Proceedings of the 13th International Conference
on Computational and Mathematical Methods
in Science and Engineering, CMMSE 2013
24–27 June, 2013.

A Tuned, Concurrent-Kernel Approach to Speed Up the APSP Problem

Hector Ortega-Arranz[1], Yuri Torres[1], Diego R. Llanos[1] and Arturo Gonzalez-Escribano[1]

[1] *Dpto. Informática, Universidad de Valladolid*

emails: hector@infor.uva.es, yuri.torres@infor.uva.es, diego@infor.uva.es, arturo@infor.uva.es

Abstract

The All-Pair Shortest-Path (APSP) problem is a well-known problem in graph theory whose objective is to find the shortest paths between any pair of nodes. Computing the distances from one source node to the rest and repeating this process for every node of the graph is an adequate solution for sparse graphs. During the last years the application of GPU devices have increased to accelerate this kind of problems. While the correctness of an NVIDIA CUDA implementation of this algorithm is easy to achieve, exploiting the GPU capabilities to obtain a good performance is a task for CUDA experienced programmers. A typical code tuning strategy is the selection of an appropriate threadBlocks size. Besides this, the concurrent deployment of several kernels that computes distances from different sources, also accelerates the execution times. In this paper we show that an adequate combination of both strategies represents a 11.5 % performance improvement between different, recommended CUDA configurations for the most costly kernel of the APSP problem.

Key words: APSP, Concurrent-kernel, Dijkstra, GPU, SSSP, ThreadBlock size.

1 Introduction

Many problems that arise in real-world networks imply the computation of shortest paths and their distances from any source to any destination point. Some examples include traffic simulations [1], databases [2], Internet route planners [3], sensor network [4] or even the computation of graph features as betweenness centrality [5]. Algorithms to solve shortest-path problems are computationally costly, so, in many cases, commercial products implement

©CMMSE ISBN: 978-84-616-2723-3

heuristic approaches to generate approximate solutions instead. Although heuristics are usually faster and do not need much amount of data storage or precomputation, they do not guarantee the optimal path.

The All-Pair Shortest-Path (APSP) problem is a well-known problem in graph theory whose objective is to find the shortest paths between any pair of nodes. Given a graph G= (V, E) and a function $w(e) : e \in E$ that associates a weight to the edges of the graph, it consists in computing the shortest paths for all pair of nodes $(u, v) : u, v \in V$. The APSP problem is a generalization of the classical problem of optimization, the Single-Source Shortest-Path (SSSP), that consists in computing the shortest paths from just one source node s to every node $v \in V$. If the weights of the graph range only in positive values, $w(e) \geq 0 : e \in E$, we are facing the so-called Non-negative Single-source Shortest-Path (NSSP) problem.

There are two ways to solve the APSP problem. The first solution is to execute $|V|$ times a NSSP algorithm selecting a different node as source in each iteration. The classical algorithm that solves the NSSP problem is due to Dijkstra [6]. The second solution is to execute an algorithm that globally solves the APSP problem using dynamic programming, as the Floyd-Warshall algorithm [7, 8]. The former approach is used for sparse graphs whereas the latter is more efficient for dense graphs.

In this paper we face the APSP problem for sparse graphs using a parallelized version of Dijkstra's algorithm for GPUs [9] combined with CUDA optimization methods. These methods include the selection of a good criteria for the threadBlock size [10, 11] and the deployment of concurrent kernels in the same application context on a single GPU device to better exploit the underlying hardware resources [12]. The naïve Dijkstra's algorithm is a greedy algorithm whose efficiency is based in the ordering of previously computed results. This feature makes its parallelization a difficult task. However, there are certain situations where parts of this ordering can be permuted without leading to wrong results neither performance losses. Some of these enhancements were posed by the authors in [13] allowing to take more profit from parallel computation systems and devices.

An emerging way of parallel computation includes the use of hardware accelerators, such as graphic processing units (GPUs). Their powerful capability have triggered their massive use to speed up high-level parallel computations. High-level languages for parallel-data computation, such as CUDA [14] and OpenCL [15], ease the general purpose programming for these devices. The application of GPU computing to accelerate problems related with shortest-path problems have increased during the last years. Some GPU-implemented solutions to the NSSP problem have been previously developed by [9, 16, 17] using some modifications of Dijkstra's algorithm. The former algorithm is the parallel implementation that we have used for our experiments.

In this paper we present GPU solutions to the APSP problem that apply the different configurations recommended by CUDA [14] and the ones recommended in [10, 11] in order

©CMMSE ISBN: 978-84-616-2723-3

Table 1: Summary of Fermi CUDA architecture parameters.

Parameter	**Fermi GF110**
Number of SPs (per-SM)	32
Max. number of blocks (per-SM)	8
Max. number of threads (per-SM)	1 536
Max. number of threads (per-block)	1 024
Max. concurrent kernel supported	16
Max. Occupancy block sizes	192, 256, 384, 512, 768

to obtain a good performance for the most costly kernel used in the GPU implementation. Our experimental results show that the correct choice of some tuning parameter values of the GPU, as the threadBlock size and the number concurrent kernels, represents a performance difference up to a 11.5 percent from the worst recommended configuration to the best one.

The rest of this paper is organized as follows. Section 2 introduces some details of the Fermi CUDA architecture. Section 3 introduces some basic concepts and notations related to graph theory, and briefly describes the sequential Dijkstra's algorithm. Section 4 explains the parallel version used and its implementation for GPU devices. Section 5 describes the optimization techniques used to speed up the GPU solution. Section 6 poses the experimental methodology and used platform, and the input sets considered. Section 7 discusses the results obtained. Finally, Sect. 8 summarizes the conclusions we have obtained and future work.

2 CUDA Overview

Graphics processing units started as image processing devices. Over the years, the GPUs have increased in performance, architectural complexity, and programmability. Currently, these devices are widely used for general purpose computing (GPGPU) [18] due to the performance improvements achieved on multiple kind of parallel applications.

CUDA (Compute Unified Device Architecture) [14] is the parallel computing architecture developed by Nvidia Company for general purpose applications. CUDA simplifies the GPGPU programming by means of high level API and a reduced set of instructions.

The second generation of CUDA architecture is Fermi [12], released on early 2010. Table 1 summarizes Fermi's main characteristics. This architecture generation has increased the number of SPs (Streaming Processors), and the maximum number of threads per SM (Streaming Multiprocessor). Fermi introduced support for concurrent kernel execution, where different kernels of the same application context can be executed on a single GPU at the same time. This concurrent execution allows to better exploit the underlying hardware resources. If the number of launched kernels overpasses the upper limit of concurrent kernels supported by the architecture, then these kernels are stored in a concurrent kernel queue.

 ISBN: 978-84-616-2723-3

3 Algorithmic Overview

3.1 Graph Theory Notation

We will briefly present some graph theory concepts and notations related to the shortest-path problem. A graph $G = (V, E)$ is composed by a set of vertices V, also called nodes, and a set of edges E, also called arcs. Every vertex v is usually depicted as a point in the graph. Every edge e is usually depicted as a line that connects two and only two vertices. An edge is a tuple (u, v) that represents a link between vertices u and v. The number of edges connected to a vertex v is called the *degree* of v. In an *undirected graph* all edges can be traversed in both directions, whereas an edge (u, v) of a *directed graph* only can be traversed from u to v. There is a weight function $w(u, v)$ associated to each edge, that represents the cost of traversing the edge.

A *path* $P = \langle s, ..., u, ..., v, ..., t \rangle$ is a sequence of vertices connected by edges, from a source vertex s to a target one t. The *weight* of a path, $w(P)$, is the sum of all the weights associated to the edges involved in the path. The *shortest path* between two vertices s and t is the path with the minimum weight among all possible paths between s and t. Finally, the minimum distance between s and t, $d(s, t)$ or simply $d(t)$, is the weight of the shortest path between them. We denote $\delta(s, t)$, or simply $\delta(t)$, to a temporal tentative distance between s and t during the computation of $d(t)$.

3.2 Dijkstra's algorithm

The basic solution for the NSSP is Dijkstra's algorithm [6]. This algorithm constructs minimal paths from a source node s to the remaining nodes, exploring adjacent nodes following a proximity criterion. The exploring process is known as *edge relaxation*. When an edge (u, v) is relaxed from a node u, it is said that node v has been *reached*. Therefore, there is a path from source through u to reach v with a tentative shortest distance. Node v will be considered *settled* when the algorithm has found the shortest path from source node s to v. The algorithm finishes when all nodes are settled.

The algorithm uses an array, D, that stores all tentative distances found from source node s to the rest of nodes. At the beginning of the algorithm, every node is unreached and no distances are known, so $D[i] = \infty$ for all nodes i, except current source node $D[s] = 0$. Note that both reached and unreached nodes are considered unsettled nodes.

The algorithm proceeds as follows:

1. (Initialization) The algorithm starts on the source node s, initializing distance array $D[i] = \infty$ for all nodes i and $D[s] = 0$. Node s is considered as the *frontier node* f ($f \leftarrow s$) and it is settled.
2. (Edge relaxation) For every node v adjacent to f that has not been settled, a new distance from source is found using the path through f, with value $D[f] + w(f, v)$. If this distance is lower than previous value $D[v]$, then $D[v] \leftarrow D[f] + w(f, v)$.

©CMMSE ISBN: 978-84-616-2723-3

3. (Settlement) The node u with the lowest value in D is taken as the new frontier node ($f \leftarrow u$). After this, current frontier node f is now considered as settled.
4. (Termination criterion) If all nodes have been settled the algorithm finishes. Otherwise, the algorithm proceeds to step 2.

In order to recover the path, every reached node stores its predecessor, so at the end of the query phase the algorithm just runs back from target through stored predecessors till the source node is reached. The *shortest path tree* of a graph from source node s is the composition of every shortest path from s to the remaining nodes.

4 GPU Parallel Version of Dijkstra's algorithm

In order to parallelize the Dijkstra algorithm, it is needed to identify which nodes can be settled and used as frontier nodes at the same time. Crauser *et al.* in [13] introduces an enhancement that tries to augment the frontier set with nodes which tentative distance is bigger than the minimum computed in the iteration i but lower than a threshold Δ_i. The algorithm computes in each iteration i, for each node of the unsettled set, $u \in U_i$, the sum of: (1) its tentative distance, $\delta(u)$, and (2) the minimum cost of its outgoing edges, $W(u) = \min\{w(u,z) : (u,z) \in E\}$. Afterwards, it calculates the minimum of these computed values, $\Delta_i = \min\{\delta(u) + W(u) : u \in U_i\}$. Finally, those nodes whose tentative distance $\delta(u)$ is lower or equal than this minimum value Δ_i can be settled, becoming the frontier set.

The four Dijkstra's algorithm steps described in Sect. 3.2 can be easily transformed into a GPU general algorithm (see Alg. 1). It is composed of three kernels that executes the internal operations of the Dijkstra vertex outer loop.

The *relax kernel* (Alg. 2 (left), invoked in line 3 of Alg. 1) decreases the tentative distances for the remaining unsettled nodes of the current iteration i through the outgoing edges of the frontier nodes $f \in F_i$. A GPU thread is associated for each node in the graph. Those threads assigned to frontier nodes, $f \in F_i$, traverse their outgoing edges, relaxing the distances of their unsettled adjacent nodes.

The *minimum kernel* (invoked in line 4 of Alg. 1) computes the minimum tentative distance of the nodes that belongs to the U_i set. To do so, the advanced *reduce3* method of the CUDA SDK has been modified to accomplish this task. Our *minimum kernel* is adapted in order to: (1) add the corresponding $W(v)$ value to $\delta(v)$, and (2) compare its new assigned values to obtain the minimum one. The resulting value of this kernel is Δ_i.

The *update kernel* (Alg. 2 (right), invoked in line 5 of Alg. 1) settles those nodes from U_i whose tentative distances are lower or equal to Δ_i. This task is carried out extracting them from the following-iteration unsettled set, U_{i+1}, and putting them to the following-iteration frontier set F_{i+1}. Each single GPU thread checks, for its corresponding node v, whether $U(v) \wedge \delta(v) \leq \Delta_i$. If so, the thread assigns v to F_{i+1} and deletes v from U_{i+1}.

©CMMSE ISBN: 978-84-616-2723-3

Algorithm 1 GPU implementation of Dijkstra's algorithm. Kernels are delimited by <<< ... >>>.

```
1: <<<initialize>>> (U, F, δ);          //Initialization
2: while (Δ ≠ ∞) do
3:     <<<relax>>> (U, F, δ);           //Edge relaxation
4:     Δ =<<<minimum>>> (U, δ);         //Settlement step_1
5:     <<<update>>> (U, F, δ, Δ);       //Settlement step_2
6: end while
```

Algorithm 2 Pseudo-code of a CUDA thread in *relax kernel* (left) and *update kernel* (right).

relax_kernel(U, F, δ)

```
1: tid = thread.Id;
2: if (F[tid] == TRUE) then
3:     for all j successor of tid do
4:         if (U[j] == TRUE) then
5:             BEGIN ATOMIC REGION
6:                 δ[j] = min{δ[j], δ[tid] + w(tid, j)};
7:             END ATOMIC REGION
8:         end if
9:     end for
10: end if
```

update_kernel(U, F, δ, Δ)

```
1: tid = thread.Id;
2: F[tid]= FALSE;
3: if (U[tid]==TRUE and δ[tid] <= Δ) then
4:     U[tid]= FALSE;
5:     F[tid]= TRUE;
6: end if
```

5 CUDA optimizations to the *relax kernel*

This section describes how the CUDA optimization techniques and the guidelines described in [10, 11] can improve the performance of the GPU solution for the APSP problem. From the kernels used in this algorithm, we are going to focus on optimizing the relax kernel (See Alg. 2) because it is the most costly kernel due to its scattered memory access pattern.

5.1 ThreadBlock size

The maximization of SM occupancy tries to hide the global memory latencies. In order to achieve a high ratio of occupancy, CUDA recommends to select a proper threadBlock size. However as we will see, this strategy not always leads to the best performance.

The authors in [10, 11] have tested a wide collection of kernels with different features using several optimization techniques. Their results showed that, for kernels with a high ratio between low-coalesced global memory accesses and the number of instructions per thread, a good performance can be obtained using threadBlock sizes that represent a medium ratio of SM occupancy.

Due to the big amount of simultaneous memory requests, whose latencies cannot be hidden, this type of kernels does not need to maximize the occupancy to obtain a good performance. Moreover, the reduction of the number of threads per block also alleviates the global memory bandwidth bottleneck, leading to better execution times. Thus, this reduction of the memory bandwidth bottleneck and bank conflicts compensates the inactivity of some idle SPs per SM.

©CMMSE ISBN: 978-84-616-2723-3

The *relax kernel* of the GPU implementation shown in Sect. 4 presents similar features as the kind of kernels described above. This fact suggest us that the use of block sizes with a medium occupancy ratio would obtain better performance behaviour than the block sizes recommended by CUDA, as can be seen in the following section.

5.2 Concurrent kernels

Since the second generation of NVIDIA architectures, CUDA supports concurrent kernel execution. With this feature different kernels can be executed concurrently, allowing better utilization of GPU resources. The maximum limit of concurrent kernels that the Fermi architecture present in GF 110 series supports up to 16 kernels.

Moreover, the advantages of concurrent kernel execution are automatically exploited by the CUDA kernel dispatcher, providing a high efficiency without an extensive user intervention. The authors in [12] show that the best performance using concurrent kernels is achieved with small kernels. The maximum number of concurrent kernels that can be allocated efficiently in the GPU depends on the use of the hardware resources made by each one. The less resources used by each kernel, the more kernels can be efficiently computed concurrently.

Although the *relax kernel* was designed to take profit of the available resources, we believe that the use of concurrent-kernels technique would provide a better exploitation of the L1 and L2 data-cache and the threadBlock/warp dispatchers, as can be seen in the following section.

6 Experimental setup

In this section we describe the methodology used for our experiments as well as the input set problem used and the different configurations evaluated.

6.1 Methodology

We have compared the GPU solution proposed in [9] with different configurations of the CUDA optimization techniques for the *relax kernel*, that are the threadBlock size and the number of concurrent kernels. Both techniques are closely related to the use of the hardware resources, and therefore, we have considered to test them together to efficiently squeeze the GPU capabilities.

Additionally, for our experiments we have used sparse graphs whose number of nodes is 1 049 088. This size has been chosen because it is a multiple of all recommended values for the threadBlock size. However, due to the large amount of computational load needed to solve the APSP in these graphs, we have bounded the APSP problem to smaller problem sets in order to reduce the global execution time.

©CMMSE ISBN: 978-84-616-2723-3

6.2 Target Architectures

The GPU device used for our experiments is the GeForce GTX 480 that has a Fermi CUDA architecture (GF 110 series). The experiments have been launched using CUDA 4.2 toolkit and the 195.36 64-bit driver. Regarding the host machine, we used an Intel(R) Core(TM) i7 CPU 960 3.20GHz, 64-bits compatible, with a global memory of 6 GB DDR3. It runs an UBUNTU Desktop 10.10 (64 bits) operating system.

6.3 Input Set Characteristics

The input set is composed by a collection of graphs randomly generated by a graph-creation tool used by [16] in their experiments. The graphs have been created adding seven adjacent predecessors to each node of the graph. Afterwards, they have inverted the graphs in order to store the node successors sequentially. These graphs are represented through adjacency lists, with the nodes numbered from $0 \ldots |V| - 1$, and integers that randomly range from $1 \ldots 10$ for edge weights.

6.4 Tuning the threadBlock and concurrent kernels

We have chosen the recommended threadBlocks sizes that maximize the SM occupancy in the Fermi architecture (192, 256, 384, 512 and 768), as we have explained in Sect. 5.1. Moreover, following the recommendations described in [10, 11], we have tested a lower value for this threadBlock size, that is 64, 96 and 128 threads per block, with a medium ratio of SM occupancy. Regarding the grid and the block shape, both have a single horizontal dimension because the graph is stored in this way.

We also wanted to check if the deployment of more concurrent kernels will lead to better performance times due to a good resource exploitation. In order to test this behaviour, we have evaluated the GPU solution for the APSP with the following number of concurrent kernels: 1, 2, 4, 8, 16, 32 and 64. As long as Fermi architecture cannot support more than 16 concurrent kernels, we suppose that the performance gain from this point will not be significantly improved. The kernels launched after this limit are stored in a queue to later be dispatched.

Since the threadBlock size and the number of concurrent kernels are closely related to the use of hardware resources, they should be evaluated testing all combinations of these parameters. First, we want to obtain the best configuration from the recommended values for both threadBlock size and number of concurrent kernels for the GPU implementation. Due to the large amount of computational time involved to solve the APSP problem we have only launched this first experiment by computing only the distance from 1 024, 2 048 and 8 192 source nodes to all nodes. These values are selected because they are multiples of the chosen numbers for the concurrent kernels launched.

©CMMSE ISBN: 978-84-616-2723-3

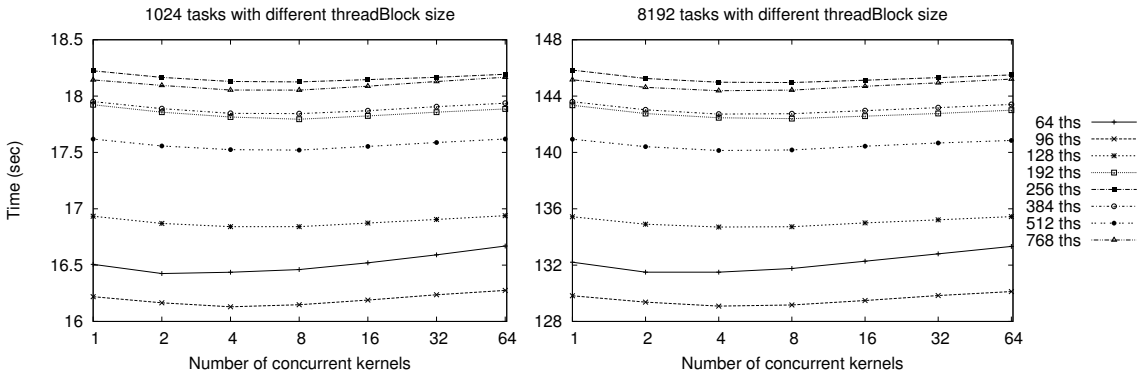

Figure 1: Relax-kernel execution times for different input sets with different configurations between the number of threads per block (ths) and number of multi-kernels.

Second, we want to show the performance difference between the worst and the best recommended configurations and to evaluate its scalability with bigger input sets. So, we have also evaluated the performance for the 16 384 and 32 768 source-node to all.

7 Experimental results

In this section we present the experimental results obtained for the different parameter configurations evaluated and the behaviour of the worst and the best configurations for the *relax kernel*.

7.1 Tuning the threadBlock and concurrent kernels

Figure 1 shows the execution times for the different configurations for the 1 024-source-node to all (left) and 8 192-source-node to all (right). The 4 096-source-node to all plot is not shown because it presents a similar performance. In all cases, the best configuration from the recommended values is reached with a block size of 96 threads and the execution of 4 concurrent kernels.

The results show that there are performance improvements from using one kernel until four (or eight, for bigger threadBlock size) concurrent kernels due to the exploitation of the data-caches and block/warp dispatchers. Through the CUDA Visual Profiler we have observed that the use of concurrent kernels slightly reduces the number of L1 data-caches misses for this kind of kernels. However, although the performance times from this point are also good, the performance cannot be improved any more by introducing new kernels, because the global memory bottlenecks and data-cache trashing effects increase even more.

 ISBN: 978-84-616-2723-3

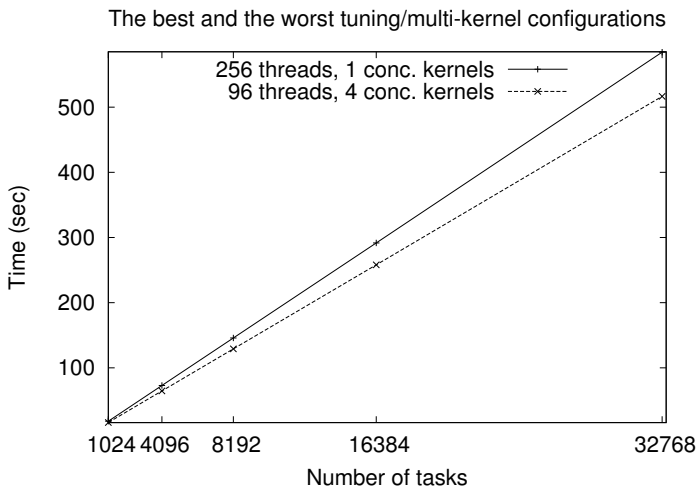

Figure 2: Execution times of the relax kernel for the best and the worst tuning/multi-kernel configurations.

7.2 Total performance gain

From the previous experiments, the worst configuration is obtained with 256 threads per block and 1 single concurrent kernel. On the other hand, the best results are reached with 96 threads per block and 4 concurrent kernels. Figure 2 shows the execution times for the best and the worst recommended configurations for bigger problems as 16 384 and 32 768-source-node to all. The global performance gain reached between the worst configuration (256 threads + 1 kernel) and the best (96 threads + 4 concurrent kernels) is an 11.5 percent.

8 Conclusions

In this paper we have used two CUDA tuning techniques, namely, the choice of a good threadBlock size and the use of concurrent kernels, to squeeze the performance of the *relax kernel* of the GPU solution to the APSP problem. We have observed that the best configuration of these two tuning techniques is reached with 96 threads per block and the use of 4 concurrent kernels. The result of using these tuning techniques leads to a performance gain up to 11.5 percent.

The results of our experiments corroborate the conclusions of the work described in [10]. For kernels with the same features as our *relax kernel*, an optimal performance is achieved if smaller threadBlock sizes that do not reach the maximum occupancy are used. This occurs because the blocks can be evicted from the SM quicker than other maximum-occupancy configurations, and the global memory bottleneck is alleviated.

©CMMSE ISBN: 978-84-616-2723-3

The characteristics of our *relax kernel* lead us to think that the performance could be improved even more by changing the configuration of the L1 data-cache. Due to our kernel is not full coalesced, the augmentation of the L1 cache will reduce the thrashing effect of this memory. Consequently, the amount of data transferred between the GPU memory hierarchy could be significantly decreased. Our future work includes the better exploitation of the L1 data-cache by choosing a good size configuration for this memory.

Additionally, also as future work, we will extend the techniques used, following the guidelines proposed in [10], to optimize the remaining kernels of the APSP-GPU implementation.

Acknowledgements

The authors would like to thank P. Martín, R. Torres, and A. Gavilanes, for letting us to use the graph-creation tools described in [16]. This research is partly supported by the Spanish Government (TIN2007-62302, TIN2011-25639, CENIT OCEANLIDER, CAPAP-H networks TIN2010-12011-E and TIN2011-15734-E), Junta de Castilla y León, Spain (VA094A08, VA172A12-2), the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative, and the ComplexHPC COST Action.

References

[1] J. Barceló, E. Codina, J. Casas, J. L. Ferrer, and D. García, "Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems," *J. Intell. Robot. Syst.*, vol. 41, pp. 173–203, 2005.

[2] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao, "The exact distance to destination in undirected world," *The VLDB Journal*, vol. 21, no. 6, pp. 869–888, Dec. 2012.

[3] G. Rétvári, J. J. Bíró, and T. Cinkler, "On shortest path representation," *IEEE/ACM Trans. Netw.*, vol. 15, pp. 1293–1306, December 2007.

[4] F. Huc, A. Jarry, P. Leone, and J. Rolim, "Brief announcement: routing with obstacle avoidance mechanism with constant approximation ratio," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing*, ser. PODC '10. New York, NY, USA: ACM, 2010, pp. 116–117.

[5] D. Gkorou, J. Pouwelse, D. Epema, T. Kielmann, M. van Kreveld, and W. Niessen, "Efficient approximate computation of betweenness centrality," in *16th annual conf. of the Advanced School for Computing and Imaging (ASCI 2010)*, 2010.

©CMMSE ISBN: 978-84-616-2723-3

[6] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[7] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–345, Jun. 1962.

[8] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.

[9] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "A new GPU-based approach to the shortest path problem," 2013, accepted in HPCS 2013.

[10] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "uBench: Exposing the impact of CUDA block geometry in terms of performance," *The Journal of Supercomputing*, pp. 1–14, 2013.

[11] Y. Torres, A. Gonzalez-Escribano, and D. Llanos, "Using Fermi architecture knowledge to speed up CUDA and OpenCL programs," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, 2012, pp. 617–624.

[12] NVIDIA, "NVIDIA CUDA Programming guide 4.2," 2012.

[13] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of Dijkstra's shortest path algorithm," in *Mathematical Foundations of Computer Science 1998*, ser. LNCS, L. Brim, J. Gruska, and J. Zlatuška, Eds. Springer Berlin / Heidelberg, 1998, vol. 1450, pp. 722–731, 10.1007/BFb0055823.

[14] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, Feb. 2010.

[15] J. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, may 2010.

[16] P. Martín, R. Torres, and A. Gavilanes, "CUDA solutions for the SSSP problem," in *Computational Science – ICCS 2009*, ser. LNCS, G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, Eds. Springer Berlin / Heidelberg, 2009, vol. 5544, pp. 904–913, 10.1007/978-3-642-01970-8_91.

[17] P. Harish, V. Vineet, and P. J. Narayanan, "Large graph algorithms for massively multithreaded architectures," Centre for Visual Information Technology, I. Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74, Feb. 2009.

[18] C. Verdiere *et al.*, "Introduction to GPGPU, a hardware and software background," *Comptes Rendus Mecanique*, 2010.

©CMMSE ISBN: 978-84-616-2723-3