CHAPTER 1

The All-Pair Shortest-Path Problem in Shared-Memory Heterogeneous Systems

HECTOR ORTEGA-ARRANZ, YURI TORRES, DIEGO R. LLANOS, PH.D., AND ARTURO GONZALEZ-ESCRIBANO, PH.D.

Departamento de Informática, Universidad de Valladolid, Spain.

This chapter faces the All-Pair Shortest-Path problem for sparse graphs combining parallel algorithms and parallel-productivity methods in heterogeneous systems. As this problem can be divided into independent Single-Source Shortest-Path subproblems, we distribute this computation space into different processing units, CPUs and graphical processing units (GPUs), that are usually present in modern shared-memory systems. Although the powerful GPUs are significantly faster than the CPUs, its combined use leads to better execution times. Furthermore, two different policies have been used for the scheduling issue, an equitable scheduling, where the workspace is equitably divided between all computational units independently of its nature, and a work-stealing scheduling, where a computational unit steals a new task when it has finished its previous work.

Keywords: *APSP, Dijkstra, GPUs, Heterogeneous Systems, Load-Balancing*

1.1 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 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 new node as source in each iteration. The classical algorithm that solves the NSSP problem is Dijkstra's algorithm [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 chapter we are going to face the APSP problem for sparse graphs combining parallel algorithms and parallel-productivity methods in heterogeneous systems. The first level of parallelism we have used is the parallelization of Dijkstra's algorithm. 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. The second level of parallelism exploited is the execution of $|V|$ simultaneous parallel algorithms. As the APSP problem can be divided into independent NSSP subproblems, we distribute the computation space into different processing units.

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 [9] and OpenCL [10], ease the general purpose programming for these heterogeneous systems with GPUs. The application of GPGPU 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 in [11, 12, 13] using some

modifications of Dijkstra's algorithm. The latter algorithm is the parallel implementation that we have used as the first level of parallelism for the GPU units in our heterogeneous system.

The new generation of high performance computing (HPC) trends to assemble different kinds of multi-core CPU and many-core GPUs in the same heterogeneous computing system. The goal of heterogeneous environments is to jointly exploit all computational capabilities of devices with different hardware-resource configurations. The different nature of these heterogeneous computational units (HCU) makes necessary to implement the same algorithm in different ways in order to take the maximum profit of each underlying architecture. However, although each HCU has its own optimized code implementation, usually some of them solve a problem faster than others due to its different resource sets. In order to palliate this imbalance and to maximally exploit the heterogeneous systems, different methods of load balancing can be applied. One of this techniques is to assign more work to the most powerful HCU, for our case the GPUs, and the remaining work to the conventional CPUs.

Load-balancing is one of the challenging problems which has a tremendous impact on the performance of parallel applications, especially in heterogeneous environments. The objective of load-balancing methods is to distribute the workload proportionally according to the computational power of the devices. In this way, these methods allow to avoid device overloads when others are idle. However, in order to obtain a good performance exploiting heterogeneous systems, the programmer needs to manually implement these load-balancing methods.

In this chapter we present parallel solutions for the APSP problem for heterogeneous systems composed by GPUs and CPU-cores that implement two different load-balancing methods. The used GPU devices have two latest architectures released by CUDA (Fermi and Kepler). Our experimental results show that the use of a heterogeneous environment for the APSP problem improves up to 65 percent the execution time compared to the fastest GPU execution used as baseline.

The rest of this chapter is organized as follows. Section 1.2 introduces some basic concepts and notations related to graph theory, and briefly describes both the sequential Dijkstra's algorithm and the parallel version used. Section 1.3 introduces some details for both Fermi and Kepler CUDA architectures. Section 1.4 describes an introduction to the heterogeneous systems and how the load-balancing techniques try to improve their performance distributing the work load. Section 1.5 explains in depth our Dijkstra GPU-implementation using the ideas presented in [14] and our heterogeneous implementations with different load-balancing methods. Section 1.6 poses the experimental methodology and used platform, and the input sets considered. Section 1.7 discusses the results obtained. Finally, Sect. 1.8 summarizes the conclusions we have obtained.

1.2 Algorithmic Overview

1.2.1 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* $P = \langle s, \ldots, u, \ldots, v, \ldots, t\rangle$ 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 $d(t)$.

1.2.2 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 $D[v] \leftarrow D[f] + w(f, v)$.

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, the algorithm 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.

1.2.3 Parallel Version of Dijkstra's Algorithm

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. The idea of inserting into the frontier set, F_{i+1}, all nodes with this minimum tentative distance in order to process them simultaneously was implemented for GPUs in [11]. A more aggressive enhancement was introduced in [14], and later implemented for GPUs in [13], 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, $\delta(u)$, and (2) the minimum cost of its outgoing edges, $\Delta_{\text{node } u} = \min\{w(u,z) : (u,z) \in E\}$. 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.

1.3 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) [15] due to the performance improvements achieved on multiple kind of parallel applications.

CUDA (Compute Unified Device Architecture) [9] 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.

Fermi [9] is the second generation of CUDA architectures, launched on early 2010, and the latest generation of CUDA architecture is Kepler [16], released on early 2012. Table 1.1 summarizes the Fermi and Kepler's main characteristics. Each new architecture generation has increased the number of SPs (Streaming Processors), and the maximum number of threads per SM (Streaming Multiprocessor). The main

change introduced by Fermi is a transparent L1/L2 cache hierarchy that has been maintained in Kepler. However, the sizes and configuration possibilities are different. The global memory is organized in several banks. The number of banks has been decreased on Fermi and Kepler. Finally, the main feature introduced by Kepler is the next generation of Streaming Multiprocessor (*SMX*) with 192 single-precision CUDA cores, four different warp schedulers and two dispatch units.

Table 1.1 Summary of CUDA architecture parameters (Fermi and Kepler)

Parameter	**Fermi**	**Kepler**
SPs (per-SM)	32	192
Max. number of blocks (per-SM)	8	16
Max. number of threads (per-SM)	1 536	2 048
Max. number of threads (per-block)	1 024	1 024
L2 cache	768 KB	$\geq$ 512 KB
L1 cache (per-SM)	0/16/48 KB	0/16/32/48 KB
Size of global memory transaction	32/128 B	32/128 B
Global memory banks	5-6	4

1.4 Heterogeneous Systems and Load-Balancing

Heterogeneous computing [17] tries to jointly exploit different kind of computational units, such as, GPUs, FPGAs or CPU-cores. Compared to traditional, symmetric CPUs, this computing paradigm offers higher peak performance while being both energy and cost efficient. However, programming for heterogeneous environments is a tedious task and has a long learning curve. The authors in [18] show the importance and the high interest of heterogeneous environments and how a heterogeneous environment could improve significantly certain kind of parallel problems.

Load-balancing methods for heterogeneous systems try to distribute the work load between any computing unit to exploit all available hardware resources. There are several load-balancing methods not only for traditional systems, but also for heterogeneous systems. A brief classification is presented bellow.

In [19] the authors create dependence graphs in order to classify as dependent or independent the application tasks. More independent tasks are launched to GPU devices in order to reduce the costly data transfers between PCI-express bus.

The work in [20] presents a model to estimate the possible execution time of each task (number of instructions and input data size), and thus, decide which hardware would be the best for each case. The size of each single task is fixed at compilation time. In [21] the author calculates the data transfer time between the different devices (GPUs and CPUs), and creates a model in order to reduce inter-GPU and CPU-

Algorithm 1.1

GPU implementation of Dijkstra's algorithm {

```
(01)   <<<initialize>>> (U, F, δ);
(02)   while (Δ ≠ ∞) {
(03)       <<<relax>>> (U, F, δ);
(04)       Δ =<<<minimum>>> (U, δ);
(05)       <<<update>>> (U, F, δ, Δ);
(06)   }
}
```

GPU communication. In [22] the authors collect all the information of each GPU hardware. By means of CUDA API and a model created by the authors, they select a good device for a given task.

For a given set of tasks with preset sizes in [23], the authors assign bigger task to more powerful devices. On the other hand, [24], for a specific problem, assign tasks with a similar size to the same device in order to equilibrate the communication imbalance factor.

A static load balancing appears when all tasks are available to schedule before any real computation starts. From the previously mentioned works, [21], [25], present static load-balancing techniques. The opposite is the dynamic load balancing. Now, the tasks are not known until mid-execution and new ones can appear. From the previously mentioned works, [26], [23], [27], and [24] present a dynamic load-balancing.

1.5 Parallel Solutions to the APSP

This section describes both the single GPU parallel implementation used as baseline and the different heterogeneous approaches implemented to solve the APSP problem.

1.5.1 GPU Implementation

We have used the implementation described in [13] for the GPU units of our heterogeneous system. It is an adaptation of the sequential Dijkstra's algorithm described in Sect.1.2.2 to the CUDA architecture (see Algorithm 1.1) following the parallel enhancements of [14]. It is composed of three kernels that executes the internal operations of the Dijkstra vertex loop:

- The *relax kernel* (invoked in line 3 of Algorithm 1.1) 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* (line 4 of Algorithm 1.1) computes the minimum tentative distance of the nodes that belong to the U_i set. To do so, the advanced *reduce3* method of the CUDA SDK [28] has been modified to accomplish this operation. 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* (line 5 of Algorithm 1.1) 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 ($U(v)\ and\ \delta(v) \leq \Delta_i$). If so, the thread assigns v to F_{i+1} and deletes v from U_{i+1}.

The nodes are numbered from $0 \ldots n-1$. 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.

1.5.2 Heterogeneous Implementation

The solution of the APSP through the V-NSSP approach allows us to divide the problem in $|V|$ independent tasks. Numbering the nodes of the graph from 0 to $|V|-1$, the task t_i solves the NSSP problem that has the node i as source.

1.5.2.1 Equitable Scheduling A simple way to apply load-balancing to a heterogeneous system is to equitably distribute the work without taking into account the computational capabilities of the devices. This kind of techniques usually lead to easy implementations, but at the expense of having a temporal cost equal to the time that the worst device needs to compute its work. Equitable Scheduling can be classified as a static load-balancing technique at compile time.

Our Equitable Scheduling (ES) approach statically divides the workspace between the computing threads giving to each one the same quantity of tasks. If nc represents the number of computing threads, id the thread identifier, and $nt = |V|/nc$ the number of tasks per thread, this approach makes each thread responsible for computing the tasks from $id \cdot nt$ to $id \cdot nt + nt - 1$. If this task division is not exact, each of the first threads takes one of the remaining tasks until there is no more work to do.

Algorithm 1.2

```
Work-stealing implementation{
 (01)    #parallel                            /* Parallel region */
 (02)    if (idThread < numGPUs){             /* For GPUs */
 (03)        selectGPU(idThread);             /* GPU selection */
 (04)        atomic{ t = steal_work(taskQueue) };
 (05)        while( t != NULL ){
 (06)              launch_GPU_Kernel(t);
 (07)              atomic{ t = steal_work(taskQueue) };
 (08)        }//while
 (09)    }else{                               /* For CPU-cores */
 (10)        atomic{ t = steal_work(taskQueue) };
 (11)        while( t != NULL ){
 (12)              launch_CPU_Kernel(t);
 (13)              atomic{ t = steal_work(taskQueue) };
 (14)    }//else
 (15)    #end parallel
}
```

1.5.2.2 Work-Stealing Scheduling Work-stealing is one of the most important techniques of load-balancing. It is commonly employed to accomplish a dynamic work scheduling between any kind of hardware device. All hardware devices of the heterogeneous system can steal a task from the global task queue. Note that the access to the global task queue must be implemented with some kind of synchronization in order to avoid that two or more devices steal the same task. Usually, this synchronization involves a bottleneck in the execution times. Work-Stealing scheduling can be classified as a dynamic load-balancing technique at runtime.

Our Work-Stealing Scheduling (WS) approach lets to an idle thread that has finished its previous work to steal the following task t_i. This task is immediately eliminated from the queue at the moment it is taken. Then, the thread computes the corresponding NSSP problem with node i as source. Finally, when the thread ends its work, it comes back to the global task queue in order to take another one, repeating the process till there is no more pending work. The synchronization of the task stealing has been implemented using an atomic region. That means that only one thread can be taking the following work at any moment.

Algorithm 1.2 is the pseudocode of work-stealing technique to solve the APSP problem. The *02* and *09* lines indicate that the first threads are assigned to GPU devices and the rest to CPU-cores. The *taskQueue* stores the list of all tasks, and the *atomic*$\{\}$ primitive creates an exclusion region to avoid a simultaneous stealing of the same task from different idle threads.

1.6 Experimental Setup

We will first describe the methodology used for our experiments, as well as the input set problems and the load-balancing techniques evaluated.

1.6.1 Methodology

We have compared our heterogeneous implementations against the single GPU implementation, that we have denominated baseline, in order to evaluate the performance gain of using heterogeneous systems for the particular APSP problem. The algorithm implemented for GPU devices is an adaptation of [14] ideas for the CUDA architecture presented in [13]. Moreover, the sequential version of this algorithm is used for the CPU devices.

Furthermore, several instances with different number of OpenMP threads, for both load-balancing methods presented, have been executed in order to determine the best configuration. These instances have been tested with graphs of $1 \cdot 2^{20}$ nodes solving the complete APSP problem. Additionally, we have used for our experiment graphs whose number of nodes is ranging from $1 \cdot 2^{20}$ to $11 \cdot 2^{20}$. However, due to the large amount of computational load needed to solve the APSP in these graphs, we have bounded the problem to a *512-source-nodes-to-all* in order to reduce the global execution time. For the selection of these source nodes we have used the random function *srand48()* from the C libraries.

1.6.2 Target Architectures

The evaluated heterogeneous system is composed by different computational units that are grouped in two categories:

- The shared-memory CPU system, the host machine, is an Intel(R) Core(TM) i7 CPU 960 3.20 GHz, 64-bits compatible, with a global memory of 6 GB DDR3, with two GPUs.

- The GPU system has two GPU devices of different architectures:

  – a GeForce GTX 680 (Kepler) Nvidia GPU device, and

  – a GeForce GTX 480 (Fermi) Nvidia GPU device.

The evaluated baseline implementation is executed in the same shared-memory host machine of the previously described heterogeneous system, but it only uses one GPU device as computational unit, that is GeForce GTX 680 (Kepler) GPU device.

Regarding the software used, the host machine runs an UBUNTU Desktop 10.10 (64 bits) operating system, and the experiments have been launched using CUDA 4.2 toolkit and the 295.41 64-bit driver.

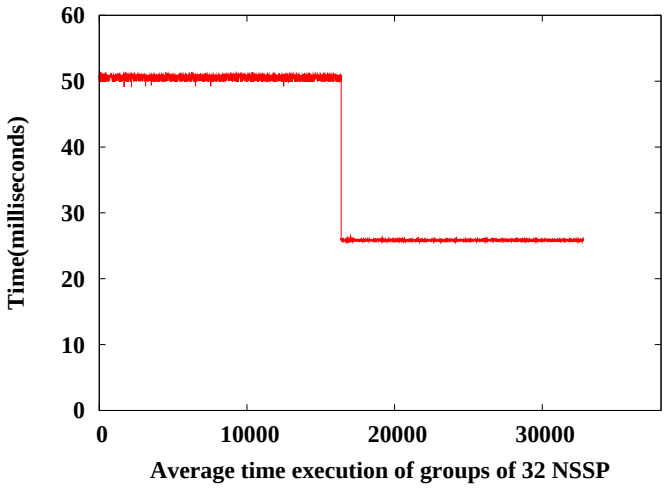


Figure 1.1 Temporal cost of the different source nodes in the graph for the Kepler GPU.

1.6.3 Input Set Characteristics

The input set is composed by a collection of graphs randomly generated by a graph-creation tool used by [11] in their experiments. They 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, the nodes are numbered from $0 \ldots |V| - 1$, and the edge weights are integers that randomly range from $1 \ldots 10$.

The node distribution of this kind of graphs shows an irregular behavior for the computational time of the APSP problem in terms of each NSSP subproblem. The iterations of the first nodes of the graph need more computational time to solve its NSSP problem than the final ones. Figure 1.1 shows, using intervals of 32 nodes, how the time needed is considerably reduced as long as the baseline implementation gets closer to the final nodes. Due to the nature of the problem, there are no inter-NSSP dependences and communication in the complete APSP computation.

1.6.4 Load-balancing Techniques Evaluated

Both load-balancing techniques described, equitable scheduling and work-stealing, have been implemented with support to different number of OpenMP threads. Several instances with different number of threads have been evaluated against the baseline implementation.

We have tagged each instance, depending which load-balancing technique implements, with the label “E” for equitable scheduling instances, and “W” for work-stealing scheduling instances, followed by a number that represents the number of OpenMP threads used (see Table 1.2). Thus, the evaluated instances “E_3” and “W_8”

Table 1.2 Experimental instances

Legend	Description
G_1	Single GPU thread (Kepler)
E_2 / W_2	2 GPU threads (Fermi & Kepler)
E_3 / W_3	2 GPU threads + 1 CPU threads
E_4 / W_4	2 GPU threads + 2 CPU threads
E_6 / W_6	2 GPU threads + 4 CPU threads
E_8 / W_8	2 GPU threads + 6 CPU threads
E_{14} / W_{14}	2 GPU threads + 12 CPU threads
E_{16} / W_{16}	2 GPU threads + 14 CPU threads

are a implementation of Equitable Scheduling with 3 threads and a implementation of Work-Stealing scheduling with 8 threads respectively.

The first two threads are always assigned to the two GPU hardware devices, one for each graphic accelerator. The rest of the threads are executed in the CPU-cores. Therefore, the instances "E_2" and "W_2" only use the GPUs resources with the corresponding load-balancing technique.

1.7 Experimental Results

In this section we present the experimental results obtained for the execution of the complete APSP with $|V| = 1{\cdot}2^{20}$ and the 512-source-to-all for graphs which number of nodes ranges from $1 \cdot 2^{20}$ to $11 \cdot 2^{20}$.

1.7.1 Complete APSP

1.7.1.1 Equitable Scheduling Figure 1.2a presents the execution times of equitable scheduling technique for instances with different number of OpenMP threads. The performance of the baseline approach (G_1) is significantly improved when a second GPU device is used (E_2). However, a $2\times$ speed-up is not reached because the architectures of the used GPUs are different. This means that the total execution time corresponds with the total execution time of the less powerful GPU device. Nonetheless, E_2 presents a 30 % of performance improvement against the baseline.

The use of one and two CPU-cores (E_3 and E_4) helps to decrease this critical execution time because the number of subproblems (SSSP problems) that the critical GPU has to resolve is reduced. The instance E_4 shows a 65 % of performance improvement. The more launched threads, the less the computation given to each device. Nevertheless, due to the irregular nature of the graph (see the distribution time in Fig.1.1), there is a threshold where the equitable partition overloads so much the work that the CPU-cores have. This occurs when the most costly tasks, that they

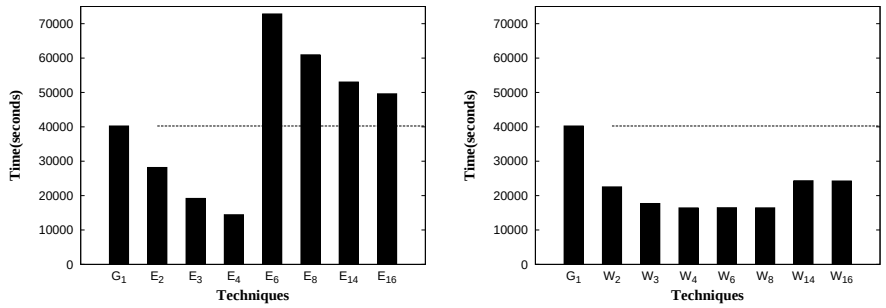


Figure 1.2 Execution times of (a) Equitable and (b) Work-Stealing Scheduling policies.

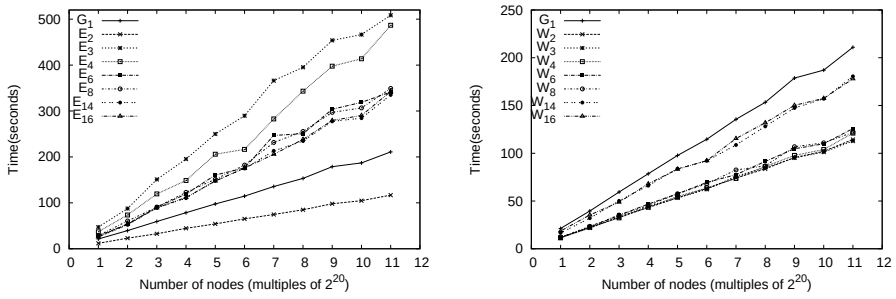
were assigned to GPUs before, are assigned to CPU-cores. For this reason, the total execution time of the approach E_6 is significantly increased even surpassing the baseline time. Furthermore, as more threads are launched from this point, the total time execution is reduced. This occurs because the number of tasks per computational unit is less and all devices are used, but the time still overpasses the baseline times.

1.7.1.2 Work-Stealing Scheduling Figure 1.2b shows the execution time results of the work-stealing technique for instances with different number of OpenMP threads. The performance of the baseline approach (G_1) is significantly improved by any experimental instance that uses the work-stealing method (W_i). The instance that uses only two GPUs has a 44 % of performance improvement against the baseline. As we increase the number of OpenMP threads, more hardware devices are used, reducing the execution times. Although the most costly tasks are also taken by the CPU-cores, while they are computing their subproblem, the GPUs are continuously stealing tasks. The instance with the fastest execution times is the W_4 instance, leading to a 60 % of performance improvement. However, when the number of launched threads exceeds the number of heterogeneous computational units (W_{14} and W_{16}), the execution of threads that belong to the same CPU-core is concurrent. This behaviour leads to slightly penalty times, reaching a performance improvement of 40 % against the baseline.

1.7.2 512-Source-Node-to-All Shortest Path

1.7.2.1 Equitable Scheduling Figure 1.3a presents the execution times for instances for the equitable scheduling implementation and different OpenMP launched threads. The best performance is obtained with the E_2 configuration, leading to a 45 % of performance improvement against the baseline.

The heterogeneous approaches with CPU-cores ($E_{\{3...16\}}$) have worse execution times than the baseline due to memory access bottlenecks. That is because the CPUs



(a) (b)

Figure 1.3 512 nodes execution times of (a) Equitable and (b) Work-Stealing Scheduling.

are taking the costly tasks due to the random nature of the 512 nodes selection. However, as it happened in the complete APSP scenario, this time is reduced when more threads are launched.

1.7.2.2 Work-Stealing Scheduling Figure 1.3b shows the work-stealing implementation for different OpenMP launched threads. As it happens in the APSP scenario, the execution time of any work-stealing instance ($W_{\{2...16\}}$) is better than the baseline (G_1). The instance of two threads that only uses GPU devices, W_2, has a very good performance against the baseline (46 % of performance improvement). Inserting an additionally CPU-core to the heterogeneous system, W_3, leads to an even better performance improvement of 47 %. However, adding more than one CPU-cores to the heterogeneous system, $W_{\{4...16\}}$, leads to slightly worse execution times compared with the best.

1.7.3 Experimental Conclusions

The best execution time for the complete APSP scenario is achieved with an equitable scheduling implementation, E_4, leading to an 65 % of performance improvement compared with the baseline G_1. However, the next approaches that closely follows this improvement are those that use a work-stealing implementation, ($W_{\{3...8\}}$), instead other equitable scheduling instances with similar thread configuration.

For the 512-source-to-all scenario, the best results are reached with a work-stealing implementation, W_3, with a 47 % of improvement compared to G_1. The equitable scheduling approach looses performance against the baseline for any thread configurations excepting the version that only uses GPUs, E_2.

These results show that (1) the equitable scheduling can be tuned up to achieve the best performance times avoiding critical code regions but it is very sensible to changes of the input graph, and (2) the work-stealing implementations have a more robust performance than the equitable scheduling because they are more independent from the graph nature.

1.8 Conclusions

We have presented solutions of the APSP problem for heterogeneous systems composed by GPUs and CPU-cores using equitable and work-stealing load-balancing techniques. These solutions have achieved a performance improvement up to 65 % compared with the baseline single-GPU solution. Moreover, the results of our experiments have shown that the work-stealing implementation with the same number of OpenMP threads have given a good performance for both tested scenarios. However, the equitable scheduling implementation that involves CPU-cores have not shown a significantly performance improvement if the nature of the graph is not taken into account.

Our first conclusion is, that the jointly use of very different computational power devices is useful to improve the total execution time compared with the fastest GPU implementation. Second, the previous study of the nature of the input problem allows us to better mapping the most costly tasks to the most powerful devices. For our case, the equitable scheduling that maps all costly tasks to the GPUs and leaves light ones to the CPU-cores leads to the best performance. Finally, the application of the work-stealing technique results in a more robust implementation against the equitable scheduling because it is less sensible to the nature of the input problem.

Acknowledgments

The authors would like to thank P. Martín, R. Torres, and A. Gavilanes, for letting us to use the source code and input sets of the graph-creation tools described in [11]. 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, VA 172A12-2), the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative, and the ComplexHPC COST Action.

H. Y. D. A.

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," *Journal of Intelligent & Robotic Systems.*, 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, pp. 869–888, Dec. 2012.
3. G. Rétvári, J. J. Bíró, and T. Cinkler, "On shortest path representation," *IEEE/ACM Transaction Network.*, 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*, PODC '10, (New York, NY, USA), pp. 116–117, ACM, 2010.
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.
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," *Communications of the ACM*, vol. 5, pp. 345–345, June 1962.
8. S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM*, vol. 9, pp. 11–12, Jan. 1962.

9. D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, Feb. 2010.
10. J. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, pp. 66–73, may 2010.
11. P. Martín, R. Torres, and A. Gavilanes, "CUDA Solutions for the SSSP Problem," in *Computational Science – ICCS 2009* (G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, eds.), vol. 5544 of *LNCS*, pp. 904–913, Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-01970-8_91.
12. P. Harish, V. Vineet, and P. J. Narayanan, "Large Graph Algorithms for Massively Multithreaded Architectures," Tech. Rep. IIIT/TR/2009/74, Centre for Visual Information Technology, International Institute of Information Technology, Hyderabad, India, Feb. 2009.
13. H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "A New GPU-based Approach to the Shortest Path Problem," in *The 2013 International Conference on High Performance Computing & Simulation, (HPCS 2013)*, p. To appear, 2013.
14. A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of Dijkstra's shortest path algorithm," in *Mathematical Foundations of Computer Science 1998* (L. Brim, J. Gruska, and J. Zlatuška, eds.), vol. 1450 of *LNCS*, pp. 722–731, Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055823.
15. C. Verdiere *et al.*, "Introduction to GPGPU, a hardware and software background," *Comptes Rendus Mecanique*, 2010.
16. NVIDIA, "NVIDIA CUDA Programming Guide 4.2: Kepler," 2012.
17. A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Journal Scientific Programming*, vol. 18, pp. 1–33, Jan. 2010.
18. S. Singh, "Computing without processors," *Communications of the ACM*, vol. 54, pp. 46–54, August 2011.
19. E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU parallelization for interactive physics simulations," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, (Berlin, Heidelberg), pp. 235–246, Springer-Verlag, 2010.
20. A. Leung, O. Lhoták, and G. Lashari, "Automatic parallelization for graphics processing units," *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java PPPJ 09*, p. 91, 2009.
21. N. R. Satish, *Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Systems*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2009.
22. A. Binotto, C. Pereira, and D. Fellner, "Towards dynamic reconfigurable load-balancing for hybrid desktop platforms," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium*, pp. 1–4, april 2010.
23. S. Tzeng, A. Patney, and J. D. Owens, "Task Management for Irregular-Parallel Workloads on the GPU," in *High Performance Graphics* (M. Doggett, S. Laine, and W. Hunt, eds.), pp. 29–37, Eurographics Association, 2010.

24. P. Yao, H. An, M. Xu, G. Liu, X. Li, Y. Wang, and W. Han, "CuHMMer: A load-balanced CPU-GPU cooperative bioinformatics application," in *HPCS'2010*, pp. 24–30, July 2010.

25. E. Burrows and M. Haveraaen, "A hardware independent parallel programming model," *Journal of Logic and Algebraic Programming*, vol. 78, pp. 519–538, 2009.

26. D. Cederman and P. Tsigas, "On sorting and load balancing on GPUs," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 11–18, 2008.

27. C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 45–55, ACM, 2009.

28. M. Harris, *Optimizing Parallel Reduction in CUDA*. NVIDIA, 2008.