ISSN:0975-9646

A Parallel Recursive Approach for Solving All Pairs Shortest Path Problem on GPU using OpenCL

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. We owe to present a highly parallel and recursive solution for solving APSP problem based on Kleene's algorithm. The proposed parallel approach for APSP is implemented using an open standard framework OpenCL which provides a development environment for utilizing massive parallel capabilities of Multi core CPU and Many-Core-Processors such as Graphics Processing Unit (GPU). Moreover due to inherent nature of data reuse in the algorithm, shared memory of these processors is exploited to achieve considerable speedup. Our experiments demonstrate a speedup gain up to 521x over NVIDIA GeForce GT 630M GPU and a speedup up to 10x over Intel Core i3-2310M CPU.The proposed OpenCL solution for APSP is for directed and dense graphs with no negative cycles. Like Floyd-Warshall (FW), this approach is also in-place in nature and therefore requires no extra space.**

Keywords—OpenCL; Graphics processing Unit (GPU);All Pairs Shortest Path(APSP); in-place; FW(Flowd Warshall); RK (Recursive Kleene); Many Core Processors

I. INTRODUCTION

The All-Pairs Shortest Path(APSP) is a well-known problem in the field of computer science and has varied application areas inclusive of, but not restricted to, IP routing, wireless sensor networks, VLSI design and Artificial Intelligence. 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.

Consider a weighted graph G (V, E) stored in the form of weight adjacency matrix represented by W where $w_{ij} \in$ W for all $\langle i, j \rangle \in$ E.

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

Although the theoretical time complexity of well-known existing sequential approaches for solving APSP such as Floyd Warshall's [14] or Dijkstra's algorithm [21] is $O(n^3)$, and is reasonably good, but many applications such as VLSI design or wireless sensor networks, IP routing for large networks [7] etc requires size of input data to be very large [2] and in such cases time consumed by algorithm grows drastically beyond acceptable levels.

Parallel approaches for solving APSP may involve running a Single Source Shortest Path (SSSP) algorithm for all |V| vertices [3] or by using parallel versions of APSP algorithms [4][18]or Johnson's Algorithm[21]. Although these algorithms are in-place in nature and are capable of providing high level of parallelism but these algorithms lack inherent data reuse and therefore cannot fully exploit architectural capabilities of GPU. Bader et al. [6] have used supercomputer CRAY MTA-2 to perform breadth-first search on very large graph. A considerable high speed up can be achieved using such massive parallel computers but many core computers such as GPU (being specialized purpose processor) offers high performance at relatively very low cost. Penner et al[5][10] have presented algorithm optimization for improving cache performance but in contrast to this die(chip) area in case of Many Core Processors such as GPU, is dedicated largely for ALU's than cache [23] and therefore only locality of reference based optimization may not prove to be much useful.

Contribution of this paper: This paper presents a parallel recursive approach for solving APSP problem on large graphs using open standard programming framework OpenCL that exploits the architectural benefits of Multi Core Processors such as CPU or massive Many Core Processors such as GPU as a processing device. The key features of this approach are highly parallel, high data reuse and in-place nature

Following may be attributed to the key contributions of this paper

A high level of parallelism is obtained by appropriately distributing SIMD (Single Instruction Multiple Data) load over CPU/GPU cores in the form of work group

By appropriately choosing the block size to fit into local shared memory of the GPU to reuse the data in future

GPU provides large global data share. Blocked parallel approach is in-place in nature so no extra memory is needed for storing intermediate results and therefore large graph problems can also be solved

Solution is implemented using open standard programming framework OpenCL that can be executed

over wide variety of low cost hardware such as CPU, GPU, FPGA's and DSP's etc. and thus provides portable and cost effective solution. Rest of the paper is organized as follows.

Section II puts insight into a brief description of OpenCL platform model and GPU as a computational resource. Section III describes related work in the field of APSP. OpenCL Parallel approaches for solving APSP problem is discussed in section IV. Experimental results are demonstrated in section V and finally section VI presents Conclusion and scope for the future work.

II. OPENCL FRAMEWORK

OpenCL is an open standard framework for parallel programming composed of several computational resources like CPUs, GPUs, DSPs, and FPGAs etc. A considerable speed up can be achieved by utilizing all such computational resources. The main advantage with OpenCL as compared to its counterpart CUDA [15], is its portability among cross vendor hardware platforms [25].

OpenCL framework comprises of following components [23][24]:

A. OpenCL Platform Model

OpenCL platform model consists of CPU as a host and OpenCL devices such as CPU, GPU or other processors. An OpenCL device is a collection of compute units which is further composed of many processing elements

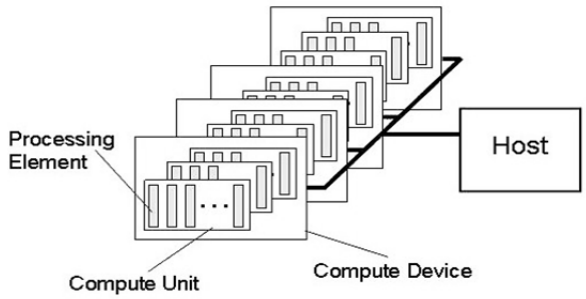

Fig. 1. OpenCL Platform Model [24]

All the processing elements within a compute unit will execute same sequence of instructions. While compute units can be executed independently

B. OpenCL Execution Model

OpenCL execution model comprises of two components: a host program that executes over CPU and kernel(s) that executes over OpenCL devices. OpenCL execution model provides a way for submitting kernels for execution. When this command is submitted it creates a collection of work-items (threads). Each work-item executes the same sequence of instructions as defined by a single kernel on different data selected through unique identification (global or local). Work-items can be organized in the form of workgroups that execute concurrently over the processing elements of the same compute unit.

C. OpenCL Memory Model

OpenCL memory model defines five different regions of memory and the way they are related to platform and execution models (Fig. 2).

- *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-tem.

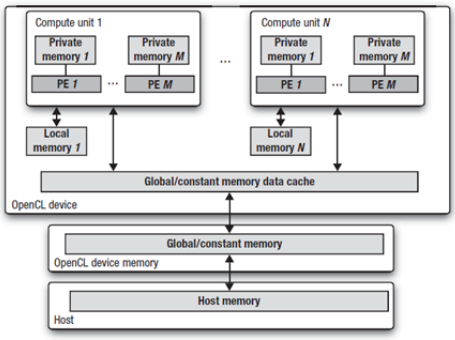

Fig. 2. OpenCL Memory Model [24]

D. OpenCL Programming Model

OpenCL supports two programming models: data parallel and task parallel models. However hybrid model can also be used.

III. RELATED WORK

Many algorithms have been proposed for solving APSP problem using Floyd-Warshall (FW) yet there is large scope in enhancing its performance. Using Johnson's algorithm, we can find all pair shortest paths in $O(n^2 logn + ne)$ time. Johnson's algorithm uses both Dijkstra [18] and Bellman-Ford [18] algorithms as subroutines. Owens J. D puts insight into GPU computing [22] and a survey on general-purpose computation on graphics hardware [20].As the instances of the problem involving graphs are generally represented in the form of adjacency matrix some fast matrix multiplication algorithm in [6][19] are also our area of concern.

The work presented in this paper is a recursive parallel OpenCL implementation of APSP that can execute on wide variety of many core processors and Kleene's algorithm [11] was our starting point

A sequential divide and conquer approach using R-Kleene's algorithm have been proposed for dense graphs for APSP in [12]. However our implementation is 512x times faster on larger graphs due to massive parallel work-items (threads) executing simple *Comp-Add* (Comparison-

Addition) operations on low power specialized purpose cores (ALU's)

Similar but independent to our work there are some CUDA based implementations for speeding up large graph algorithms on GPU [8][16], but they are for vendor specific NVIDIA[29] hardware. In contrast to this our solution runs on wide variety of platforms/vendor independent hardware.CPU implementations have several limitations of performance so some cache optimization techniques and cache friendly implementations are given in [1][5][10][17] using recursion for dense graphs. But unlike cache which is exploited by the virtue of locality of reference, local memory of GPU can be explicitly exploited and is programmer controlled.

IV. OpenCL Implementation For Solving APSP Problem

A. 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.

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

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

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) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7          end for
8       end for
9   end for
```

Fig. 3. FW Algorithm pseudocode

B. 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. 5

Pseudo code for OpenCL parallel FW *OPENCL_PARALLEL_FW (A, N)* is shown in Fig.4, 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
    end for
```

Fig. 4. 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. 5. Pseudo code for FW kernel in OpenCL

In each k^{th} 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.

The theoretical time complexity of parallel Floyd Warshall's algorithm is O(n) because outermost for loop executes sequentially O(n) times where as inner parallel for statement takes O(1) time, assuming n^2 processing elements

C. In-place Parallel Recursive approach to APSP problem using kleene's algorithm

The recursive approach for solving APSP is inspired by Kleene's algorithm [2], as shown in Fig.6, for finding transitive closure that computes the existence of path between every possible pair of vertices $\langle i, j \rangle$.

Kleene's algorithm divides the nodes of the graph into $n/\sqrt{s}$ zones as shown in Fig.7. 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.

ALGORITHM KLEENE'S_TANSITIVE_CLOSURE(A, N)

```
1/* Divide the graph 'A' into n/√s zones */
2  for k = 1 to n/√s do
3       /* Compute M*_k,k, the transitive closure of M_k,k */
4       M_k,k = M*_k,k
5       for i = 1 to n/√s do
6       for j = 1 to n/√s do
7                   M_i,j = M_i,j + M_i,k × M_k,k × M_k,j
8       end for
9       end for
10  end for
```

Fig. 6. Pseudo code for FW kernel in Open

$M_{1,1}$	$M_{1,2}$	⋯	$M_{1,n/\sqrt{s}}$
$M_{2,1}$	$M_{2,2}$	⋯	$M_{2,n/\sqrt{s},}$
⋮	⋮		⋮
$M_{n/\sqrt{s},1}$	$M_{n/\sqrt{s},2}$	⋯	$M_{n/\sqrt{s},n/\sqrt{s}}$

Fig. 7. A n×n matrix having $n/\sqrt{s}$ Zones

Each entry $e_{ij} \in M_{ij}$ refers to the shortest path from every possible vertex from zone 'i' to zone 'j' going through no more zone greater than zone 'k' and is computed using $e_{ij} += \sum_{k=1}^{n} e_{ik} * e_{kj}$. Here operator '+' refers to 'min' and operator '*' refers to '+'

For $n/\sqrt{s} = 2$, Kleene's algorithm in Fig. 6 unrolls to following 10 steps

$M_{11} = A$ Zone 1	M_{12}= B
$M_{21} = C$	M_{22}=D Zone 2

Fig. 8 A n × n matrix divided into $n/\sqrt{s}$ =2 Zones

KLEENE'S ALGORITHM LOOP UNROLLING FOR $n/\sqrt{s}$ =2 ZONES

1	$M_{11} = M_{11}^*$
2	$M_{11} += M_{11} * M_{11} * M_{11}$
3	$M_{12} += M_{11} * M_{11} * M_{12}$
4	$M_{21} += M_{21} * M_{11} * M_{11}$
5	$M_{22} += M_{21} * M_{11} * M_{12}$
6	$M_{22} = M_{22}^*$
7	$M_{11} += M_{12} * M_{22} * M_{21}$
8	$M_{12} += M_{12} * M_{22} * M_{22}$
9	$M_{21} += M_{22} * M_{22} * M_{21}$
10	$M_{22} += M_{22} * M_{22} * M_{22}$

Following simplifications are performed to obtain algorithm in Fig.9

Step 1 :$M_{11} = M_{11}^*$

$M_{11}^*$is the matrix closure of M_{11}. It calculates shortest path between every possible pair of vertices in zone1 (ranging from 1 to $n/2$) going through no more vertex higher than vertex $n/2$. This solves smaller sized sub problem consisting of first $n/2$ vertices.

Step 2:$M_{11} += M_{11} * M_{11} * M_{11}$

This computation is redundant and therefore can be removed because if A is a matrix closure (shortest path matrix) then A*A=A

Step 3: $M_{12} += M_{11} * M_{11} * M_{12}$

Calculates shortest path between every possible pair of vertices from zone 1 (ranging from 1 to $n/2$) to zone 2 (ranging from $n/2 + 1$ to n going through no more vertex higher than vertex in zone 1. This computation can be simplified to $M_{12} += M_{11} * M_{12}$ by following $M_{11} * M_{11} = M_{11}$

Step 4: $M_{21} += M_{21} * M_{11} * M_{11}$

Calculates shortest path between every possible pair of vertices from zone 2 (ranging from $n/2 + 1$ to n) to zone 1 (ranging from 1 to $n/2$) going through no more vertex higher than vertex in zone 1.This computation can be simplified to $M_{21} += M_{21} * M_{11}$ by following $M_{11} * M_{11} = M_{11}$

Step 5: $M_{22} += M_{21} * M_{11} * M_{12}$

Calculates shortest path between every possible pair of vertices from zone 2 (ranging from $n/2 + 1$ to n) to zone 2 (ranging from $n/2 + 1$ to n) going through no more vertex higher than vertex in zone 1.This computation can be simplified to $M_{22} += M_{21} * M_{12}$ by following either step 3 or step 4

Step 6: Computation on line 7 can be postponed to the last without affecting the end result

Following similar simplifications at Line 6, 7, 8, 9 and 10 following algorithm is reached

	KLEENE_ALGO	KLEENE_ALGO	REC_KLEENE_ALGO
1	$M_{11} = M_{11}^*$	$A = A^*$	$A = Recur(A)$
2	$M_{12} += M_{11} * M_{12}$	$B += A * B$	$B += A * B$
3	$M_{21} += M_{21} * M_{11}$	$C += C * A$	$C += C * A$
4	$M_{22} += M_{21} * M_{12}$	$D += C * B$	$D += C * B$
5	$M_{22} = M_{22}^*$	$D = D^*$	$D = Recur(D)$
6	$M_{12} += M_{12} * M_{22}$	$B += B * D$	$B += B * D$
7.	$M_{21} += M_{22} * M_{21}$	$C += D * C$	$C += D * C$
8.	$M_{11} += M_{12} * M_{21}$	$A += B * C$	$A += B * C$

Fig. 9 Simplification of Kleene's Operations.

Here Matrix multiplication (MM) operation $X = X + Y * Z$ is computed using $X_{ij} = X_{ij} + \sum_{k=1}^{n} Y_{ik} * Z_{kj}$ where scalar addition $(+)$ is minimum of two numbers i.e. $a + b = \min(a, b)$and scalar multiplication $(*)$ is addition i.e. $a * b = a + b$. These MMs are defined in closed semi ring. It is clear from the algorithm that it is recursive and in-place in nature and uses data locality to improve cache performance. Recursive calls are made to A and D as mentioned in Step 1 and Step 5.

D. OpenCL Parallel Implementation of Recursive Kleene's Algorithm

In recursive Kleene's algorithm step 2 and step 3 can be executed in parallel, likewise step 6 and step 7 can be executed in parallel. Fig.10 represents the precedence graph for the steps in recursive algorithm

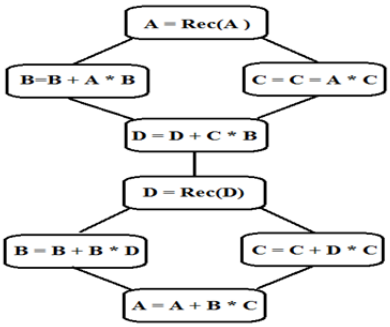

Fig.10 Precedence graph kleene's operation

As OpenCL does not support recursion so we have implemented recursive function in host program which calls OpenCL kernel recursively as shown in Fig.11. This recursive nature exploits data locality and increases data reuse ratio.

ALGORITHM OPENCL-PARALLEL-RK(J, N)

```
/* J is n × n matrix */
1    if (base case)
2        call OpenCL-Parallel-FW(J, n)
3    else
4        divide matrix J in matrices A, B, C & D
5        n ← n/2
6        call OpenCL-Parallel-RK(A, n)
7        for all n² matrix elements in parallel do
8            call RKMM_KERNEL (B, A, B, n)
9        end for
10       for all n² matrix elements in parallel do
11           call RKMM_KERNEL (C, C, A, n)
12       end for
13       for all n² matrix elements in parallel do
14           call RKMM_KERNEL (D, C, B, n)
15       end for
16       call OpenCL-Parallel-RK(D, n)
17       for all n² matrix elements in parallel do
18           call RKMM_KERNEL (B, B, D, n)
19       end for
20       for all n² matrix elements in parallel do
21           call RKMM_KERNEL (C, D, C, n)
22       end for
20       for all n² matrix elements in parallel do
21           call RKMM_KERNEL (A, B, C, n)
22       end for
```

Fig.11. Pseudo code for OpenCL parallel implementation of in-place recursive R-Kleene

The matrix multiply kernel which is called in each recursive calls of OpenCL-Parallel-RK(J, n) is shown in Fig. 12.

KERNEL RKMM_KERNEL(X, Y, Z, N)

```
1    (i, j) ← getThreadID
2    for k=1 to ndo
3        X[i, j] ← min(X[i, j], Y[i, k] + Z[k, j])
4    end for
```

Fig.12. Pseudo code for OpenCL naive MM kernel

The stopping condition for this recursive implementation is when matrix size (i.e. n) becomes equal to a threshold value which is also called the base case of our algorithm. In our scenario base case is when matrix can be fitted in a single workgroup which is 16x16 in case of our GPU. And matrix of this size can also be fitted in shared memory. Kernel of base case is shown in Fig. 13.

KERNEL FW_BASE_KERNEL(A, N)

```
1        (i, j) ← getThreadID
2        Ms[16,16] //block in Shared memory (SM)
3        copy related block from Global memory (GM) to
Ms
4        for k ← 1 to n do
5        Ms[i, j] ← min (Ms[i, j], Ms[i, k] + Ms[k, j])
6        end for
7        copy block Ms from SM to GM
```

Fig.13. FW kernel pseudo code for base case

V. EXPERIMENTAL RESULTS

We have implemented sequential Floyd Warshall's algorithm (FW_Sequential) and sequential recursive Kleene's Algorithm (RK_Sequential). The sequential versions of the algorithms have been tested on Intel Core i3-2310M (CPU): 2095 MHz clock, 2 GB RAM.

We have also implemented parallel versions of Floyd Warshall's algorithm (FW_Parallel) and four OpenCL parallel recursive Kleene's algorithm (RK_Parallel_AMD6450, RK_Parallel_AMD6850, RK_parallel_NVIDEA, RK_Parallel_Intel)on four different hardware platforms using OpenCL SDK 2.0 as given below

Intel Core i3-2310M (CPU): 4 Compute units, 2095 MHz clock, 2048MB Global Memory., 32KB Local Memory, 1024 work group size with AMD APP SDK v2.8.

AMD Radeon HD 6450(GPU): 2 Compute units, 625 MHz clock, 2048MB Global Memory., 32KB Local Memory., 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.

AMD Radeon HD 6850 (GPU): 12 Compute units, 860 MHz clock, 1024MB Global Memory, 32KB Local Memory, 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.

NVIDIA GeForce GT 630M (GPU): 2 Compute units, 950 MHz clock, 1023MB Global Memory, 48 KB Local Memory, 1024 work group size on a system having Intel Core i5 CPU-3210M @ 2.5GHz and 4096MB RAM

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.

A. Results of OpenCL parallel vs Serial Implementaion

In Fig. 14, log plot of execution time in milliseconds and no. of nodes in a graph is presented. OpenCL parallel implementation for RK based approach is tested on various GPU devices and also on CPU device. Timings for RK & FW sequential implementation are also shown.

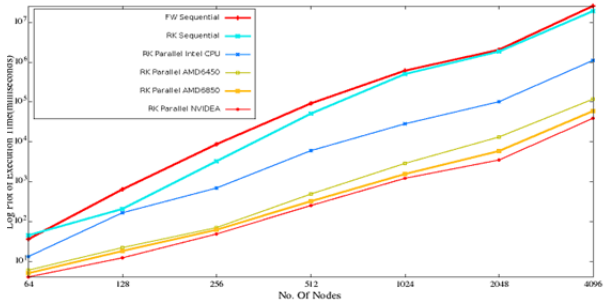

Fig.14 Execution time of sequential and OpenCL parallel Algorithms

Speedup for RK based OpenCL implementation with respect to FW sequential is shown in Fig. 16 and speedup for RK based OpenCL implementation with respect to RK sequential is shown in Fig. 15

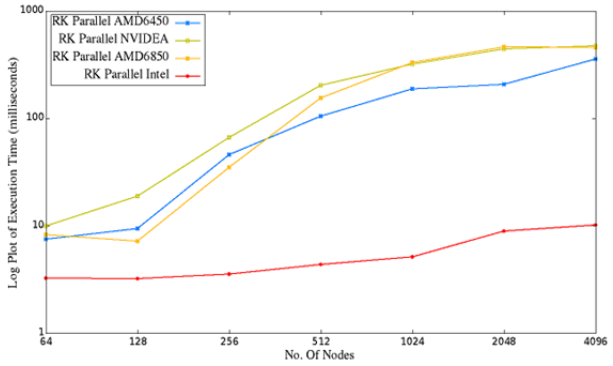

Fig.15. Speedup for RK parallel implementation w.r.t. RK serial implementation

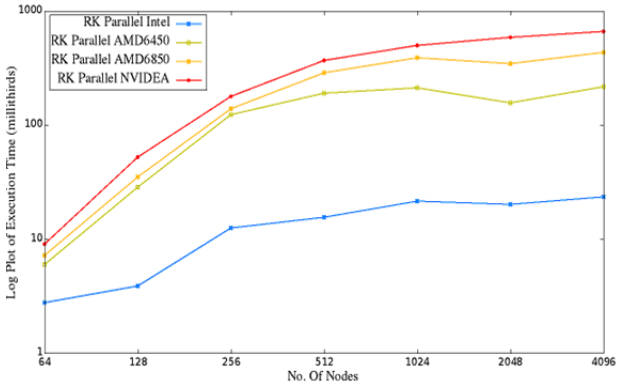

Fig.16. Speedup for RK parallel implementation w.r.t. FW serial implementation

In Table 1, speedup comparison for RK based OpenCL parallel implementation on various devices over RK serial CPU implementation is shown.

TABLE 1. SPEEDUP COMPARISON FOR VARIOUS DEVICES

No. of nodes	Intel CPU	AMD 6450 GPU	AMD 6850 GPU	NVIDIA GT 630M GPU
64	3.4	7.5	9.0	11.2
128	3.5	9.3	11.4	17.1
256	4.7	47.8	51.7	66.7
512	5.9	111.4	157.1	202.9
1024	6.3	171.6	317.9	409.3
2048	9.7	160.2	308.5	521.9
4096	10.4	165.6	318.6	489.7

B Comparison between parallel RK & parallel FW implementation on same GPU

Fig. 17, Fig.18 and Fig. 19 shows comparison between RK OpenCL parallel and FW OpenCL parallel implementation on various GPUs. Our RK based OpenCL implementation takes lesser time in comparison to FW OpenCL implementation on same GPU device.

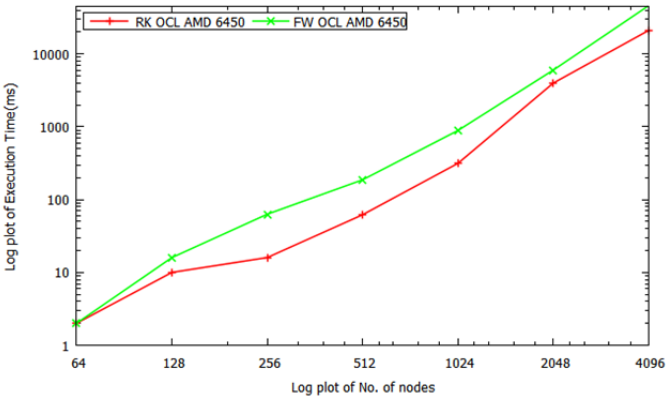

Fig.17. Comparison between RK based OpenCL parallel and FW OpenCL parallel implementation on AMD 6450 GPU

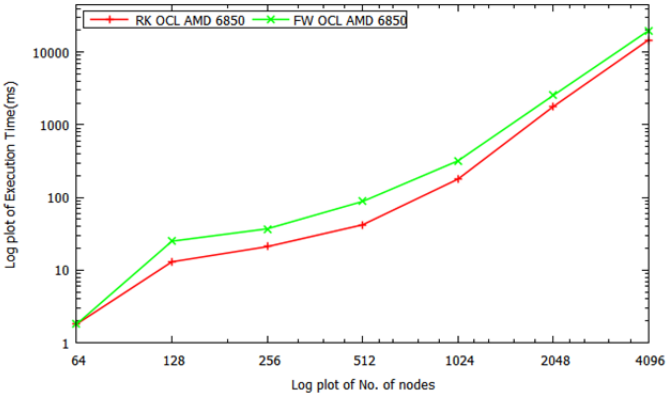

Fig.18. Comparison between RK based OpenCL parallel and FW OpenCL parallel implementation on AMD 6850 GPU

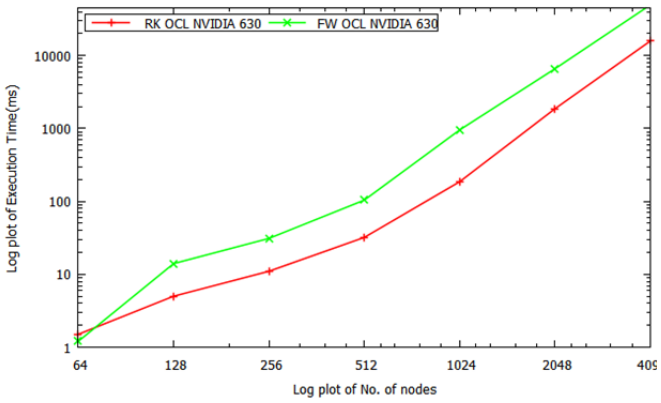

Fig.19. Comparison between RK based OpenCL parallel and FW OpenCL parallel implementation on NVIDIA 630 GPU

VI. Conclusions And Future Work

OpenCL parallel implementation showed a significant speedup up to 521x on NVIDIA GPU's, up to 318x on AMD 6850 GPU, up to 171x on AMD 6450 GPU and up to 10 x on Intel CPU with respect to RK serial implementation. Also it is clear from execution time plots in Fig. 17, Fig. 18 and Fig.19 OpenCL parallel recursive algorithm shows a significant speedup over OpenCL parallel Floyd Warshall's algorithm over same GPU.

As OpenCL can execute on wide variety of platforms there is a significant scope for hybrid implementation for solving APSP problem that involves heterogeneous computing environment consisting of CPU and GPU both. An appropriate portion of the work can be offloaded to CPU and GPU so as to utilize these processing devices at its best and to obtain further speedup.

References

[1] 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.

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

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

[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] Govindaraju N. K., Manocha D., "Cache efficient numerical algorithms using graphics hardware". Parallel Computing 33, 10-11 (2007), 663–684.

[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.