexity of transitive closure. In: Proceedings of the1990 ACM SIGMOD International Conference on Management of Data, Volume 19, 1990.

[12] Paolo D’Alberto, A. Nicolau, “R-Kleene: a high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks”, Algorithmica 47 (2) (2007) pp. 203–213.

[13] NVIDIA OpenCL Resources, http://developer.nvidia.com/opencl.

[14] Floyd, R.: Algorithm 97: Shortest path. Communications of the ACM 5 (1962).

[15] OpenCL 1.2 reference pages, KHRONOS, 2012. http://www.khronos.org/registry/cl/sdk/1.2/docs/ man/xhtml.

[16] Gary J. Katz, Joseph T. KiderJr, “All-Pairs Shortest-Paths for Large Graphs on the GPU”, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pages 47-55, 2008.

[17] Venkataraman G., Sahni S., Mukhopadhyaya S., “A blocked all-pairs shortest-paths algorithm”. J. Exp. Algorithmics 8 (2003), 2.2.

[18] Han S.-C., Franchetti f., Püschel M., “Program generation for the all-pairs shortest path problem”. In: Parallel Architectures and Compilation Tech-niques (PACT) (2006), pp. 222–232.

[19] Larsen E., Mcallister D., “Fast matrix multiplies using graphics hardware”. In: Supercomputing, ACM/IEEE 2001 Conference (10-16 Nov. 2001), 43–43.

[20] Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., Purcell T, “A survey of general-purpose computation on graphics hardware”. In: Computer Graphics Forum 26, 1 (Mar. 2007), 80–113.

[21] Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest and Clittord Stein, “An Introduction To Algorithms”, McGraw-Hill Book Publication, First Edition, 1990.

[22] Owens J.D., Davis, Houston, M., Luebke, D., Green, S., “GPU Computing”, in: Proceedings of the IEEE, Volume: 96 , Issue: 5 , 2008.

[23] AMD Inc., “AMD Acclerated Parallel Processing OpenCL Programming Guide”, July 2012.

[24] A. Munshi, B. R. Gaster, T.G. Mattson, J. Fung, D. Ginsburg, “OpenCL Programming Guide”, Addison-Wesley pub., 2011.

[25] OpenCL Specification, http://www.khronos.org/ registry/cl/specs/opencl-1.2.pdf.

[26] Manish Pandey, Sanjay Sharma, “A Parallel Recursive Approach for Solving All Pairs Shortest Path Problem on GPU using OpenCL”, International Journal of Computer Science and Information Technologies(IJCSIT), ISSN:0975-9646,vol 5(6),2014,8198-8204