A New GPU-based Approach to the Shortest Path Problem

Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos, and Arturo Gonzalez-Escribano
Dept. Informática, Universidad de Valladolid, Spain.
{hector|yuri.torres|diego|arturo}@infor.uva.es

***Abstract*—The Single-Source Shortest Path (SSSP) problem arises in many different fields. In this paper we present a GPU-based version of the Crauser *et al.* SSSP algorithm. Our work significantly speeds up the computation of the SSSP, not only with respect to the CPU-based version, but also to other state-of-the-art GPU implementation based on Dijkstra, due to Martín *et al.* Both GPU implementations have been evaluated using the last Nvidia architecture (Kepler). Our experimental results show that the new GPU-Crauser algorithm leads to speed-ups from 13× to 220× with respect to the CPU version and a performance gain of up to 17% with respect the GPU-Martín algorithm.**

***Keywords*—Dijkstra; GPU; Kepler; NSSP; Parallel Algorithms; SSSP**

I. INTRODUCTION

Many problems that arise in real-world networks imply the computation of the shortest path and its distances from a source to one or more destination points. Some examples include car navigation systems [1], traffic simulations [2], spatial databases [3], Internet route planners [4], and web searching [5]. Algorithms to solve the shortest-path problem are computationally costly, and in many cases commercial products implement 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 Single-Source Shortest Path (SSSP) problem is a classical problem of optimization. Given a graph $G = (V, E)$, a function $w(e) : e \in E$ that associates a weight to the edges of the graph, and a source node s, it consists on computing the shortest paths from 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.

The classical algorithm that solves the NSSP problem is Dijkstra's algorithm [6]. Being $n = |V|$ and $m = |E|$, the complexity time of this algorithm is $O(n^2)$. 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. Other algorithms for this problem, such as the Bellman-Ford algorithm, are more easily parallelizable. However, with an asymptotical complexity of $O(m \cdot n)$, this algorithm is not as efficient as Dijkstra's algorithm solving this problem.

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. The application of GPGPU computing to accelerate computational problems related with shortest-path problems have increased during the last years. Some GPU-solutions to the NSSP problem have been previously implemented using different algorithms as the Dijkstra's algorithm in [7], [8], Contraction Hierarchies based approach in [9], and Bellman-Ford algorithm in [10] among others.

In this paper we present an adapted version of the Crauser's algorithm [11] to the GPU architectures and an experimental comparison with both CPU and GPU implementations of Martín *et al.* [7]. We have measured these GPU implementations with the last CUDA architecture, named Kepler.

The experimental results show a speed-up from 13× to 220× with respect to the CPU times and a performance improvement up to 17% with respect to the GPU-Martín algorithms.

The rest of this paper is organized as follows. Section II introduces some basic concepts and notations related to graph theory, and briefly describes both the sequential Dijkstra's algorithm and some proposed parallel implementations. Section III introduces some Kepler architecture details. Section IV explains in depth our GPU-implementation of Crauser *et al.* algorithm and the Martín *et al.* CUDA solution for the SSSP problem. Section V poses the experimental methodology and used platform, and the input sets considered. Section VI discusses the results obtained. Finally, Sect. VII summarizes the conclusions we have obtained and describes some future works.

II. ALGORITHMIC OVERVIEW

A. Graph Theory Notation

We will first 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 $d(t)$.

B. Dijkstra's Algorithm

The basic solution for the Non-negative, Single-source, Shortest-Path problem (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 $D[v] \leftarrow D[f] + w(f, v)$.
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 criteria) If all nodes have been settled the algorithm finishes. Otherwise, it proceeds to step 2.

In order to recover the path, every node reached 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.

C. Parallel Versions of Dijkstra's Algorithm

We can distinguish two parallelization alternatives that can be applied to Dijkstra's approach. The first one parallelizes the internal operations of the sequential Dijkstra algorithm, while the second one performs several Dijkstra algorithms through disjoint subgraphs in parallel [12]. This paper is focused in the first solution.

The key of the parallelization of a single sequential Dijkstra algorithm resides in the inherent parallelism of its loops. For each iteration of Dijkstra's algorithm, the *outer loop* selects a node to compute new distance labels. Inside this loop, the algorithm relaxes its outgoing edges in order to update the old distance labels, that is the *inner loop*.

Parallelizing the *outer loop* implies to compute in each iteration i a frontier set F_i of nodes that can be settled in parallel without affecting the algorithm correctness. The main problem here is to identify this set of nodes v which tentative distances $\delta(v)$ from source s must be the minimum shortest-distance $d(v)$. Some algorithms that are based on this idea are [11], [13]. Parallelizing the *inner loop* implies to traverse simultaneously the outgoing edges of the frontier node. One of the algorithm presented in [14] is an example of this kind of parallelization.

III. CUDA OVERVIEW

Graphics processing units started as image processing devices. Over the years, GPUs have increased in performance, architectural complexity, and programmability. Currently, these devices are widely used for general purpose computing (GPGPU) due to application performance improvements achieved on parallel code regions.

CUDA (Compute Unified Device Architecture) [15] is the parallel computing architecture developed by Nvidia Company for global purpose applications. CUDA simplifies the GPGPU programming by means of high level API.

Kepler [16] is the latest Nvidia generation of CUDA architecture, released in early 2012. The main feature introduced by this architecture is the next generation of Streaming Multiprocessor (*SMX*). Each SMX unit has 192 single-precision CUDA cores, each one with floating-point and integer arithmetic-logic units. Every SMX includes four different warp schedulers with two dispatch units (eight per each SMX).

CUDA Branch divergence (*divergent branch*) has a significant impact on the performance of GPU programs. In the presence of a data dependent branch that causes different threads in the same warp to follow different paths, the warp

serially executes each branch path taken, disabling threads that are not on that path. Thus, divergent branch can hurt performance due to lower utilization of the execution units, which cannot be compensated for through increased levels of parallelism.

IV. PARALLEL DIJKSTRA WITH CUDA

This section describes how our implementation parallelizes the Dijkstra algorithm through the *outer loop* following the ideas of Crauser *et al.* [11]. As we have explained before, the main problem of these kind of parallelization is to identify as many nodes as possible that can be inserted in the following frontier set.

A. Defining the Frontier Set

Dijkstra's algorithm, in each iteration i, calculates the minimum tentative distance of the nodes belonging to the unsettled set, U_i. The node whose tentative distance is equal to this minimum value can be settled and becomes the frontier node. Its outgoing edges are traversed to relax the distances of the adjacent nodes.

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. Martín *et al.* [7] inserts into the frontier set, F_{i+1}, all nodes with this minimum tentative distance with the aim to process them simultaneously. Crauser *et al.* [11] introduces a more aggressive enhancement, augmenting the frontier set with nodes with bigger tentative distance. 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, and (2) the cost of its outgoing edges. Afterwards, it calculates the minimum of these computed values. Finally, those nodes whose tentative distance are lower or equal than this minimum value can be settled becoming the frontier set.

We define the concept of Δ_i as the limit value computed in each iteration i that holds that any unsettled node u with $\delta(u) \leq \Delta_i$ can be safely settled. The bigger the Δ_i value, the more parallelism is exploited. However, depending on the particular graph being processed, the use of a very ambitious Δ_i may induce overheads that destroys any performance gain with respect to sequential execution.

Our implementation of Dijkstra's algorithm follows the idea proposed by Crauser *et al.* [11] of incrementing each Δ_i. For every node $v \in V$, the minimum weight of its outgoing edges, that is, $\Delta_{\text{node } v} = \min\{w(v,z) : (v,z) \in E\}$, is calculated in a precomputation phase. For each iteration i of the external loop, having all tentative distances of the nodes in the unsettled set, we define

$$\Delta_i = \min\{(\delta(u) + \Delta_{\text{node } u}) : u \in U_i\} \quad (1)$$

Thus, it is possible to put into the frontier set F_{i+1} every node v whose $\delta(v) \leq \Delta_i$.

Algorithm 1 GPU implementation of Dijkstra's algorithm. CUDA 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*.

```
 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
```

B. Our GPU Implementation: The General Variant

The four Dijkstra's algorithm steps described in Sect. II-B) 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, 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 [17] has been modified to accomplish this task. Our *minimum kernel* is adapted in order to: (1) add the corresponding $\Delta_{\text{node } v}$ value to $\delta(v)$, and (2) compare its new assigned values to obtain the minimum one. The resulting value of the *minimum kernel* is the Δ_i.

The *update kernel* (Alg. 3, 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}.

Our implementation supports two types of graph repre-

Algorithm 3 Pseudo-code of a CUDA thread in *update kernel*.

```
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
```

sentations, both adjacency lists and matrices. The nodes are numbered from $0 \dots n-1$. Besides the basic structures to hold nodes, vertices, and their weights, three vectors are defined:

- Vector U, that stores in $U[v]$ whether node v is an unsettled node.
- Vector F, that stores in $F[v]$ whether node v is a frontier node.
- Vector δ, that stores in $\delta[v]$ the tentative distance from source to node v.

C. An Economic Variant

We have developed an additional version that needs less memory space at the cost of being less powerful than the previous variant described, that we have called the *general variant*.

Our *general variant* uses a vector of size n to store the $\Delta_{\text{node } v}$ value of each node v. Instead, the *economic variant* uses a single value, Δ_{base}, that is a lower bound for every $\Delta_{\text{node } v}$. This value is the the minimum weight associated to any edge e of the graph, $\Delta_{\text{base}} = \min\{w(e) : e \in E\}$. Then, for each iteration i of the external loop having the tentative distance of the following node to be settled, we define

$$\Delta_i = \min\{\delta(u) : u \in U_i\} + \Delta_{\text{base}} \quad (2)$$

For the development of this approach we need to calculate Δ_{base} in a precomputation phase. Note that the computation of the minimum value in the *minimum kernel* is simplified because every thread does not need to add $\Delta_{\text{node } v}$ to the tentative distance. Now, the Δ_{base} is added to the value returned by the *minimum kernel*.

Regarding the exploited parallelism degree, this *economic variant* cannot include as many nodes into frontier set as the *general variant*, leading to more iterations of the external loop. Thus, in spite of consuming less space, the exploited parallelism degree is lower than the *general variant*.

D. Martín et al. Successor Variant

In this subsection we will describe the GPU approach of Dijkstra's algorithm developed by Martín *et al.* [7]. They have presented some parallel implementations of Dijkstra's algorithm executed on the first CUDA architecture (now called pre-Fermi).

In order to parallelizes the Dijkstra algorithm, they have introduced a conservative enhancement to increase the frontier set, inserting all nodes with the same minimum tentative distance. According to our notation presented above, their frontier set of any iteration i, F_{i+1}, is composed by every node $x \in U_i$ with equal tentative distance $\delta(x)$ than Δ_i being

$$\Delta_i = \min\{\delta(u) : u \in U_i\} \quad (3)$$

Their *update kernel* also differs from ours in the frontier-set check condition, $U(v) \wedge \delta(v) = \Delta_i$.

E. Martín et al. Predecessor Variant

The authors have presented a different variant of Dijkstra's algorithm, called the *predecessor variant*. They have implemented both a sequential version for CPU, and a parallel one for GPU devices.

This variant differs from the original one, called *successors*, in the way of relaxing the tentative distances of the unsettled nodes. That is, for every unsettled node, the algorithm checks if any of its predecessors nodes belongs to the current frontier set. In that case, the tentative distance is relaxed if the new distance through this frontier node is lower than the previous one.

The GPU predecessor implementation assigns a single thread for each node in the graph. The *relax kernel* only computes those threads assigned to unsettled nodes $u \in U_i$. Every thread traverses back the incoming edges of its associated node looking for frontier nodes.

V. EXPERIMENTAL SETUP

We will first describe the methodology used for our experiments, as well as the input set problem used for each variants and the code versions evaluated.

A. Methodology

We have compared our GPU algorithm implementation, the *general approach*, with: (a) the *CPU successor variant*, and (b) the *GPU successor variant* of Martín *et al.*. We have adapted our algorithm to also support the *predecessor variant* in order to compare with (c) the *CPU predecessor variant*, and (d) the *GPU predecessor variant* of Martín *et al.*. To fairly compare the performance improvement of our algorithms, we also use the same run-time configuration of the Grid, and theadBlock size and geometry as Martín *et al.*.

Moreover, in order to check the performance degradation by reducing the memory space, both *general* and *economic variant* are also compared. This experiment is also carried out with the same kernel configuration described before.

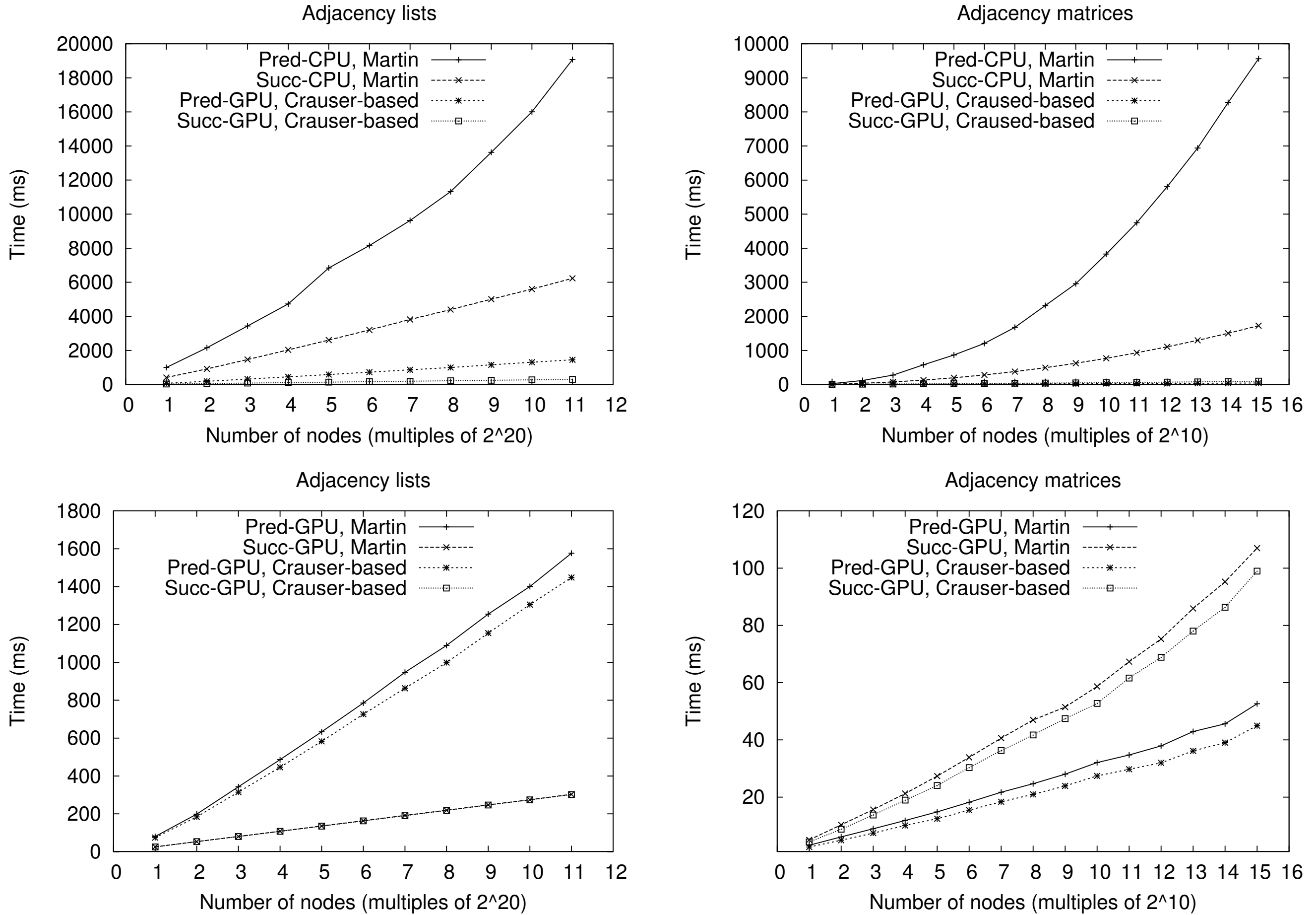

Figure 1. CPU-Martín vs. our Crauser-based GPU implementation (top) and GPU-Martín vs. our Crauser-based GPU implementation (bottom) execution times for both input sets considered.

B. Target Architectures

The performance results described by Martín *et al.* were obtained using a pre-Fermi architecture. We started by replicating these results in the same architecture, in our case using a GeForce GT 9600. After checking that results were consistent, we repeat the experiments using a GeForce GTX 680 (Kepler) Nvidia GPU device.

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) operative system. The experiments have been launched using CUDA 4.2 toolkit and the 295.41 64-bit driver.

C. Input Set Characteristics

In order to compare the results of our implementation with the implementation of Martín *et al.*, we have replicated the experimental examples using their graph creation tools. The input sets are composed by a collection of graphs generated randomly for each kind of problem, adjacency lists and adjacency matrices, with different number of nodes.

1) Adjacency Lists: The graphs stored in adjacency list have sizes that range from $1 \cdot 2^{20}$ to $11 \cdot 2^{20}$ vertices. There are 25 different graph instances for each size. We kept the degree chosen by Martín *et al.* (degree seven), so the generator tool created seven adjacent predecessors for each vertex. They had also inverted the generated graphs in order to study approaches based on the successor version. Note that the degree seven cannot be kept for these inverted graphs. The edge weights are integers that randomly range from 1 to 10.

2) Adjacency Matrices: The graphs stored as adjacency matrices have sizes that range from $1 \cdot 2^{10}$ to $15 \cdot 2^{10}$ vertices. There are also 25 different graphs instances for each size. In order to evaluate the approaches with similar features graphs, the adjacency matrices of our experiments have been designed to have also degree seven. The edge weights are integers that randomly range from 1 to 10.

D. Code Versions Evaluated

From the suite of different implementations described in [7], we have taken the fastest ones that use a full CPU and a full GPU computation. That means we have left out the hybrid approaches that mix the execution of some phases in the

CPU and others in the GPU. We denominated these relevant implementations as: (1) "Pred-CPU and Pred-GPU" for the *predecessor variants* for CPU and GPU, and (2) "Succ-CPU and Succ-GPU" for the *successor variants* for CPU and GPU.

E. Divergent Branch and Dummy Computation

The threads of the *relax kernel* for both, *predecessor* and *successor* variants, have a divergent branch. Two different kinds of threads are identified due to this divergent branch: (1) *dummy threads*, that do not make any computation for its assigned node, and (2) *working threads*, that carry out the relax operation from the assigned node.

Usually, the effect of the divergent branch is negative for the performance application due to the serialization of the work-flow. Thus, in order to discuss if its presence causes a significant performance degradation, we have carried out an experiment to measure the efficiency ratio of divergent branch, by means of CUDA VisualProfiler.

Moreover, the presence of so many dummy threads in the *kernel relax* implies to spend too much futile computation. With the aim of knowing if it is possible to compute this kernel more efficiently, we have measured both the total number of executed threads and the number of *working threads*.

VI. EXPERIMENTAL RESULTS

This section describes the performance comparison between the GPU-Crauser (general) implementation against the CPU and GPU-Martín approaches, the *economic* and *general variants* of the GPU-Crauser implementation, and the performance degradation of the divergent branch and dummy computations.

TABLE I
MARTÍN *et al.* CPU VERSIONS VS. OUR GPU IMPLEMENTATION SPEED-UPS.

Data structure	successors	predecessors
Adjacency Lists CPU vs. GPU	20.65×	13.17×
Adjacency Matrices CPU vs. GPU	17.43×	219.79×

TABLE II
PERFORMANCE IMPROVEMENT BETWEEN MARTÍN *et al.* GPU VERSIONS VS. OUR GPU IMPLEMENTATION.

Data structure	successors	predecessors
Adjacency Lists GPU vs. GPU	1.07%	8.14%
Adjacency Matrices GPU vs. GPU	7.51%	16.97%

A. Performance Improvement

Figure 1 (top) shows the execution time of *predecessor* and *successor* variants for CPU Martín *et al.* and our GPU implementation in Kepler's architecture for adjacency lists (left) and for adjacency matrices (right). We can observe in Table I a performance speed-up from 13× to 220× with respect to the CPU times.

Figure 1 (bottom) shows the execution time of *predecessor* and *successor* variants in Kepler's architecture for both GPU implementations, Martín *et al.* and ours, for adjacency lists (left) and for adjacency matrices (right). The Table II shows a performance gain up to 17% with respect to the GPU-Martín algorithm.

B. Economic vs. General

For the input set used in the experiments, the *economic variant* has a similar performance compared with the *general variant*. These graphs have seven outgoing adjacent nodes on average, with integer weights that range from $1..10$. Therefore, there is a high probability that for every node v the values $\Delta_{\text{node } v} = \Delta_{\text{base}}$, leading to an analogous behaviour. With graphs with less outgoing adjacent nodes and a bigger range of weights, this probability is decreased. In such cases, the *general variant* will take more advantage of its precomputed data with respect to the *economic variant*.

C. Divergent Branch and Dummy Computations

Figure 2 shows the total number of executed threads in *relax kernel* and the number of *working threads*. In both cases, the number of *working threads* is significantly lower than the total number of launched threads. The percentage of *dummy threads* vs. total threads goes from 42%, for the *predecessor variant* with adjacency matrices, to 96%, for the *successor variant* with adjacency lists.

The results of the CUDA VisualProfiler have shown that the efficiency ratio of the divergent branch in the *relax kernel* is good, from 94.3% to 99.5%. Thus, the serialized work-flows, due to the divergent branch, hardly have affected the performance of this kernel.

The execution of *dummy warps*, that are warps of 32 *dummy threads*, do not lead to serialize different work-flows, because all threads of these warps processes the same dummy instruction. Therefore, the fact of having much more *dummy warps* than *mixed warps*, warps filled with both *dummy* and *working threads*, is the reason because the performance is hardly affected by the divergent branch.

VII. CONCLUSIONS AND FUTURE WORK

We have developed the adaptation of Crauser *et al.* SSSP algorithm to exploit the GPU architectures. We have compared our GPU approach with the most relevant CPU and GPU implementations presented in [7], obtaining up to 220× speed-up with respect to the CPU version and a performance improvement up to 17% with respect to the GPU versions.

We have also shown that, although the *relax kernel* contains divergent branches, they do not affect significantly the performance. In spite of the good performance improvements obtained, we have detected that there is a high amount of dummy instructions executed in *relax kernel*, up to 96%. Thus, we believe that the partially or totally reduction of the dummy computational load could lead to better performance times.

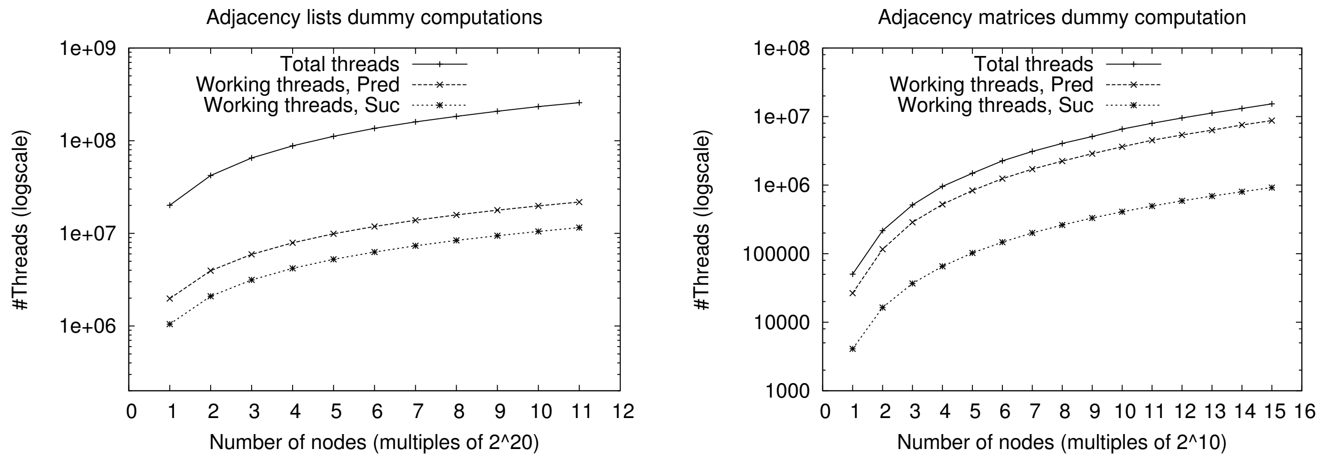

Figure 2. Total vs. Working Threads Number in the *relax kernel* for Adjacency List (left) and Matrices (right).

Our future work includes the study of performance improvements by the use of CUDA optimization techniques, and how a dynamic adjustment of the Grid and threadBlock size and geometry to the frontier-set cardinality can eliminate the global dummy workload in the *relax* kernel.

ACKNOWLEDGMENT

The authors would like to thank P. Martín, R. Torres, and A. Gavilanes, for letting them to use the source code and input sets of the algorithms described in [7]. 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] P. Sanders, D. Schultes, and C. Vetter, “Mobile route planning,” in *ESA’08*. Berlin: Springer, 2008, pp. 732–743. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87744-8_61

[2] 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. [Online]. Available: http://dx.doi.org/10.1007/s10846-005-3808-2

[3] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, “Query processing in spatial network databases,” in *VLDB’03*. Berlin: VLDB Endowment, 2003, pp. 802–813. [Online]. Available: http://dl.acm.org/citation.cfm?id=1315451.1315520

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

[5] C. Barrett, R. Jacob, and M. Marathe, “Formal-language-constrained path problems,” vol. 30, pp. 809–837, 2000.

[6] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[7] 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.

[8] P. Harishm, V. Vineet and P.J. Narayanan, “Large graph algorithms for massively multithreaded architectures,” Centre for Visual Information Technology, International Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74, Feb. 2009.

[9] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, “Phast: Hardware-accelerated shortest path trees,” *Journal of Parallel and Distributed Computing*, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151200041X

[10] S. Kumar, A. Misra, and R. Tomar, “A modified parallel approach to single source shortest path problem for massively dense graphs using cuda,” in *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, sept. 2011, pp. 635–639.

[11] 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. [Online]. Available: http://dx.doi.org/10.1007/BFb0055823

[12] D. P. Singh and N. Khare, “A study of different parallel implementations of single source shortest path algorithms,” *International Journal of Computer Applications*, vol. 54, no. 10, pp. 26–30, September 2012, published by Foundation of Computer Science, New York, USA.

[13] J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader, “Advanced shortest paths algorithms on a massively-multithreaded architecture,” in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, march 2007, pp. 1–8.

[14] M. Papaefthymiou and J. Rodrigue, “Implementing parallel shortest-paths algorithms,” in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1994, pp. 59–68.

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

[16] NVIDIA, “Nvidia geforce gtx 680,” 2012, http://www.nvidia.pl/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf, Last visit: April 2013.

[17] M. Harris, *Optimizing Parallel Reduction in CUDA*, nVidia, 2008.