International Journal of Computer Applications (0975 – 8887)
Volume 77 – No.9, September 2013

Performance Analysis of Single Source Shortest Path Algorithm over Multiple GPUs in a Network of Workstations using OpenCL and MPI

Krishnahari Thouti
Dept. of CSE
VNIT, Nagpur
India, pin-440010

S. R. Sathe
Dept., of CSE
VNIT, Nagpur
India, pin – 440010

ABSTRACT
Graphics Processing Units (GPUs) are being heavily used in various graphics and non-graphics applications. Many practical problems in computing can be represented as graphs to arrive at a particular solution. These graphs contains very large number, up to millions pairs of vertices and edges. In this paper, we present performance analysis of Dijkstra's single source shortest path algorithm over multiple GPU devices in a single machine as well as over a network of workstations using OpenCL and MPI. Experimental results prove that parallel execution of Dijkstra's algorithm has good performance when algorithm is run over multi-GPU devices in a single workstation as opposed to multi-GPU devices over a network of workstations. For our experimentation, we have used workstation having Intel Xeon 6-core Processor; supporting hyper-threading and a total of 24 threads with NVIDIA Quadro FX 3800 GPU device. The two GPU devices are connected by SLI Bridge. Overall, on average we achieved performance improvement up to an order of 10-15x.

General Terms
Parallel Computing, Parallel Algorithms

Keywords
GPU Computing; OpenCL; Multi-node GPU Cluster; Dijkstra's algorithm; Single source shortest path

1. INTRODUCTION
In computer science, graphs describe relationship between various objects. Many practical problems in computing, networking, data analysis, decision making, linear programming, computational biology and other areas can be modeled as graphs to solve such problems. As the size of graph grows, complexity of problem solving becomes enormous. Hence, it becomes necessary to utilize the computational powers of Graphics processing Unit (GPU) [1].

Graphics Processing Unit (GPU) is a highly parallel, multithreaded; many core processor with tremendous computational power and very high memory bandwidth. GPUs use aggressive multithreading so that whenever thread is stalled, waiting for data, the thread can efficiently switch to execute another thread. Parallel programming languages such as Brook+ [2], NVidia's CUDA [3], OpenCL (Open Computing Language) [4], have been recently introduced to help programmers in writing parallel programs to take benefit of GPUs for high performance computing.

A graph is defined in terms of pairs of number of vertices and edges. The time-complexity of typical sequential version of graph algorithm is in the order of number of vertices and edges. As the number of vertices and edges increases, graph algorithms takes large amount of time. Graph algorithms can harness the powers of GPU, if graph can be efficiently expressed in terms of set of parallel and un-parallel computations. The Parallel Boost Graph Library (ParaBGL) [5] is a library which provides such a facility for parallel and distributed computations. ParaBGL offers essential data structures, algorithms and syntax for very large graphs; for solving such large-scale graph problems in distributed and parallel environment.

SnuCL [6] is an OpenCL framework for heterogeneous CPU-GPU clusters with MPI [7]. SnuCL allows the application to utilize compute devices in a compute node as if they were in the host node. As a result, OpenCL application compiled with SnuCL generates MPI+OpenCL kernels that can run on the cluster without any modifications.

In this paper, we present implementation details of Dijkstra's single source shortest path algorithm on (i) single GPU (ii) two GPUs on a single machine (iii) two GPUs over LAN using OpenCL programming model, MPI library and SnuCL framework.

The paper is organized a follows. Section 2 describes prior work done on Dijkstra's algorithm. A precise description about architecture of GPU device and OpenCL programming model is given in Section 3 and 4 respectively. In Section 5, we review Dijkstra's algorithm and present various parallel version of Dijkstra's algorithm for single GPU, dual GPUs. Experimental results are presented in Section 6 and finally Section 7 concludes and presents future scope.

2. RELATED WORK
There are many papers available in literature involving graph algorithms [8, 9, 10, 11, 12, 13, 14]. Parallel shortest path algorithm was implemented in [15] for a super-computer. They map logical processors of machine to physical processing nodes of supercomputer to parallelize Dijkstra's algorithm using Hamiltonian cycles and priority queues.

An efficient GPU implementation of parallel global path-finding SSSP algorithm using the CUDA programming environment is presented in [16] and they achieved very good performance over irregular and divergent algorithms. In [17] authors propose GPU implementation of Dijkstra's algorithm and they call it as Parallel Hardware-Accelerated Shortest Path Tress (PHAST). However, PHAST only works with low high way dimensions [18]. All pairs shortest paths algorithm is described in [19] including single source shortest path algorithm for large graphs using CUDA API. It describes a shared memory cache efficient GPU implementation to solve transitive and other properties of graphs.

Harish and Narayanan [20] accelerated large graph algorithms on the GPU using CUDA. They computed single source shortest path on a 10 million vertex graph in 1.5 seconds using the NVIDIA 8800GTX GPU. In [21] authors computed shortest paths on Graphic Processing Units. They implemented blocked recursive elimination strategy and their implementation runs more than two orders of magnitude faster on an NVIDIA 8800 GPU.

An OpenCL based parallel implementation of Dijkstra's algorithm is implemented in [22]. It describes implementation of the kernels that compute Dijkstra's algorithm in parallel on single as well as multiple GPUs available on a single workstation. However, they do not take into consideration of multiple GPU available over a network of workstations.

3. ARCHITECTURE OF GPU DEVICE

General Purpose Computing on Graphics Processing Units (GPGPU) is the technique of using a GPU to solve computational problems which are traditionally handled by CPU. Earlier, GPU was designed only for handling computations needed for computer graphics. GPUs are only numeric computing engines, they may perform well for graphical applications but in some cases may not perform well on some tasks on which CPUs are designed to perform well. So, the most applications will use both CPUs and GPUs, executing the sequential parts of program (or application) on CPU and numerically intensive parts on GPUs.

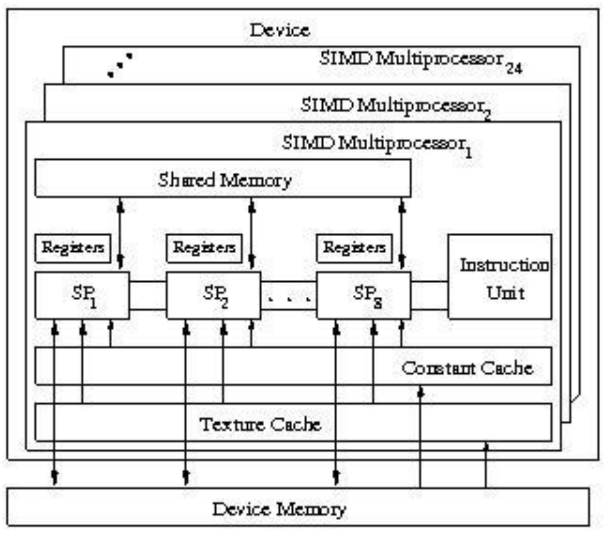

Fig 1: Architecture of NVidia Quadro FX 3800GPU

The structure of typical GPU differs from CPU structure. The main parts of GPU are processing elements (or cores). Fig. 1 shows typical architecture of NVidia CUDA-enabled GPU device. GPU is a two level architecture. At top level, it is made up of array of highly threaded processors termed as Streaming Multiprocessors (SM) and each SM contains eight processing elements termed as Symmetric Processors (SP). For NVidia Quadro FX 3800 GPU Device there are 24 SMs and 192 SPs; however, the number of SMs and SPs can vary from one generation of GPUs to another generation. Each SM has 8,192 registers that are shared among all threads assigned to the SM. The threads on a SM core execute in SIMD (single-instruction, multiple data) fashion, with the instruction unit (IU) broadcasting the current instruction to the eight SPs. Each SP has one arithmetic unit that performs single-precision floating point arithmetic and 32-bit integer operations.

In addition, a GPU device has up to 4GB of GDDR RAM referred to as global memory. These are frame buffer memory which holds video images, and texture information. The NVidia Quadro FX 3800 GPU has several on-chip memories that can exploit data locality and data sharing, e.g. a 64 KB off-chip constant memory and an 8 KB single-ported constant memory cache in each SM. If multiple threads access the same address during the same cycle, the cache broadcasts the address to those threads with the same latency as a register access. In addition to the constant memory cache, each SM has a 16 KB shared (data) memory that is either written and reused or shared among threads. Finally, for read-only data that is shared by threads but not necessarily to be accessed simultaneously, the off-chip texture memory and the on-chip texture caches exploit 2D data locality. For complete study of this topic, refer [25, 26].

4. OPENCL PROGRAMMING MODEL

OpenCL (Open Computing Language) is an open standard targeted for parallel programming of heterogeneous systems. OpenCL provides an interface for handling CPU, GPU, FPGA and various other different types of processors and combination of these processors. OpenCL is suited for interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines.

An OpenCL application consists of two parts (i) kernel – execute on one or more OpenCL devices (ii) host program – execute on host (CPU). The host program creates context and is responsible for managing all the intra and inter-communication between host and OpenCL devices through a set of command queues.

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution, a lot of work-items (i.e. group of threads) are lunched. Work-item is the smallest execution entity, each one executing the same code. An index space, called as NDRange in OpenCL, defines the work-items and how data are mapped to the work-items. A work-item is defined in index space by global ID.

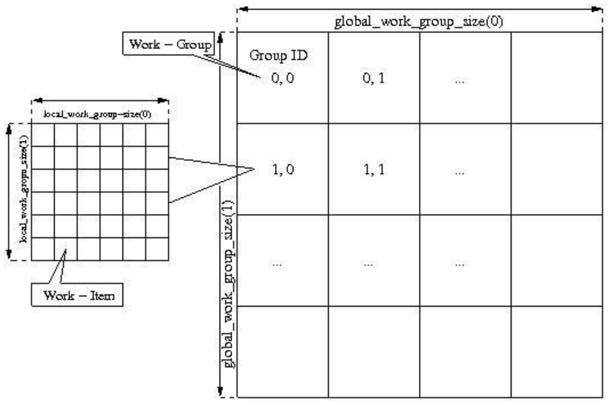

Fig 2: OpenCL Execution Model showing how global IDs, local IDs and work-group indices are related

A group of work-items collectively form blocks of threads called as work- groups. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. Synchronization among work-items in the same work-group is achieved by using barriers. Work-items in different work-groups cannot synchronize with each other.

Fig. 2 shows 2-dimensional NDRange index space showing all work-items, their global IDs and their mapping to their work-groups and local IDs. See [23, 24, 25, 26, 27, 28] for more details on this topic.

5. DIJKSTRA'S SINGLE SOURCE SHORTEST PATH ALGORITHM

Dijkstra's algorithm finds shortest path between pair of vertices in a graph. This algorithm is also called as single source shortest path algorithm as one can find shortest path from given source vertex to all vertices in graph [29, 30] shown as Algorithm 1.

Algorithm 1: Sequential Dijkstra's Algorithm

1. Define a graph as *G = (V, E)* where *V* is a set of vertices and *E* is set of edges.
2. Define a set *S* of vertices whose shortest path from source have already been determined and *(V-S)* be the set of remaining vertices.
3. Define *d* – array of best estimates of shortest path to each vertex and *pi* – array of predecessors for each vertex.
4. Initialize *d* and *pi*.
5. Set *S* to empty
6. While there are still vertices in *(V-S)*
    a. Sort the vertices in *V-S* according to the current best estimates of their distance from source,
    b. Add u, the closest vertex in *V-S*, to *S*,
    c. Relax all the vertices still in *V-S* connected to *u*

The 'Relax' procedure updates the cost of all vertices *v*, connected to vertex *u*, if at all there is a cheapest path to *v* via *u*.

5.1 Single GPU Implementation

We have implemented SSSP using OpenCL. Our implementation is based on the algorithms in [20, 22]. The algorithm consists of two phases. In first phase, we visit marked vertices in Mask array and calculate its cost with its neighbor vertices. The first phase of the SSSP is shown in Algorithm 2.

```
SSSP_kernel_find_neigbour_cost (Ver, Edge, Mask, Weight,
Cost, updCost, ver_cnt, edg_cnt)
{
  Ver [ ]: Vertex array
  Edge [ ]: Edge array
  Mask [ ]: Boolean Mask array of size |Ver|
  Weight [ ]: Weight array of size |Edge|
  Cost [ ]: Cost array
  updCost [ ]: array for storing updated cost
  ver_cnt: No. of vertices
  edg_cnt: No. of edges

  int tid = get_thread_id();
  if (Mask[tid] != 0)
  {
    Mask [tid] = 0;
    int edgeStart = Ver [tid];
    if (tid + 1 < ver_cnt)
    int edgeEnd = Ver[tid + 1]
  }
  else int edgeEnd = edg_cnt;
  for ( e = edgeStart to edgeEnd)
  {
     int nid = Edge[e];
     if (updCost[nid] > Cost[tid] + Weight[e])
         updCost[nid] = Cost[tid] + Weight[e]
  }
}
```

Algorithm 2: Phase 1 of Dijkstra's Algo for single GPU

The second phase of SSSP, checks whether for each vertex cheaper cost have been determined and accordingly updates Cost array. The second phase is shown in Algorithm 3.

```
SSSP_kernel_find_smaller_cost(Ver, Edge, Mask, Weight,
Cost, updCost, ver_cnt)
{
  Ver [ ]: Vertex array
  Edge [ ]: Edge array
  Mask [ ]: Boolean Mask array of size |Ver|
  Weight [ ]: Weight array of size |Edge|
  Cost [ ]: Cost array
  updCost [ ]: array for storing updated cost
  ver_cnt: No. of vertices
  edg_cnt: No. of edges

  int tid = get_thread_id();
  if (Cost[tid] > updCost[tid])
  {
     Cost [tid] = updCost[tid];
     Mask [tid] = 1;
  }
    updCost[tid] = Cost[tid];
}
```

Algorithm 3: Phase 2 of Dijkstra's algo for single GPU

5.2 Multi-GPU Implementation

The general idea is to partition the work into equal blocks. The partitioned blocks can be handled by the available GPUs on the machine. The Dijkstra's algorithm for multiple GPU devices available on a single machine is presented in Algorithm 4.

```
multi_gpu_work_load()
{
  gpu_cnt = get the no. of available gpus

  // create threads for gpus
  pthread_t gid = gpu_cnt

  // divide worload for gpus
  int gpu_work = total_work / gpu_cnt;

  //gpu_work = 0.5; // Allot 50% work to each of the GPUs

  for each of the available GPU device
     • create context //gpu_context[gid]
     • create graph and initialize all the parameters
       associated with the graph

  // Launch all the threads for each of the available GPUs
  pthread_create(gid, dijkstra_algo(gpu_context[gid]))

  // wait for result from all gpus
```

```
   pthread_join(gid, resultst_from_gpu_context[gid]);
}
```

Algorithm 4: Dijkstra Algo. for multiple GPU devices on a single machine

Algorithm for Dijkstra Algo across multiple GPU devices over network of Workstations is presented in Algorithm 5. At the beginning of execution, MPI process initializes and identifies its neighboring nodes. Thereafter, it updates its working area and accordingly defines adjacent borders for neighboring nodes from the corresponding MPI processes. With this information, buffer space can be allocated on various remote GPUs. After this phase, data is transferred to the GPU and from there on the underlying OpenCL application works as usual.

```
Multi_gpu_mpi_opencl()
{
  int rank, size;
  MPI_INIT();

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for each of the rank()
      •  get_node_info();
      •  list_gpu_devices();

  gpu_list[] = get opencl devices available on LAN  /* this
uses MPI calls and SSH protocols */

  /* test gpu_list for equality first. All the GPU should be of
same configurations */

  test_gpu_list(gpu_list);

 // divide worload for gpus
 int gpu_work = total_work / gpu_list.size();

 // gpu_work = 0.5; // allot 50% work to each of the available
gpu device

 // MPI process separation
 MPI_Status s;

if (rank==0) // sender
MPI_Send(&data,1,MPI_INT,source_rank,1,MPI_COMM_W
ORLD);

if (rank!=0){
MPI_Recv(&data,1,MPI_INT,dest_rank,1,MPI_COMM_WO
RLD, &s);
MPI_Send(&data,1,MPI_INT,dest_rank,2,MPI_COMM_WO
RLD);
}

for at each node
      •  allocate_gpu();
      •  create context
      •  create graph and initialize all the parameters
         associated with the graph
      •  Launch kernel_Dijkstra;

if (rank==0)
MPI_Recv(recv_data,recv_data_size,dest_rank,2,MPI_COM
M_WORLD, &s);
Merge_Result(recv_data_from_all_gpus);
MPI_Finalize();
}
```

Algorithm 5: Dijkstra Algo. for multiple GPU devices on Network of Workstations

6. RESULTS AND ANALYSIS

6.1 Experimental setup

In this section, we provide detailed hardware specifications for the CPU and GPU we have used during our experiments.

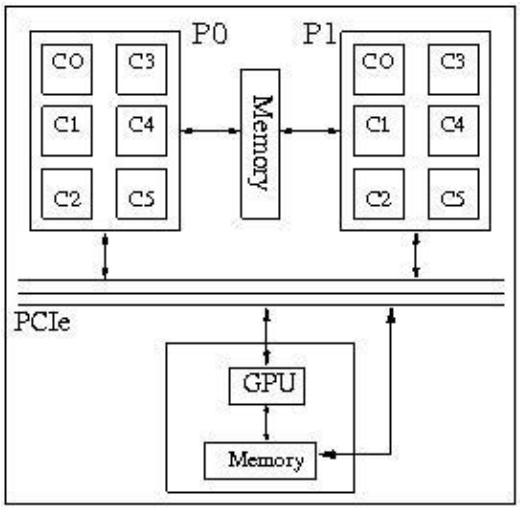

Fig 3: Architecture of our Workstation

Fig. 3 shows architecture of our workstation (host machine). The host machine has Intel Xeon X5650 Twin CPUs running at 2.67GHz. Each core of two processors has 32KB L1 Data cache, 256 KB L2 cache shared between 2 threads of that core. In addition to that, there is 12MB L3 cache shred among all the threads.

The GPU device used in our experiment was NVidia Quadro FX 3800. The device has 192 stream processors (cores) running at 1204 MHz. Table 1 gives detailed GPU specifications including the sizes of global and other memory details.

Table 1. GPU NVidia Quadro FX 3800 Specifications

Type	Number
No. of Cores	192
Max. Compute Units	24
Max. work-item Dim.	3
Max. work-item sizes	512, 512, 64
Max. work-group size	512
Global Memory size	1GB
Max. Constant Buffer Size	64 KB
Max. Constant Arguments	9
Local Memory size	16 KB

The GPU device was connected to CPU through X58 I/O Hub PCI Express. The environment used for experimentation is Open Suse Operating system with kernel version 2.6. OpenCL

1.1 with CUDA 4.2.1 was the primary platform used for executing the OpenCL programs.

For multi-GPU connections, we have two cases. For the first case, two GPUs are connected to each other via SLI Bridge. In second case, two nodes each having identical NVidia Quadro FX 3800 GPU, were connected over the LAN.

6.2 Experimental Results

Table 2 shows execution time for GPU kernel execution on device. For experimentation purpose, we took various numbers of vertices, each with degree 10 and we ran SSSP algorithm for 100 source vertices. All the experiments were repeated 20 times and results reported are averaged over twenty runs.

Performance of SSSP algorithm is shown in Fig. 4. As expected, as the number of GPUs increases, the performance also increases. It can be seen from Figure 4 that dual-GPUs in SLI mode gives good performance if they are available on single workstation.

7. CONCLUSION AND FUTURE SCOPE

This paper shows that Dijkstra's Single Source Shortest Path algorithm can be solved efficiently using multiple GPU devices. It can be seen from Fig. 4 that, average speedup achieved is 3.65 when single GPU is used. As the number of GPUs and size of the problem increases, there is cubic order of performance improvement. In Fig. 4, for 5000000 vertices, speedup achieved for SLI mode is 19.2, as opposed to 14.73 using MPI in non-SLI mode. MPI applications use communication protocols and suffer from latency and bandwidth problems. As a result, there is drop in speedup achieved.

From the experiments we can conclude that a SLI enabled multi-GPU system has more advantage in compute-intensive applications than a non-SLI system.

As a future work, we are planning to investigate various algorithms from allied fields and use different GPU architectures from different vendors for our analysis.

Table 2. Execution Time for SSSP Algorithm (in sec)

No. of Vertices	Sequential	Single GPU (OpenCL)	Dual GPU	
			OpenCL	MPI + OpenCL
100000	8.737	2.721	1.632	2.026
200000	19.819	5.981	2.961	3.129
300000	31.656	9.596	4.022	4.966
500000	55.16	16.799	7.377	8.135
1000000	118.557	35.931	13.325	17.232
5000000	1020.779	196.927	53.176	69.281

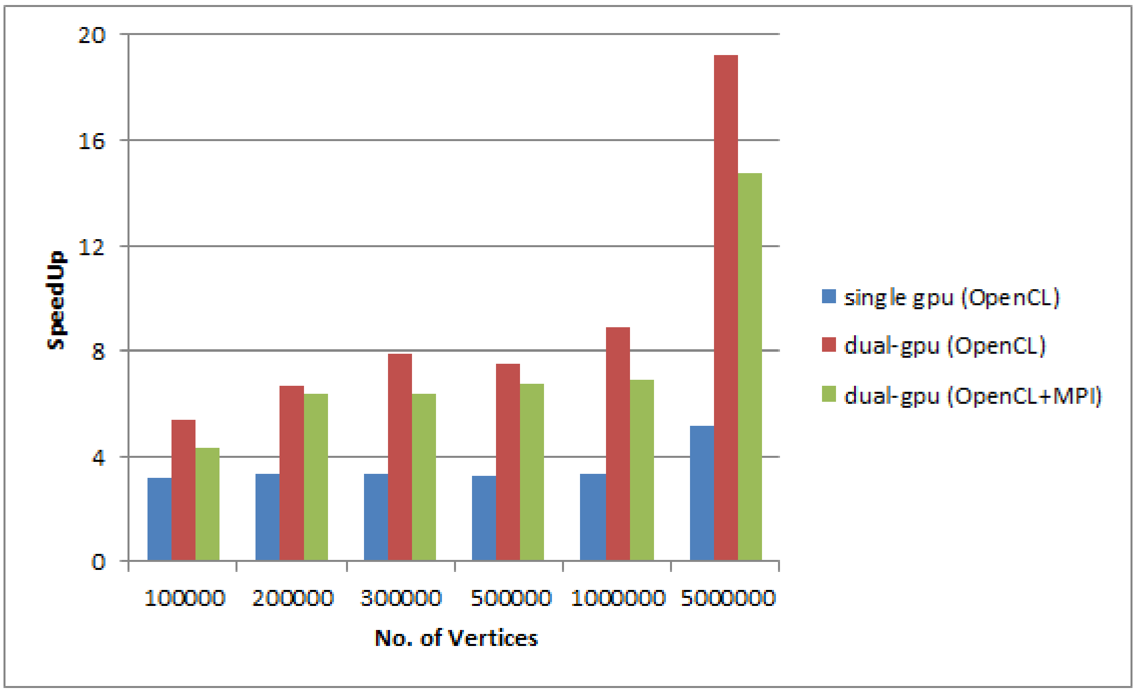

Fig 4: Performance of SSSP Algorithm

8. REFERENCES

[1] General-purpose computations using Graphics hardware, http://www.gpgpu.org/

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fathalian, M. Houston and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," ACM Trans.Graph, Vol.23, No.3, 2004, pp. 777-786

[3] NVIDIA CUDA, http://developer.nvidia.com/cuda/

[4] OpenCL, http://www.khronos.org/registry/cl/

[5] Parallel Boost Graph Library, http://osl.iu.edu/research/pbgl/

[6] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," ICS '12 in Proceedings of the 26th International Conference on Supercomputing, pp. 341 — 352, San Servolo Island, Venice, Italy, June 2012.

[7] OpenMPI: Open Source High Performance Computing, http://www.open-mpi.org/

[8] Jun-Dong Cho, Salil Raje, and Majid Sarrafzadeh. "Fast Approximation Algorithms on Maxcut, K-coloring, and K-color ordering for VLSI Applications," IEEE Transactions on Computers, 47(11):1253–1266, 1998.

[9] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph," ACM Trans. Program. Lang. Syst., 1(1):121–141, 1979.

[10] P. J. Narayanan. "Single Source Shortest Path Problem on Processor Arrays," in Proceedings of the 4th IEEE Symposium on the Frontiers of Massively Parallel Computing, pages 553–556, 1992.

[11] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," Computer Graphics Forum 26, 1 (Mar. 2007), pp. 80–113

[12] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A Blocked All-Pairs Shortest-Paths Algorithm," J. Exp. Algorithmics 8 (2003), 2.2.

[13] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer and Peter Sanders, "A Parallelization of Dijkstra's Shortest Path Algorithm," MFCS 1998, pp. 722-731

[14] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine, "Single-Source Shortest Paths with the Parallel Boost Graph Library," in 9th {DIMACS} Implementation Challenge: The Shortest Path Problem, November 2006.

[15] G. Stefano, A. Petricola and C. Zaroliagis, "On the implementation of parallel shortest path algorithms on a supercomputer", in Proc. of ISPA'06, pp. 406-417, 2006.

[16] Avi Bleiweiss, "GPU Accelerated Pathfinding," Graphics Hardware 2008: pp. 65-74

[17] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk and R. F. Werneck, "PHAST: Hardware-Accelerated Shortest Path Trees," IPDPS 2011, pp. 921-931

[18] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck, "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms," in Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10), 2010, pp. 782–793

[19] Gary J. Katz and Joseph T. Kider Jr, "All-Pairs Shortest-Paths for Large Graphs on the GPU," Graphics Hardware 2008, pp. 47-55

[20] P. Harsh and P.J. Narayan, "Accelerating large graph algorithms on the GPU using CUDA", IEEE High Performance Computing, 2007. vol. 4873 of Lecture Notes in Computer Science, Springer, pp. 197–208

[21] Aydin Buluç, John R. Gilbert and Ceren Budak, "Solving path problems on the GPU," Parallel Computing 36(5-6): pp. 241-253 (2010).

[22] R. Pienaar, B. Fischl, V. Caviness, N. Makris, and P. E. Grant, "Parallelizing Dijkstra's Single Source Shortest-path Graph Algorithm", Chapter 16, OpenCL Programming Guide, Addison-Wesley Publishers, 2011.

[23] The OpenCL specifications www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[24] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa, "Heterogeneous Computing with OpenCL", Morgan Kaufmann Publishers, 2011.

[25] AMD Accelerated Parallel Processing OpenCL Programming Guide, Advanced Micro Devices, Inc. 2012. http://developer.amd.com/appsdk

[26] D. Kirk and W-m. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan-Kaufmann Publishers, 2010.

[27] A. Munshi, B. Gaster, T. Mattson, J. Fung and D. Ginsburg, OpenCL Programming Guide, Addison-Wesley Publishers, 2011.

[28] M. Scarpino, "OpenCL in Action," Manning publications, 2011.

[29] Dijkstra's Algorithm, http://www.cs.auckland.ac.nz/~jmor159/PLDS210/dijkstra.html

[30] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to Algorithms, 3rd Edition, The MIT Press.