J Supercomput (2014) 70:786–798
DOI 10.1007/s11227-014-1212-z

Optimizing an APSP implementation for NVIDIA GPUs using kernel characterization criteria

Hector Ortega-Arranz · Yuri Torres · Arturo Gonzalez-Escribano · Diego R. Llanos

Published online: 18 May 2014

Abstract During the last years, GPU manycore devices have demonstrated their usefulness to accelerate computationally intensive problems. Although arriving at a parallelization of a highly parallel algorithm is an affordable task, the optimization of GPU codes is a challenging activity. The main reason for this is the number of parameters, programming choices, and tuning techniques available, many of them related with complex and sometimes hidden architecture details. A useful strategy to systematically attack these optimization problems is to characterize the different kernels of the application, and use this knowledge to select appropriate configuration parameters. 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 pairs of nodes in a graph. This problem can be solved by highly parallel and computational intensive tasks, being a good candidate to be exploited by manycore devices. In this paper, we use kernel characterization criteria to optimize an APSP algorithm implementation for NVIDIA GPUs. Our experimental results show that the combined use of proper configuration policies, and the concurrent kernels capability of new CUDA architectures, leads to a performance improvement of up to 62 % with respect to one of the possible configurations recommended by CUDA, considered as baseline.

H. Ortega-Arranz · Y. Torres · A. Gonzalez-Escribano · D. R. Llanos (✉)
Dpto. Informática, Universidad de Valladolid, Valladolid, Spain
e-mail: diego@infor.uva.es

H. Ortega-Arranz
e-mail: hector@infor.uva.es

Y. Torres
e-mail: yuri.torres@infor.uva.es

A. Gonzalez-Escribano
e-mail: arturo@infor.uva.es

Keywords APSP · Cache configuration · Concurrent kernel · GPU · Kernel characterization · Threadblock size

1 Introduction

Nowadays, GPUs are among the most powerful HPC devices. However, their use with programming models such as CUDA implies to take several decisions in terms of configuration parameters. Some of these CUDA specific configuration parameters, such as the threadblock size, the configuration of the L1-cache size, and the amount concurrent kernels [9], do not need changes in an application code. Optimizations, such as prefetching, coalescing-maximization, and unrolling, that require the modification of the original algorithm are not considered in this paper. The joint use of the considered techniques can lead to significant performance improvements, but there are many possible combinations. Until now, the only way to ensure which are the best values for these configuration/optimization parameters is to carry out an empirical study that explores the complete search space. In order to avoid this costly task, CUDA recommends the use of threadblock sizes that maximize the occupancy as a proper choice for a good NVIDIA GPU performance [9]. However, the authors of [15] have shown through synthetic micro-benchmarking that this recommendation does not always correlate with the values that lead to the optimal performance. Instead, they proposed some rules to select the optimal values depending on the kernel features.

Many problems that arise in real-world networks imply the computation of shortest paths, from any source to any destination point. Some examples include traffic simulations [1], or Internet route planners [13]. 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 in a graph $G = (V, E)$ where V and E are the set of nodes and edges, respectively. It is a generalization of the Single-Source Shortest-Path (SSSP) problem that consists in computing the shortest paths from one source node s to every $v \in V$.

There are two classical ways to solve the APSP problem. The first solution is to execute $|V|$ times a SSSP algorithm selecting a new node as source on each iteration. Let m be the number of edges of the graph, and n the number of vertices of the graph. Then, the asymptotic complexity of the execution time of this approach is $O(m+n \log n) \times n$. The second solution is to execute an algorithm that globally solves the APSP problem using dynamic programming, for example the Floyd–Warshall algorithm [2] presenting an asymptotic complexity of $O(n^3)$. Sparse graphs (graphs with $m << n^2$) are representative of a wide class of problems, such as routing in different contexts [1,13]. For this kind of graphs, the $n \times SSSP$ strategy is asymptotically more efficient than the second one [2], and it is the focus of this work.

The classical algorithm that solves the SSSP problem is due to Dijkstra [5]. Although this algorithm has three steps that must be carried out sequentially and iteratively, each step can be parallelized using the GPUs. Some GPU implementations have been previously developed to solve this problem [10,12] using some modifications of Dijkstra's algorithm. The solution presented in [12] offers better performance against the other state-of-the-art GPU solutions.

The contributions of this paper are the following: (1) An extended criteria to select optimal values of the previously described parameters in terms of kernel code characterization; (2) an empirical study covering a wide range of the search space for the parameter combinations for the APSP problem; and (3) a study of the impact of the use of concurrent kernel techniques in the selection criteria, and overall performance of the APSP problem. As far as we know, there are no rules to predict the optimal values for the concurrent kernel technique, and no study to assess if its use disturbs the selection criteria for the threadblock size and cache configurations.

Our experimental results show that, for the APSP problem, the correct choice of the parameter values leads to a performance improvement of up to 62 % compared with a baseline choice based on the CUDA programming guidelines.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 introduces the chosen GPU implementation [12]. Section 4 presents our characterization of the kernels, and the predicted values for the parameters. Section 5 describes the experimental setup. Section 6 discusses the results obtained. Finally, Sect. 7 summarizes the conclusions and future work.

2 Related work

There is a lack of studies focused on the tuning of CUDA configuration parameters. A correct choice of them is critical to take advantage of GPU capabilities, even when applied previous code optimizations techniques. For example, Grauer-Gray et al. in their work [7] use annotated sequential code to generate optimized CUDA kernels. This kind of works focus on high-level code transformations, such as tiling, unrolling, permutation, and fusion-fision techniques. Ocelot [6] is a dynamic compilation framework for heterogeneous systems. This tool transforms a PTX instruction set into different formats, such as CUDA or LLVM, and optimizes the code applying unrolling and independent-instruction gathering techniques. Our kernel characterization can be applied after these code changes in order to get better performance using the same GPU solution.

Our work can also be used to complement other tuning or auto-tuning techniques. For example, Ocelot [6] analyzes the PTX virtual instruction set and returns a collection of statistics and relevant metrics related with the kernel performance. These data are used by [4] with the aim to create an analyzer of kernel performance.

3 GPU parallel version of Dijkstra's algorithm

As we described in Sect. 1, in this work, we focus on the optimization of the $n \times SSSP$ parallel approach. We have chosen the best performance SSSP implementation reported so far for GPUs [12], an implementation of the Crauser's algorithm [3]. In that implementation, the three Dijkstra's algorithm steps (relax step, minimum-computation step, and update step) have been mapped into three CUDA kernels supporting Crauser modifications.

The *relax kernel* (Alg. 1 (left)) decreases the tentative distances for the remaining unsettled nodes at the current iteration i, $u \in U_i$, through the outgoing edges of the

Algorithm 1 Pseudo-code of the *relax kernel* (left) and the *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:       δ[j] = Atomic min{δ[j], δ[tid] + w(tid, j)};
6:     end if
7:   end for
8: end if
```

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

frontier nodes $f \in F_i$. A GPU thread is associated for each node in the graph. Those threads assigned to frontier nodes traverse their outgoing edges, relaxing the distances of their unsettled adjacent nodes.

The *minimum kernel* computes the minimum tentative distance of the nodes that belongs to the U_i set plus its corresponding Crauser values. To accomplish this task, we have used the *reduce4* method included in the CUDA SDK [8], just inserting an additional sum operation per thread before the reduction loop. The result value of this reduction is Δ_i.

The *update kernel* (Alg. 1 (right)) settles the nodes that belong to the unsettled set, $v \in U_i$, whose tentative distance, $\delta(v)$, is lower or equal to Δ_i. This settling-node task extracts them from U_i to generate the following-iteration unsettled set, U_{i+1}, and putting the extracted ones into 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, it assigns v to F_{i+1} and deletes v from U_{i+1}.

4 Kernel characterization

In this section, we describe the APSP kernel characterization with respect to a classification criteria based on the work presented in [15]. This criteria is based on three kernel code features that can be obtained by inspection of the source code: *Memory access pattern*, *Computational load ratio*, and *Data sharing*. We will refine the criteria by determining new specific value ranges, and classification methods for each of these three features. These ranges have been determined by experimental measures for the platforms considered [14].

Memory access pattern: It refers to how each thread accesses the global memory positions in an instant of time. Three different kind of patterns are defined as follows: *(a) Full-coalesced:* Each warp requests only one transaction segment (also known as cache line) at the same time. This means that every thread is requesting data in the same segment, and therefore, the number of memory requests is small. The memory requests are overlapped with the instruction computation or the memory request latencies of the following warps. This overlapping is optimized when the SM occupancy is maximized [9]. *(b) Medium-coalesced:* Each warp requests between two and four transaction segments at the same time. This means that there are threads of the same warp that request data from different segments. Thus, the overlapping benefits of computation and global memory latencies depend on other kernel features described below. *(c) Scatter:* Each warp requests more than four transaction segments. Thus, the

number of memory requests significantly increases with respect to the full-coalesced pattern needing more computational load warps to compensate the memory latencies.

Computational load ratio (CLr): It refers to the ratio of logic or arithmetic instructions per thread compared to the memory accesses of the same thread. Note that this metric is related to the *operational intensity* metric of [16], but our metric considers the number of memory transfers independently of their number of bytes, which differs depending on the L1 activation configuration (32 bytes when deactivated, and 128 otherwise). We define three ranges for this ratio: *Low*, ratio values between 0 and 10; *Medium*, ratio values between 10 and 100; and *High*, ratio values over 100. Note that during the time that one or more warps are computing, other warps blocked due to global memory request can finish their communication phase, hiding the memory latencies. Broadly described, the communication-computation overlapping is optimized using maximum-occupancy threadblock sizes for full-coalesced memory access patterns with a low CLr, medium-coalesced memory access patterns with a medium CLr, and scatter access patterns with a high CLr.

Data sharing across blocks ratio (DS): It refers to the ratio of data sharing compared to the number of memory accesses per thread. We name the limit values for this metric as: *High DS*, when all threads of a block re-uses all values fetched by other threads; *Low DS*, when there is no DS (each thread accesses to different data). We consider as *Medium DS* all situations between the limits described. When there is a high DS in a kernel, the recommendation in [15] is to increase the number of threads per block in order to take profit of the data present in the cache memories. Moreover, they recommend to augment the L1 cache size in order to store more reused values for Fermi architecture, or to increase the L1 local data bandwidth for Kepler's.

4.1 Input set parameters

Let $d(G)$ be the mean fan-out degree of the input graph $G(V, E)$, defined as the mean number of outgoing edges for the graph nodes, and $n = |V|$. For the APSP problem, the kernel characterization criteria not only depends on the kernel programming but also on these graph properties. We define as *low d(G) graphs* those graphs with $d(G) \in [1, 20)$. *Medium d(G) graphs* those graphs with $d(G) \in [20, 200]$. And *high d(G) graphs* those graphs with $d(G) > 200$.

The behavior of the GPU differs when the number of launched threads overpasses the maximum limit of active threads. In this case, we say that the GPU enters in a stressed situation. Let *stressing ratio* (*st*) be the ratio between the number of launched threads and the maximum number of active threads in the GPU device. For the APSP problem, the number of launched threads is equal to n. Thus, we can associate this graph property with the previously defined ratio, $st(G)$. We define *low* $st(G)$ as those with $st(G) \in (0, 1.5]$. *Medium* $st(G)$ as those with $st(G) \in (1.5, 3]$. And *high* $st(G)$ as those with $st(G) > 3$.

Depending on these two graph parameters, the number of required hardware resources and the memory hierarchy bottlenecks vary for the programmed kernels. Thus, the recommendation for proper values of the configuration parameters depending on the kernel characterization could vary.

Table 1 Characterization of the APSP kernels depending on the stressing ratio (A), memory access pattern (B), computational load ratio (C), and data sharing across blocks (D)

Kernel	A	B	C	D
Relax	Low	Medium-coalesced	Low	Low
	Medium	Medium-coalesced	Low	Ledium
	High	Medium-coalesced	Low	High
Minimum	Low/med./high	Full-coalesced	Medium	Low
Update	Low/med./high	Full-coalesced	Low	Low

4.2 APSP kernel characterization

Table 1 summarizes the *relax*, *minimum*, and *update* kernel properties using the classification criteria described above. **Relax kernel:** The first condition [(line 2 of Alg. 1 (left)] performs coalesced memory accesses, whereas the inner instructions carry out low- or medium-coalesced accesses. Therefore, we consider that it has a medium-coalesced pattern. The CLr is low because there are less than 10 logic/arithmetic instructions on the code of Alg. 1, and several global memory accesses. Finally, the DS will increase as $d(G)$ and n increase. **Minimum kernel:** It has a full-coalesced pattern because contiguous threads access to contiguous memory addresses. It has a medium CLr. The DS is not affected by $d(G)$ nor n, and hardly any DS is performed (low DS). **Update kernel:** All data structures are accessed with a full-coalesced pattern, and there is just one instruction per access (low CLr). There is no DS present in this kernel.

4.3 APSP predicted values

A trial-and-error optimization of the GPU performance through the considered techniques would imply a very large experimentation space due to the big number of possible combinations. Through the kernel characterization described above and the guidelines proposed in [15], we predict which configuration values lead to good performance for this GPU implementation. **Relax kernel:** A medium-coalesced access pattern with a low CLr suggests the use of the minimum value of the threadblock sizes that maximize the occupancy. Depending on the $st(G)$ the kernel DS across blocks varies. As this reutilization increases, the optimal threadblock size is higher: (a) For low $st(G)$: The lowest value that maximizes the occupancy (192 for Fermi, and 128 for Kepler); (b) For medium $st(G)$: The two lowest values that maximize the occupancy (192/256 for Fermi, and 128/256 for Kepler); (c) For high $st(G)$: Even higher values than the ones predicted for medium $st(G)$ due to high reutilization (192/256/384 for Fermi, and 256 for Kepler). **Minimum kernel:** A full-coalesced access pattern with a medium CLr suggests the use of the minimum value of the threadblock sizes that maximize the occupancy, or even slightly lower values, although they do not fully maximize the occupancy. (128/192 for Fermi, and 96/128 for Kepler). **Update kernel:**

A full-coalesced pattern with a low CLr leads to the use of the minimum value of the threadblock sizes that maximize the occupancy. (192 for Fermi, and 128 for Kepler).

L1 cache management. There are three possibilities: without cache L1, normal state, and increased state. In kernels where the shared memory is not totally filled up, the increase on the size of the cache L1 memory does not lead to a performance degradation, and it alleviates memory thrashing effects. This is the case for the APSP kernels. The number of requested transaction segments from global memory will be highly increased for high $st(G)$. When there is too much thrashing in the L1 cache, the performance can be increased by deactivating it, and consequently reducing the size of these segments (from 128 to 32 bytes). Therefore, our conclusion is that disconnecting the cache L1 memory for high $st(G)$ will lead to performance improvements.

5 Experimental setup

In this section, we describe an experimental study to evaluate the benefits in terms of performance of the described configuration and optimization techniques and to validate the predictions based on kernel characterizations.

5.1 Methodology, platform, and input sets

We experimentally tested a wide range of combinations of the possible configuration values, across the full search space. We issued this experimental study for the APSP problem, measuring the execution time of the whole program and of each kernel separately. The measures were repeated with different input sets, and for two different architectures. The values obtained were then compared with the predictions suggested in Sect. 4 by the kernel characterization.

Target architectures: The experiments have been carried out using the CUDA Toolkit 4.2, and the GPU devices GTX 480 (Fermi GF 110) and GTX 680 (Kepler GK 104) with 23,040 and 16,384 maximum concurrent threads, respectively. The host machine used is an Intel(R) Core(TM) i7 CPU 960 3.20GHz, with 6 GB of memory with an Ubuntu Desktop 10.10 (64 bits).

Input set characteristics: The input set is composed by a collection of random graphs generated with a technique designed to produce random structures [11]. This decision was taken in order to: (1) Avoid dependences between a particular graph structure of the input sets and performance effects related to the exploitation of the GPU hardware resources; and (2) avoid focusing on specific domains, such as road maps, physical networks, or sensor networks among others, that would lead to loss of generality. Because the GPUs have more integer arithmetic computational units than float ones, the weight of each edge is chosen as a random integer number uniformly distributed in the range $[1 \ldots 100]$.

In order to evaluate how the studied techniques are dependent on some graph features, we have generated a collection of nine kind of graphs using three sizes $n \in \{24\,576, 49\,152, 98\,304\}$ and three fan-out degrees $d(G) \in \{2, 20, 200\}$. These sizes have been chosen to have graphs with the stressing ratio in the three levels previously defined (24k–low, 49k–medium, 98k–high).

5.2 Considered values for CUDA configuration/optimization techniques

Threadblock sizes: CUDA recommends the use of threadblock sizes that maximize the SM occupancy of the GPU [9]. These sizes are dependent on the GPU architecture, being 192, 256, 384, 512, and 768 for Fermi, and 128, 256, 512, and 1,024 for Kepler. Nevertheless, following the recommendations of [15], we have also evaluated lower values (occupancy ≥ 0.75). Note that all values should be multiples of 32 (warp size) in order to maximize the core exploitation of the SM. Consequently, the block sizes that lead to a medium ratio of SM occupancy have been also tested (96 and 128 for Fermi, and 96 for Kepler). The grid and blocks use one-row shapes to couple the thread indexes with the elements of the unidimensional array where the graph is stored.

Cache L1 state: We have included in our experimentation the three different states that this cache can adopt: Normal state (16K of cache L1), Augmented state (48K of cache L1), and No-L1 state (cache L1 deactivated).

Number of concurrent kernels: We have evaluated the following number of concurrent kernels: 1, 2, 4, 8, 16, and 32. The concurrent kernel technique allows to send several instances of the same kernel simultaneously with a synchronized end. Each of these instances operates in a different memory workspace to solve its corresponding SSSP task. Due to the concurrent kernel synchronization, these memory workspaces are transfered with a single operation.

Best versus Baseline values: Once we have obtained the best configuration combining the three techniques, we compare it with what we have called the baseline configuration. This configuration applies the following values: (1) From the best threadblock sizes recommended by CUDA, the one that returns the lowest performance; (2) No cache modifications (normal state); and (3) No kernel concurrency (one kernel at a time).

6 Experimental results and their evaluation

In this section, we present and discuss the experimental results through a performance comparison between the best configuration and the baseline one, between the optimal values obtained and the predicted ones, the lessons learned that could be applied to other kernels, and the architecture differences.

Performance improvements for the APSP scenarios. Figure 1 shows the execution time breakdown of the different scenarios chosen for Fermi architecture. For each scenario, we present the baseline and the best execution times experimentally obtained for any tested configuration value combination, together with the performance gain percentages. The execution breakdown of Kepler is similar. For Fermi, the use of the considered techniques turns out in a global performance gains from 21.5 to 53.9 %, whereas for Kepler they are in the range from 33.75 to 58.53 %.

The joint use of these optimization techniques not only returns performance improvements in the total execution time of the GPU implementation, but also it always improves the execution time of each kernel independently (see non-white boxes in Fig. 1). Comparing the best values of the configuration parameters with the results for baseline values, we obtain the following performance gains for Fermi: (1)

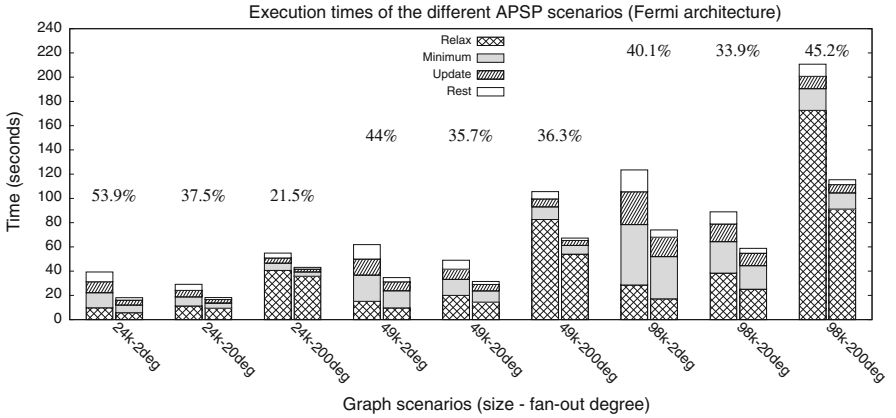

Fig. 1 Execution time breakdown of the APSP kernels as well as other operations (data transfers) on the different scenarios, and the performance gain percentages for Fermi architecture between the baseline (*left column*) and the best (*right column*) configurations

relax kernel from 12 to 47.2 %, (2) *minimum* kernel from 25.4 to 48.5 %, and (3) *update* kernel from 29.2 to 54.4 %. For Kepler, the performance gains are as follows: (a) *relax* kernel from 28.8 to 62.4 %, (b) *minimum* kernel from 27.1 to 43.8 %, and (c) *update* kernel from 27.3 to 45.3 %.

Additionally, the use of the concurrent kernel technique has significantly reduced the memory transfer times between CPU and GPU (see the white boxes in Fig. 1). This transfer time reduction is due to the fact that it is more efficient to transfer bigger data sets than a lot of smaller data sets. Due to the data offloading-transfer management chosen for our implementation, the higher the number of concurrent kernels, the bigger the amount of data in a single transfer, and the less transfer iterations needed.

Optimal values for configuration/optimization techniques. Table 2 shows the best and the baseline values of CUDA parameters for the APSP kernels. The first column contains the graph properties, n and $d(G)$. The remaining ones show the best values found for the studied parameters with the performance gain between the best and the baseline configuration. These optimal values match the ones predicted through the kernel characterization (see Sect. 4).

Summary of optimization guidelines. For those kernels of any application that fulfill the same characterizations presented in Table 1, we expect that they will present the same performance behavior when using the same configuration values. Thus, we can extrapolate the following optimization guidelines:

1. *Use the minimum size that maximizes the occupancy for kernels with a:* (i) Full-coalesced access pattern, a low CLr, and a low DS (see all scenarios of *update* kernel in Table 2c). (ii) Medium-coalesced access pattern, a low CLr, and a low DS (see 24k-scenarios of *relax* kernel in Table 2a). (iii) Full-coalesced access pattern, a medium CLr, a low DS, and high stressing ratio (see 98k-scenarios of *minimum* kernel in Table 2b).

Table 2 The best values of configuration parameters (threadblock size, number of concurrent kernels, and L1 cache state) obtained experimentally for the *relax*, *minimum* and *update kernels* and their percentage of performance gain with respect to the baseline configuration

Graph properties	Fermi GF110	Kepler GK104
(a) Relax kernel		
24K-nodes deg2	192-8-Increased→ 41.4 %	128-4-Increased→ 35.3 %
24K-nodes deg20	192-4-Increased→ 16.5 %	128-4-Increased→ 28.8 %
24K-nodes deg200	192-1-Increased→ 12.0 %	128-1-Increased→ 30.8 %
49K-nodes deg2	192-4-Increased→ 35.4 %	256-4-Increased→ 34.3 %
49K-nodes deg20	192/256-2-Increased→ 28.2 %	256-2-Increased→ 38.3 %
49K-nodes deg200	256-1-Increased→ 34.0 %	256-1-Increased→ 49.4 %
98K-nodes deg2	192-8-Increased→ 39.7 %	256-8-Increased→ 40.9 %
98K-nodes deg20	256-2-Increased→ 34.8 %	256-2-Increased→ 48.9 %
98K-nodes deg200	384-1-Increased→ 47.2 %	256-1-Increased→ 62.4 %
Baseline configurations	768-1-Normal	1024-1-Normal
(b) Minimum kernel		
24K-nodes deg2	128-8-Increased→ 48.5 %	96-8-Increased→ 43.8 %
24K-nodes deg20	128-4-Increased→ 41.6 %	96-8-Increased→ 38.1 %
24K-nodes deg200	128-4-Increased→ 41.1 %	96-8-Increased→ 35.5 %
49K-nodes deg2	128-4-Increased→ 34.7 %	96-8-Increased→ 40.5 %
49K-nodes deg20	128-2-Without→ 30.0 %	96-4-Increased→ 34.1 %
49K-nodes deg200	128-2-Without→ 30.5 %	96-4-Increased→ 34.7 %
98K-nodes deg2	192-4-Without→ 30.0 %	128-4-Increased→ 36.8 %
98K-nodes deg20	192-2-Without→ 25.4 %	128-2-Without→ 27.1 %
98K-nodes deg200	192-2-Without→ 26.0 %	128-2-Without→ 28.1 %
Baseline configurations	768-1-Normal	1024-1-Normal
(c) Update kernel		
24K-nodes deg2	192-8-Increased→ 54.4 %	128-8-Increased→ 45.3 %
24K-nodes deg20	192-8-Increased→ 45.5 %	128-8-Increased→ 36.8 %
24K-nodes deg200	192-8-Increased→ 45.0 %	128-8-Increased→ 34.5 %
49K-nodes deg2	192-8-Increased→ 44.6 %	128-8-Increased→ 37.4 %
49K-nodes deg20	192-8-Increased→ 37.4 %	128-8-Increased→ 28.6 %
49K-nodes deg200	192-8-Without→ 38.1 %	128-8-Without→ 29.6 %
98K-nodes deg2	192-8-Without→ 41.0 %	128-8-Without→ 40.2 %
98K-nodes deg20	192-8-Without→ 29.2 %	128-8-Without→ 27.3 %
98K-nodes deg200	192-8-Without→ 32.0 %	128-8-Without→ 30.4 %
Baseline configurations	512/768-1-Normal	1024-1-Normal

2. *Use lower sizes than the minimum one that maximize the occupancy for:* Full-coalesced patterns with a medium CLr, a low DS, and a low/medium *st* kernels (see 24k and 49k scenarios of *minimum* kernel in Table 2b).

3. *Jump to higher maximizing occupancy sizes from the minimum one for:* Medium-coalesced patterns with a low CLr, and a medium/high DS kernels (see 49k and 98k-scenarios of *relax* kernel in Table 2a).
4. *Increase the L1 cache size for non-high stressing situations for the GPU, or high data sharing situations.* As predicted, the use of a bigger cache L1 size speeds up the execution time compared with the normal size configuration, because thrashing effects are alleviated, and compared with a deactivated cache configuration, because the global memory requests are slower than cache ones. That is the case of 24k-scenarios (*low st*(G)) and most of the 49k-scenarios (*medium st*(G)), and the 98k-scenarios of *relax* kernel (*high DS*).
5. *Deactivate the L1 cache size when the GPU enters in a high stressed state with non-high data sharing.* In the 98k-scenarios (*high st*(G)) with non-high DS, the number of memory accesses is increased, due to the thrashing effect. As it was predicted, in order to alleviate the memory traffic, it is better to disconnect the L1 cache, due to the reduction of the transaction segment size.

Concurrent kernel technique. The results suggests that the use of the concurrent kernel execution with the other two techniques does not interfere with the predictions made. In all cases, the best configuration values using only one concurrent kernel matches the same value using more concurrent kernels.

Using this technique, a performance improvement in all scenarios is obtained; up to 52.8 % for Fermi, and up to 44.3 % for Kepler (both cases for the *update* kernel in 24k-nodes deg2), with the exception of the deg200 cases for the *relax* kernel. In there, it was better to leave a sequential-kernel execution instead launching concurrent kernels. This occurs because the number of memory accesses significantly increases when each thread is looking for the "200" successors of its corresponding frontier node (see line 3 of Alg. 1 (left)). The minimum kernel has also to carry out more memory accesses when calculating the minimum value for graphs with bigger $d(G)$ because there are more reached nodes with tentative distance values to compute in each iteration. This effect can be seen in the deg200 scenarios, where the optimal concurrent kernel number is lower compared to the other cases. Otherwise, for the *update* kernel, the increment of $d(G)$ of each graph does not lead to more memory accesses, so the optimal number of concurrent kernels is the same for all cases.

Following these results, we can include into the programmer guidelines the recommendation of reducing the number of concurrent kernel deployment for kernels with big number of memory accesses.

Similar performance behavior for Fermi and Kepler. Although each architecture has its own values that maximize the occupancy, both architectures have presented a similar behavior in our experimentation. We can observe from Table 2 that the optimal values for all kernels deployed on both architectures follows the guidelines described in Sect. 4. Thus, we conclude that the lessons learned described above can be applied to any Fermi and Kepler.

7 Conclusions

This paper shows how the combined use of different configuration and optimization techniques can significantly enhance the kernel performance of a GPU solution. When

applied to the our non-synthetic problem case study, the APSP problem, we obtained a global performance improvement up to 62 % compared with baseline configurations. We have shown how the kernel characterization technique was a useful procedure to predict the configuration parameter values for the threadblock size and the cache L1 state, leading to significant performance improvement in NVIDIA GPUs in the APSP problem. We have shown that the CUDA recommended values are not always the proper choice, and due to the big search space of possible combinations, we find these predictions and guidelines very helpful for non-expert CUDA programmers, or even for auto-tuning tools that aim to automatically configure the kernel execution for an optimal performance. Regarding to the concurrent kernel technique, the experimental results suggest that its use does not interfere with the predictions of the kernel characterization criteria. This technique results more profitable for kernels with few memory accesses.

Our future work includes to extend this experimentation for other non-synthetic applications with different kernel characterizations. Finally, an interesting work would be to adapt an analyzer already present scientfic community in order to automatically obtain the kernel characterization needed to apply our prediction values.

Acknowledgments This research has been partially supported by Ministerio de Economía y Competitividad (Spain) and ERDF program of the European Union: CAPAP-H4 network (TIN2011-15734-E), MOGECOPP project (TIN2011-25639); and Junta de Castilla y León (Spain) ATLAS project (VA172A12-2).

References

1. Barceló J, Codina E, Casas J, Ferrer JL, García D (2005) Microscopic traffic simulation: a tool for the design, analysis and evaluation of intelligent transport systems. J Intell Robot Syst 41:173–203
2. Cormen TH, Stein C, Rivest RL, Leiserson CE (2001) Introduction to algorithms, 2nd edn. McGraw-Hill Higher Education, Burr Ridge, Il 60521
3. Crauser A, Mehlhorn K, Meyer U, Sanders P (1998) A parallelization of Dijkstra's shortest path algorithm. In: Brim L, Gruska J, Zlatuška J (eds) Mathematical foundations of computer science 1998, LNCS, vol 1450. Springer, Berlin, pp 722–731
4. Dasgupta A (2011) CUDA performance analyzer. Ph.D. thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology
5. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1:269–271
6. Farooqui N, Kerr A, Diamos G, Yalamanchili S, Schwan K (2011) A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In: Proceedings of 4th workshop on GPGPU, GPGPU-4, x. ACM, New York, NY, pp 9:1–9:9
7. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. InPar 2012:1–10
8. Harris M (2008) Optimizing parallel reduction in CUDA. NVIDIA
9. Kirk DB, Hwu WW (2010) Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, San Francisco, CA, USA, p 258
10. Martín P, Torres R, Gavilanes A (2009) CUDA solutions for the SSSP problem. In: Allen G, Nabrzyski J, Seidel E, van Albada G, Dongarra J, Sloot P (eds) Computational science—ICCS 2009, LNCS, vol 5544. Springer, Berlin, pp 904–913
11. Nobari S, Lu X, Karras P, Bressan S (2011) Fast random graph generation. In: Proceedings of 14th international Conference on EDBT/ICDT '11. ACM, NY, pp 331–342
12. Ortega-Arranz H, Torres Y, Llanos DR., Gonzalez-Escribano A (2013) A new GPU-based approach to the shortest path problem. In: High performance computing and simulation (HPCS), 2013 international Conference on, pp 505–512

13. Rétvári G, Bíró JJ, Cinkler T (2007) On shortest path representation. IEEE ACM Trans Netw 15:1293–1306
14. Torres Y, González-Escribano A, Llanos DR (2012) uBench: performance impact of CUDA block geometry. In: Techniocal report IT-DI-2012-0001, Universidad de Valladolid
15. Torres Y, Gonzalez-Escribano A, Llanos DR (2013) uBench: exposing the impact of CUDA block geometry in terms of performance. J Supercomput 65:1–14
16. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76