International Journal of Multimedia and Ubiquitous Engineering
Vol.9, No.3 (2014), pp.269-280
http://dx.doi.org/10.14257/ijmue.2014.9.3.25

High Performance Computing for Large Graphs of Internet Applications using GPU

Jia Uddin[1], Emmanuel Oyekanlu[2] Cheol-Hong Kim[3] and Jong-Myon Kim[1,*]

[1]School of Electrical Engineering, University of Ulsan, Ulsan, South Korea
[2]Department of Electrical and Computer Engineering, Drexel University, Philadelphia, Pennsylvania, USA
[3]School of Electronics and Computer Engineering, Chonnam National University, South Korea

jia@mail.ulsan.ac.kr, eao48@drexel.edu, chkim22@chonnam.ac.kr, jmkim07@ulsan.ac.kr

Abstract

The high speed CPU based routers currently in use could not handle the massive data required for real-time multimedia communication. Graphics processing units (GPUs) offer an appreciable alternative due to high computation power which results from their parallel execution units. This paper presents the implementation of the Dijkstra's link state IP routing algorithm using GPU. Experimental results show that the proposed GPU-based approach outperforms the same sequential CPU-based implementation in terms of execution time for the same dense graph. In addition, the proposed GPU-based approach reduces about 99% energy consumption over the CPU-based implementation.

Keywords: *GPU, Dijkstra's algorithm, link state, IP routing, energy consumption*

1. Introduction

The world is connected by the internet. Among the various network devices, internet routers play a significant role by serving as the internet backbone due to its strategic function of handling traffic packets between networks. Across the internet landscape, conventional hardware routers initially proved to be a relative success with its high processing speed. However, for many instances those routers become insufficient to ensure the high speed communication such as multimedia internet. The custom hardware struggles to meet the numerous packet specifications and throughput needs [1]. On the other hand, software routers (SR) were deployed with more flexible features and lower costs. However, their performances were also curtailed by limited traffic throughput less than 10Gbps. As a result, SR ultimately fails to support the needs for high data rate (100Tbps) real time applications such as 3G and 4G [2].

Nowadays, graphic processing units (GPUs) are playing a significant role in large scale applications, where massive parallel processing is needed [3, 4]. The advent of the GPU is quite appreciable in graph processing as it works without graph reductions. However, even

[*] Corresponding author.

ISSN: 1975-0080 IJMUE

though GPUs are optimized for graph processing and operations, they are quite challenging to use their highly restrictive programming models. It is also challenging to understand the performance boundaries of the GPU relative to multi-core CPU implementations [5]. In [6], authors examined the performance boundary of a fine-tuned GPU implementation of the genetic algorithm against the performance of an optimized CPU implementation. In [2], the performance of a GPU based linear search framework for packet classification was evaluated. A GPU-based high-performance software router such as PacketShader was proposed for general packet processing [7]. Zhu *et al.* proposed Herms which is an integrated CPU/GPU micro-architecture for quality of service aware high speed routing [8].

A number of researchers often make a comparison between highly parallelized GPU implementation and an un-optimized single core CPU implementation. It often makes the speedup ratio to be skewed in the favor of the GPU. This paper establishes a fair speedup comparison between parallel GPU implementation and a four core CPU implementation (e.g. Intel(R) Core(TM) i5 CPU). In this paper, we proposed a GPU-based implementation of the Dijkstra's algorithm, where some parts of the algorithm are implemented on the host (hosting CPU) while some other parts implemented on the device (GPU). We have also separated the performance index to identify several different interactions between the CPU and the GPU. Our CPU/GPU comparison metrics include execution time, throughput, power and energy consumption between CPU and GPU.

The rest of this paper is organized as follows. Section 2 includes the background study with a detailed overview of compute unified device architecture and the Dijsktra's algorithm. Section 3 describes the proposed parallel implementation of Dijkstra's algorithm using CUDA. Section 4 presents a short overview of different performance metrics, and Section 5 presents experimental results and discussion. Finally, Section 6 concludes the paper.

2. Background Information

2.1. GPU and CUDA Interaction

CUDA is a suite of combined software and hardware architecture. The ease programming features of CUDA enable the transformation of the GPU to the data parallel computing device. In addition, CUDA becomes enjoyable among GPU programmers for the similarity of programming features of the C programming language [9, 10]. Commonly used CUDA compatible GPU devices include NVidia's G100, GeForce 400, and NVidia's Fermi or Tesla series.

If we program the GPU to execute some instructions on a batch of data threads using CUDA programming language, the GPU acts as a co-processor to the CPU. As a co-processor, GPU is capable to execute a high volume of threads in parallel pattern using a CUDA program. The body of the CUDA program to be executed on the GPU is called a kernel. In the memory hierarchy of GPU, a CUDA kernel executes directly on the SPs (streaming processors) or cores. Figure 1 presents detailed of GPU and CUDA interaction along with memory allocation.

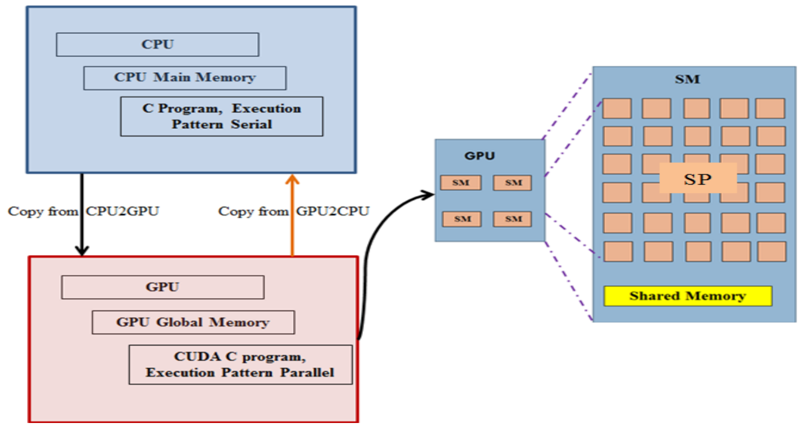

Figure 1. GPU and CUDA interaction with memory allocation

In order to ensure seamless programmability and to avoid the problem of data and instruction association, the device maintains dynamic random access memory (DRAM) separated from that of the host (*i.e.*, CPU). The DRAM enables both the host and the device to be able to dynamically access any location in memory in real time in other to fetch or to do some other needed operation with any chunk of data. Whenever a CUDA kernel is called on the GPU, it can perform its set of instructions on a block of data organized in threads. Each thread in a block has a unique identifier which enables it to maintain its identity separated from other threads in a block. Thread indexing is further maintained on the block by organizing the threads in the form of uni-dimensional or multi-dimensional blocks. Threads of the same blocks communicate with each other, which is separated from the communication paths of other threads in other blocks [11].

2.2. Graph Mapping on CUDA

A graph G could be defined as a set of nodes (V) and edges (E); *i.e.*, $G = (V, E)$. A dense graph will have $|E|$ very close to $|V|^2$ whereas a sparse graph will have $|E|$ quite less than $|V|^2$. For non-weighted dense graphs, the construction of an adjacency matrix which is a 2-D array of Booleans representing the graph topology is often very useful for the purpose tracking graph edges [12]. This property tends to be quite useful for CUDA since the adjacency matrix allows graphs of arbitrary sizes to be created and manipulated in a parallel fashion. Using an adjacency matrix, the vertexes of a graph can be stored in an array while the edges are stored in another adjacency list array. If we denote the array of edges as V and the array of edges as E, then we can utilize Figure 2 to describe the representation of graphs with adjacency lists of arrays and edges in CUDA. Each vertex array entry in V will map to the starting index of its corresponding adjacency list in the edge array E.

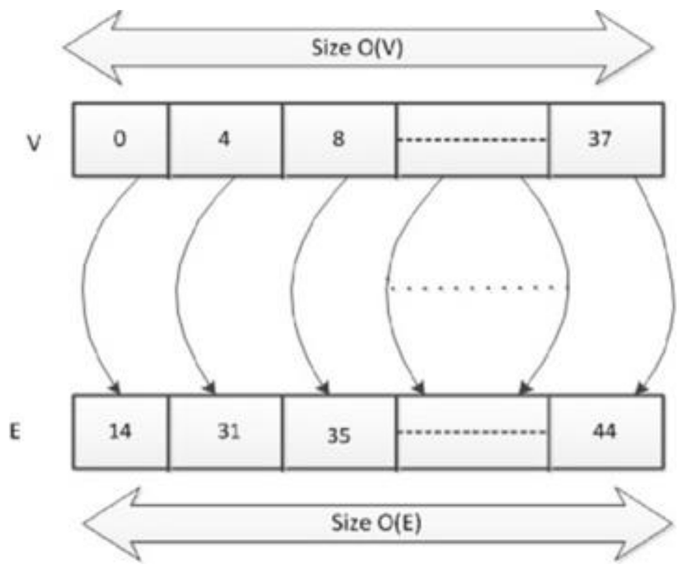

Figure 2. CUDA's graph mapping with a vertex list pointing to an edge list

Thus, a dense graphs containing numerous edges and vertices may be contained in two lists: one is a list of edges and the other is a list of vertices with appropriate pointers of mapping the array of edges to array of vertices in CUDA. The important contribution of this mapping paradigm is that we do not need for graph reductions/sparse matrix computation in order to process the matrices associated with these two graphs. As a result, we do not loose crucial graph specific information. The graphs are processed as is with each element in the array of edges and their associated elements in the array of vertices. They can be processed concurrently in different GPU threads using thread indexing to associate the array list with relevant processing instructions.

2.3. The Dijkstra's Algorithm

The Dijkstra's algorithm is commonly known as the shortest path first (SPF) which works by iterating on a graph to find the paths with the least cost for traversing from a start node to a destination node using a routing database [13]. It is a graph search algorithm and solves the single source shortest path problem for a graph with non-negative edge path cost. Let K be the nodes whose least cost paths are definitely known. Initially, $K = u$, where u is the source node. For each iteration, another node with a known path cost will be added to K until the routing database is completed. We then assign $C(v)$ to be the current cost of paths from source u to node v. At the onset of the iteration, let $C(v) = c(u,v)$ for all nodes v adjacent to the starting node u and let $C(v)$ be infinity for all other nodes v which are not directly connected to u. We can now update $C(v)$ continually until all shorter paths on the entire graph are known. Dijkstra's is one of the best implementation of the general link state algorithms.

For a given source vertex in the graph, the algorithm finds the path with the lowest cost between the source vertex and every other vertex. Three basic steps including insert, extract-min and update-cost are involved in the Dijkstra's algorithm. The insert phase is an initialization phase where numerous positive costs between the vertices are generated. In this stage, the source node is assigned to 0 and for other vertices is assigned to ∞. In the second stage, the neighbour vertex of each vertex is sorted out. In the third stage, the cost for finding out the shortest possible router is updated based on one condition: $C(v)=\min(C(v), C(w)+c(w,v))$ in Algorithm 1.

Algorithm 1: Dijkstra's Algorithm

```
Initialization
K = {u}
for all nodes v
if v is adjacent to u, then
C(v) = c(u,v)
else C(v) = infinity
Loop
ExtractMin: Find w not in K with the smallest C(w)
Add the w to K
Update C(v) for each neighbour v adjacent to w and not in K
Set C(v) = min(C(v), C(w) + c(w,v))
Until K nodes in K
```

3. Implementation of Parallel Dijkstra's Algorithm Using CUDA

The device used for the proposed GPU-based implementation of the Dijkstra's algorithm is the GeForce GT 630, which is a DirectX 11 GPU [14]. It has 96 CUDA cores equipped with a chip-level power enhancements and a GDDR5 memory interface, where GPU clock speed is 810MHz. The processor clock rate is 1620MHz while the graphics clock rate is 810MHz.

CUDA is useful in computing the link states of the entire graph whenever the graph topology changes. The routers are the nodes of a graph and the link weight which should not be negative is the associated costs with each link. Each router has a whole network map and the router recognizes the link states periodically whenever the network topology changes. Each router runs the Dijkstra's algorithm locally where thread parallelization scheme is offered by CUDA [15].

Figure 3 shows a flow diagram of the proposed parallel algorithm, where extract-min and update-cost kernels of the Dijkstra's algorithm are implemented on a GPU. Step 2 to step 6 of the pseudo code of Algorithm 2 is indicative of the whole transactions for the extract-min and the update-cost. The proposed algorithm runs on a heterogeneous system [16] including CPU and GPU.

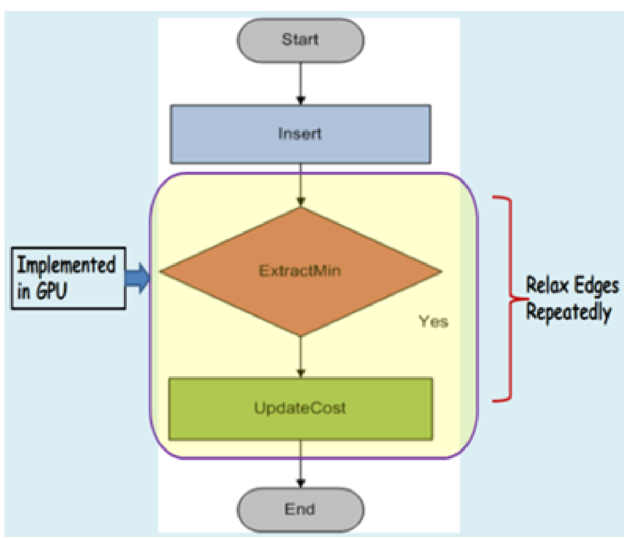

Figure 3. A block diagram of the proposed GPU implementation of the Dijsktra's algorithm

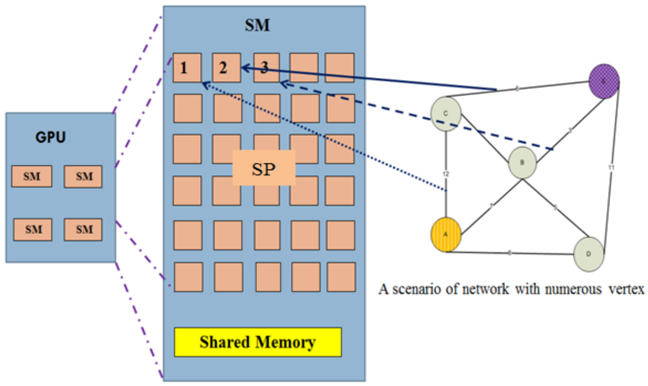

Figure 4. A sample scenario of the proposed parallel algorithm implemented on NVIDIA's GPU with memory allocation

Algorithm 2: CUDA Dijkstra's Pseudo code

```
Step 1:
Generate the cost matrix for the scenarios with different number of routers; in this stage the
source-node is assigned to 0 and for other vertices is assigned to ∞
Step 2:
Allocate the memory in CUDA depending on the number of routers
Step 3:
Transfer the scenarios information (Costmat) from CPU to GPU
Step 4:
(i.)Extract Min
(ii.) Updating once for every edge
v.predecessor = u
Step 5
Transfer the processed data from GPU to CPU
Step 6:
Free CUDA function
```

Algorithm 3 presents a CUDA C main program that includes a kernel function *dij*. In the experiment, we set the kernel function using 2-D blocks and grids in the following ways [17]: blockDim.x=32 and blockDim.y=32, gridDim.x= *n*/32 and gridDim.y= *n*/32, where *n* varies depending on the number of vertex. More explicitly, the Dijkstra's global computation function is shown in Algorithm 4; where, a variable i is used which depends on the two 2-D variables (*x* and *w*). A detailed of 2-D global memory allocation in GPU is presented in Figure 5.

Algorithm 3: Main function of CUDA C code

```
main(){
…//omitted: allocate and initialize memory
//invoke parallel dij kernel with 1024 threads /block
//Number of blocks/grid is varied depending on the
total edges
dim3 dimGrid(n/32,n/32);
dim3 dimBlock(32,32)
dij<<<n/threads_per_block,
threads_per_block>>>(dev_cost, dev_dist, v);
//omitted: transfer results from GPU to CPU }
```

Algorithm 4: Global function

```
_global__ void dij(int *cost, int *dist, int v){
//omitted initialization
int x= blockIdx.x*blockDim.x+threadIdx.x;
int w= blockIdx.y*blockDim.y+threadIdx.y;
int i=x+w*n;
if(i<n&&w<n){
//omitted some part of program
if((dist[u]+cost[u*(n-1)+i]<dist[i]) &&!flag[i])
dist[i]=dist[u]+cost[u*299*(n-1)+i];
}}
```


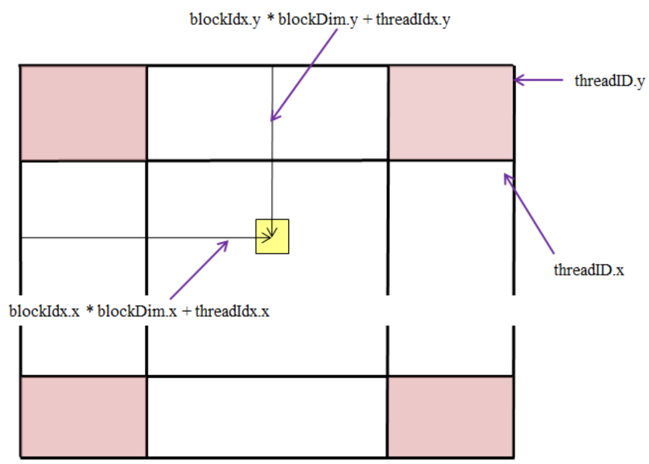

Figure 5. 2-D global memory allocation in GPU

4. Performance Metrics

In order to obtain a reasonable performance comparison between the Intel(R) Core(TM) i5 CPU and the NVIDIA GeForce GT 630, we segment the performance index into different parameters in order to be able to completely examine the performance of the CUDA implementation of the Dijkstra's algorithm. In this paper, we consider the following performance metrics including throughput, execution time, speedup, and power and energy consumption.

4.1. Throughput

GPU throughput is a measure of how much work gets done every second, which is measured in bit/sec. It can be affected by bandwidth and latency.

4.2. Speedup

GPU speedup over CPU is the rate of the speed increment of the parallel instruction execution of the GPU over an equivalent sequential execution on the CPU. It can be computed as $Speedup = Tc / Tg$; where, T_c is the total time taken by the sequential CPU instruction execution while T_g is the total time taken for the GPU-based implementation.

4.3. Power and Energy Consumption

Power consumption is measured in unit of watt (joules/sec). In this paper, we measure the consumed power by subtracting the power by CPU/GPU during the kernel execution and the power consumed in the ideal state of CPU/GPU, such as power consumption $(P_c) = P_e - P_i$, where P_e is the consumed power by a device (CPU/GPU) during the kernel execution and P_i is the consumed power by a device in ideal state. Energy is power integrated over time such as $Energy = Power \times Time$.

5. Experimental Results and Analysis

To evaluate of the performance of the proposed GPU-based approach, we use a standard NVIDIA tool called Visual Profiler (nvprof) [18] which enables the collection of a timeline of any applications for CPU and GPU activities including kernel execution, memory transfers and CUDA API calls. The profiling results are displayed in the console after the application is completed. All values are reliable as it collects the average values by running a program 21 times. In this section, the number of available nodes for processing in numerous network scenarios is analyzed.

Table 1. Execution time of the GPU-based implementation for varying the number of vertices

	Execution time(sec)				Speedup	
V	T_e^c	T_e^g	T_m	T_{em}^g	S^g	S_m^g
300	0.129	0.000946	0.0005021	0.00144854	136x	89x
500	0.591	0.001535	0.0009793	0.00251468	385x	235x
1000	4.574	0.007717	0.0031237	0.01084026	593x	422x
1500	12.98	0.019922	0.0061010	0.02602259	652x	499x
2000	35.74	0.059353	0.0101254	0.06947846	602x	514x

V: Number of Vertex. T_e^c: CPU execution time Algorithm 1 in sec. T_e^g: GPU execution time of Algorithm 2 in sec without considering memory transfer. T_m: Memory transfer time in sec = CPU2GPU+GPU2CPU. T_{em}^g: Sum of execution time and memory transfer time for Algorithm 2 in GPU in sec. S^g: Speed up gained by Algorithm 2 without memory transfer time. S_m^g: Speed up gained by Algorithm 2 with memory transfer time.

5.1. Performance of the GPU-based Approach

In this section, the performance of the proposed GPU-based approach is evaluated in terms of execution time. Table 1 shows a performance summary of the GPU-based approach. In addition, the performance of the GPU-based approach is compared with that of the equivalent CPU-based implementation, which includes five cores.

Execution time of the CPU-based implementation exponentially increases when the number of nodes increase, while execution time of the GPU-based approach shows almost steady due to the parallel execution pattern of the GPU. In other words, the extract-min and update-cost kernels of each vertex are calculated in different blocks of GPU. Figure 10In addition, and. Since numerous blocks and similar threads execute concurrently on streaming multiprocessors, the execution time to sort out the cost matrix in extract-min and updating cost is reduced significantly. On the other hand, owing to the sequential execution pattern CPU latency continues to rise. The CUDA implementation of the algorithm is highly beneficial when significant amount of nodes are processed because this is what the GPU is optimized for. As shown in Figure 10, the speedup of the entire CUDA implemented Dijkstra's algorithm exhibits an increasing pattern when the graph becomes denser. It happens due to the higher execution time of CPU for the scenarios with large amount of nodes.

In the heterogeneous system for CPU and GPU, memory transfer time becomes performance bottleneck of GPU computing. This memory transfer time plays a significant role where massive data transfer is demanded. In the proposed GPU-based implementation, the memory transfer time of CPU to GPU (CPU2GPU) and vice versa is very small for small amount of nodes as depicted in Figure 8. However, memory transfer time gradually increases with the number of nodes, which significantly hampers the overall system performance. It is noted that the transfer time of GPU to CPU is slightly greater than that of reverse direction due to the lower clock frequency of GPU.

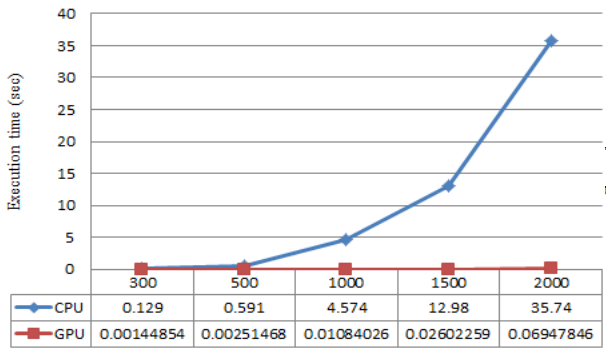

Figure 6. Execution time of GPU and CPU

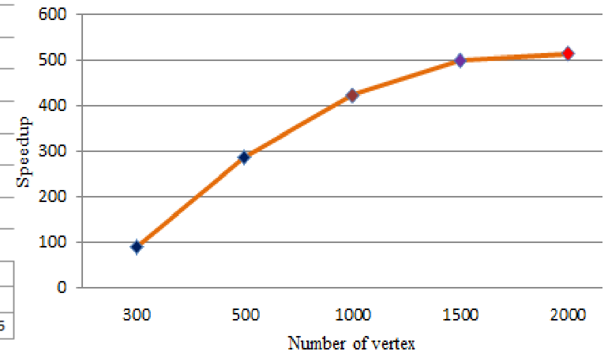

Figure 7. Speedup of GPU over CPU

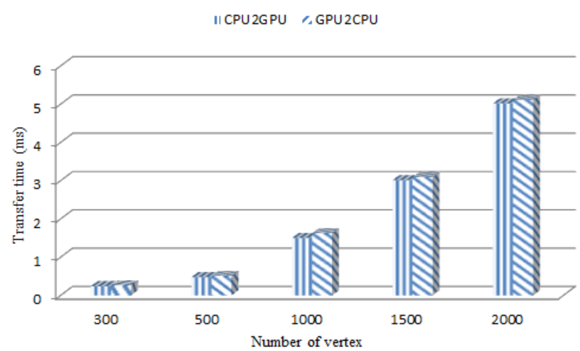

Figure 8. Transfer time of GPU and CPU

5.2. Average Power and Energy Consumption of GPU over CPU

The power consumption of the GPU-based and CPU-based implementations is measured by using the INSPECTOR tool (INSPECTOR II SE).Detailed statistics of power and energy consumed by the GPU and CPU is presented in Table 2.

Table 2. Power and energy consumption for varying the number of vertices

	Power Consumption (Watt)				Energy Consumption(Watt-Sec)	
V	P_c	P_g	T_e^c	T_e^g	E_c	E_g
300	33.13	47.43	0.129	0.0014485	4.27377	0.068704
500	35.63	48.53	0.591	0.0025147	21.0573	0.122037
1000	36.43	51.13	4.574	0.0108403	166.631	0.554262
1500	41.33	53.53	12.98	0.0260226	536.463	1.392989
2000	47.94	59.03	35.74	0.0694785	1713.38	4.101313

V: Number of vertex. P_c: CPU power consumption in watt. P_g: GPU power consumption in watt. T_e^c: Execution time of CPU in sec. T_e^g: Execution time of GPU in sec. E_c: Consumed Energy by CPU (watt/sec). E_g: Consumed Energy by GPU (watt/sec).

Although the consumed power of both GPU- and CPU-based approaches slowly increases with the number of vertices, the average power consumption of the GPU-based approach is comparatively larger than that of the CPU-based implementation, as presented in Figure 9, due to the use of CPU and GPU together to run the CUDA kernels. However, for energy consumption of the GPU-based approach requires significantly less energy compared to the

CPU-based implementation because the energy consumption is proportional to the execution time and power consumption. Therefore, although the proposed GPU-based implementation consumes more power than the CPU-based implementation, it consumes significantly less energy than the CPU-based implementation by highly reducing the execution time as shown in Figure 10. The average energy consumption is measured by the following equation: $(E_c - E_g)/E_c x100$, where E_c is the consumed energy by CPU and E_g is the consumed energy by GPU. The proposed GPU-based approach reduces above 98% for the average energy consumption of the sequential implementation.

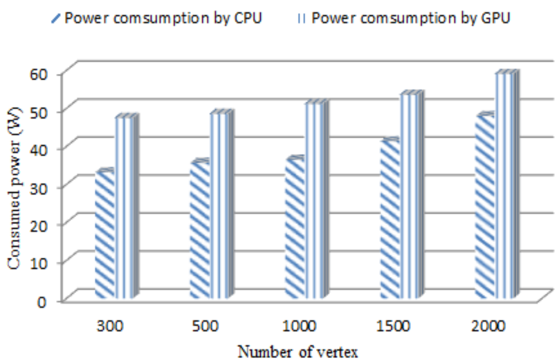

Figure 9. Average power consumption by CPU and GPU

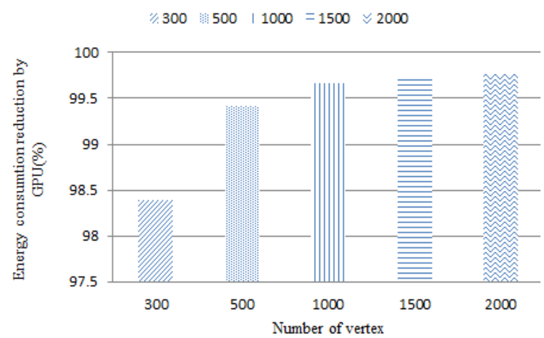

Figure 10. Percentage of average energy consumption reduction by GPU over CPU

6. Conclusions

In this paper, we implemented the link state Dijkstra's algorithm using GPU. In addition, we compared the performance of the proposed GPU-based implementation with the equivalent sequential algorithm using CPU in terms of execution time and energy consumption. The proposed GPU-based approach outperforms the CPU-based program for all the number of vertices in terms of execution time. In addition, the proposed GPU-based implementation reduces about 99% energy consumption of the CPU-based implementation. In the future, we will explore other performance metrics, such as power and energy efficiency, and kernel optimization of the proposed algorithm.

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. NRF-2013R1A2A2A05004566) and by the Leading Industry Development for Economic Region (LeadER) grant funded the MOTIE (Ministry of Trade, Industry and Energy), Korea in 2013. (No. R0001220).

References

[1] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng and S. Zhang, "IP Routing with Graphic Processors", Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), **(2010)** March 8-12; pp. 93-98, Dresden, Germany.

[2] K. Kang and Y. S. Deng, “Scalable packet classification via GPU meta-programming”, Proceeding of the Design, Automation & Test in Europe Conference & Exhibition (DATE), **(2011)** March 14-18; pp. 1-4, Grenoble, France.
[3] T. Y. Liang and Y. W. Chang, “GridCuda: A Grid-enabled CUDA Programming Toolkit”, Proceeding of the IEEE Workshop of International Conference in Advanced Information Networking Application, **(2011)** March 22-25; pp. 141-146, Biopolis, Singapore.
[4] J. Zhao, X. Zhang, X. Wang, Y. Deng and X. Fu, “Exploiting graphics processors for high-performance IP lookup in software routers”, Proceeding of the IEEE INFOCOM, **(2011)** April 10-15; pp. 301-305, Shanghai, China.
[5] S. Hong, T. Oguntebi and K. Olukotun, “Efficient Parallel Graph Exploration on Multi-Core CPU and GPU”, Proceeding of the International Conference on Parallel Architectures and Compilation Technologies (PACT), **(2011)** October 10-14; pp. 78-88, Texas, USA.
[6] J. Jaros and P. Pospichal, “A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark”, Applications of Evolutionary Computation, LNCS, Springer-Verlag Berlin Heidelberg, vol. 7248, **(2012),** pp. 426-435.
[7] S. Han, K. Jang, K. Park and S. Moon, “PacketShader: A GPU-Accelerated Software Router”, Proceeding of the ACM SIGCOMM conference, **(2010)** August 30-September 3; pp. 195-206, New Delhi, India.
[8] Y. Zhu, Y. Deng and Y. Chen, “Hermes: An Integrated CPU/GPU Microarchitecture for IP Routing”, Proceeding of the 48th Design Automation Conference, **(2011)** June 5-9; New York, NY.
[9] L. Shi, H. Chen and J. Sun, “vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines”, IEEE Transactions on Computers, vol. 61, no. 6, **(2012),** pp. 1-11.
[10] D. Luebke, “CUDA: Scalable Parallel Programming for High Performance Scientific Computing”, Proceeding of the 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, **(2008)** May 14-17; pp. 836-838, Paris, France.
[11] P. Harish and P. Narayanan, “Accelerating Large Graph Algorithms on the GPU Using CUDA”, Proceeding of the 14th International Conference on High Performance Computing, LNCS, vol. 4873, **(2007),** pp. 197-208.
[12] J. A. Bleiweiss, “GPU Accelerated Pathfinding”, Proceeding of the 23rd ACM SIGGRAPH/ EUROGRAPHICS symposium on Graphics hardware, **(2008)** June 20-21; pp. 65-74, Switzerland.
[13] V. Paxson, “Shortest-Path Routing: Link-State and Distance Vector”, Faculty of Electrical Engineering, University of California, Berkeley, (**2007**).
[14] GEFORCE Hardware Specification, http://www.geforce.com/hardware/desktop-gpus/geforce-gt-630/specifications.
[15] T. V. Luong, N. Melab and E. Talbi, “GPU Computing for Parallel Local Search Metaheuristic Algorithm,” IEEE Transactions on Computers, vol. 62, no. 1, **(2013),** pp. 173-185.
[16] H. Choi, D. Son, S. Kang, J. Kim, H. Lee and C. Kim, “An Efficient Scheduling Scheme Using Estimated Execution Time for Heterogeneous Computing Systems,” Journal of Supercomputing, vol. 65, no. 2, (**2013**), pp. 1-17.
[17] NVIDIA, CUDA Programming Guide, CUDA Driver, Toolkit, and SDK Code Samples, http://www.nvidia.com/object/cudaget.htm.
[18] CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual-profiler.

Authors

Jia Uddin

He received the B.Sc. degree in Computer & Communication Engineering from International Islamic University Chittagong (IIUC), Bangladesh, in 2005, the M.Sc. degree in Electrical Engineering emphasis on Telecommunications from Blekinge Institute of Technology (BTH), Sweden, in 2010. Currently, he is pursuing Ph.D. in Computer Engineering in University of Ulsan (UoU), South Korea. He is an Assistant Professor (now in study leave) in Faculty of Science & Engineering at IIUC, Bangladesh. His research interests include High speed Computing, Wireless Networks and Multimedia Communications. He is a member of the IEB and the IACSIT.

Emmanuel Oyekanlu

He received the B.Tech. Degree in Computer Engineering from Ladoke Akintola University of Technology, Nigeria in 2004. In 2011, he received three Master of Science degrees in Telecommunication, Signal Processing and Electrical Engineering; all from the Blekinge Institute of Technology, Sweden

Later, in 2011, he joined the Department of Electrical and Computer Engineering at Drexel University, Philadelphia, Pennsylvania, USA for his PhD study and he currently work on the application of power-line communication and adaptive signal processing to the problem of intelligent control of the electrical power micro-grid system for enhanced integration of renewable energy sources.

Cheol-Hong Kim

He received a B.S., an M.S., and a Ph.D. in Computer Engineering from Seoul National University, Seoul, Korea, in 1998, 2000, and 2006, respectively. He is currently an associate professor of Electronics and Computer Engineering at Chonnam National University, Gwangju, Korea. His current research interests include embedded systems, mobile system, system on chip design, and parallel processing.

Jong-Myon Kim

He received the BS degree in electrical engineering from the Myongji University, Yongin, Korea, in 1995, the MS degree in electrical and computer engineering from University of Florida, Gainesville, in 2000, and the PhD degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, in 2005. He is an associate professor of Electrical Engineering at University of Ulsan, Korea. His research interests include multimedia processing, multimedia specific processor architecture, parallel processing, and embedded system. He is a member of the IEEE and the IEEE Computer Society.

