Noname manuscript No.
(will be inserted by the editor)

Interval-based Performance Modeling for the All-Pairs-Shortest-Paths-Problem on GPUs

Jörg Dümmler · Sebastian Egerland

the date of receipt and acceptance should be inserted later

To cite this version:
Dümmler, J.; Egerland, S.: Interval-based Performance Modeling for the All-Pairs-Shortest-Paths-Problem on GPUs. In: The Journal of Supercomputing, Bd. 71, Nr. 11: S. 4192-4214. Springer – ISSN 0920-8542, 2015. DOI: 10.1007/s11227-015-1514-9

Abstract The article proposes a cost model for two implementations of the all-pairs-shortest-path (APSP) problem on graphics processing units (GPUs) with a particular focus on the correct reproduction of the shape of the runtime curve for different input problem sizes and thread block configurations. The model categorizes thread blocks according to the number of active warps and defines four different approaches to estimate the mapping of thread blocks to the hardware execution resources. Each of these approaches leads to a runtime prediction, which together span an interval for the expected execution time. An experimental evaluation shows that most characteristics of the runtime curve are correctly predicted, which is especially important for small graphs where a tiny increase of the input size may result in a significant increase of the execution time. For large graphs, we show that the cost prediction deviates less than 1% from the measured times in many cases.

Keywords Performance Modeling · GPGPU · Benchmarking · All-pairs-shortest-path

J. Dümmler
Chemnitz University of Technology, 09107 Chemnitz, Germany,
E-mail: djo@cs.tu-chemnitz.de

S. Egerland
Chemnitz University of Technology, 09107 Chemnitz, Germany,
E-mail: egse@hrz.tu-chemnitz.de

1 Introduction

Graphics Processing Units (GPUs) are increasingly used in the area of high performance computing because they offer a much higher parallel peak performance than current multi-core processors (CPUs) [2]. Since the hardware architecture of GPUs is substantially different from CPUs, applications need to be specifically adapted for an execution on GPUs to obtain the best possible performance. For the coding of GPU applications several high-level programming approaches have been developed including Nvidia CUDA [20] and OpenCL [12].

Performance prediction models for parallel applications are required for several reasons. First, a performance model allows the identification of hardware and software parameters that influence or limit application performance. Thus, it can be used as a guideline for code optimization for a specific target platform. Second, it allows performance assessment on future hardware and is, thus, useful in determining whether to perform a hardware upgrade. Third, it can be used to select the best program version if multiple implementation variants are available and, finally, it can provide the required input for the scheduling of task-based applications. Performance prediction for GPUs is a challenging task, since minor changes in the implementation or in the execution configuration can lead to significant performance differences. A variety of performance models have been proposed for GPUs in the past, see [16] for an overview. The vast majority of these models concentrates on the runtime prediction for a specific execution scenario, i.e., a fixed input problem size and a fixed execution configuration.

In the article, we propose a performance modeling approach for GPUs that particularly focuses on the prediction of the execution time for different input problem sizes and different kernel thread block configurations. The presented cost model is tailored to a GPU implementations of the all-pairs-shortest-path (APSP) problem based on the Min-plus algorithm [15] and the Floyd-Warshall algorithm, respectively. APSP is a fundamental problem of graph theory with many application areas including routing in communication networks and the calculation of transport routes. The general approach used in this article can easily be transferred to applications with a similar compute structure, i.e., applications operating on matrices in a block-based fashion, such as for example the multiplication of two matrices.

The performance measurements performed for the APSP implementation show that the execution time depends on the input problem size in an approximately piecewise linear way with jumps in-between the linear segments of the obtained runtime curve. The major goal of the proposed approach is to correctly predict the shape of the runtime curve, i.e., the location of the jumps and the slope of the linear segments. For this purpose, we combine analytical modeling with empirical data collected by benchmark runs. The analytical part delivers a cost prediction in symbolic time units that leads to a curve of appropriate shape if applied to different input problem sizes. The analytical model is parameterized with the hardware configuration of the GPU, such as the number of multiprocessors and the total number of cores. The empirical data is used to translate the symbolic time units in actual time units. For obtaining the empirical data, a single benchmark run for a predefined reference problem size is required. Based on the result of this benchmark run, performance predictions for different problem sizes are computed.

For an accurate performance prediction, the presented model examines the thread blocks defined by the algorithm in detail. In particular, different categories of thread blocks are identified that differ in the number of active warps. The mapping of thread blocks to GPU multiprocessors is modeled based on one out of four available approaches (*Min*, *Sorted*, *Full*, *Buckets*). Two of these approaches are based on optimistic and the other two on pessimistic assumptions. Overall, this leads to four different performance predictions that span in interval in which the actual execution time is expected.

The experimental evaluation has been performed on two different GPU architectures (Kepler and Fermi) using different thread block sizes as well as small and large input problems. The results show

that most of the jumps in the runtime curves are predicted correctly. In particular, the *Buckets* approach mimics the shape of the runtime curve closely. Especially for large graphs on the Fermi architecture, we have observed very good runtime predictions that deviate less than 1% from the measured execution times.

The article is structured as follows. Section 2 introduces the terms and concepts of the CUDA programming model that are relevant for the cost model. Section 3 defines the APSP problem and discusses an appropriate GPU implementation. Section 4 explains the proposed cost model in detail. Section 5 includes an experimental evaluation of the model. Section 6 discusses related work and Sect. 7 concludes.

2 CUDA Programming Model

Several programming approaches have been developed for the implementation of general purpose applications on GPUs including Nvidia CUDA [20] and OpenCL [12]. In this article, we concentrate on CUDA and use the corresponding notions, but in principle the proposed runtime model can also be used for an OpenCL implementation. A CUDA program consists of a host program that is executed by the CPU and a set of kernels each of which containing parallel code running on the GPU. The host program controls the execution of the kernels and initiates the required data transfers between the main memory (accessible by the CPU) and the global memory located on the GPU.

A GPU is a manycore platform consisting of a large number of streaming processors (SPs) that are organized in streaming multiprocessors (SMs). In the following, the number of SMs is denoted as $\#SMs_{Dev}$ and the number of SPs per SM is denoted as $\#CoresPerSM_{Dev}$. There are several types of memory on a GPU, including the global GPU memory that can be accessed by each SP, and the shared memory located on each SM.

A CUDA kernel is executed by a number of threads that are logically grouped into thread blocks. The number of threads and thread blocks is specified when invoking the kernel in the host program. Threads within the same thread block may cooperate with each other using synchronization operations or data located in the shared memory, but threads of different thread blocks are completely independent of each other.

The computations of a kernel are mapped to the hardware execution resources in multiple steps. First, entire thread blocks are assigned to the SMs where each SM can hold at most $\#BlocksPerSM_{Occ}$ thread blocks at any given point of time. The value $\#BlocksPerSM_{Occ}$ depends on the ratio of the required resources (number of threads, shared memory and registers) of a thread block and the available hardware resources [19]. In the second step, each thread block is subdivided into a number of warps consisting of $\#ThreadsPerWarp_{Dev}$ threads each. All threads of the same warp are executed in lock step, usually by different SPs of the same SM. Different warps of the same thread block may be executed concurrently (if there are enough SPs available) or one after another. In the latter case, the execution of warps often overlaps to hide latencies resulting from memory accesses.

The number of warps created for a given kernel has to be a multiple of the warp allocation granularity $WarpGranularity_{Dev}$. Depending on the number of execution threads specified in the host program, this hardware parameter may lead to the creation of additional (empty) warps when executing the kernel. For the definition of the cost model in Sect. 4, we use the function $\#WarpsPerBlock(n)_{Dev}$ which returns the number of warps for a thread block with n threads. If multiple threads of a given warp access neighboring elements in the global GPU memory, $\#CoalescableMemAccesses_{Dev}$ memory requests are fused into a single transaction. This operation is called memory coalescing. The value of

Table 1 Overview of the hardware-specific model parameters defined in Sect. 2.

Parameter	Description
$\#SMs_{Dev}$	Number of streaming multiprocessors (SMs)
$\#CoresPerSM_{Dev}$	Number of CUDA cores per SM
$\#BlocksPerSM_{Occ}$	Maximum number of active thread blocks on an SM
$\#ThreadsPerWarp_{Dev}$	Number of threads in a warp
$WarpGranularity_{Dev}$	Minimum number of warps allocated at the same time
$\#WarpsPerBlock(n)_{Dev}$	Number of warps created for a thread block with n threads
$\#CoalescableMemAccesses_{Dev}$	Number of memory requests fused into a single transaction

$\#CoalescableMemAccesses_{Dev}$ depends on the bus width of the GPU and the size of each data element accessed. The hardware parameters defined in this section are summarized in Tab. 1.

3 The APSP problem and its GPU implementation

In the following, we define the all-pairs-shortest-path (APSP) problem in Subsect. 3.1, discuss the Min-plus algorithm for the solution of this problem in Subsect. 3.2, and present a parallel CUDA implementation in Subsect. 3.3.

3.1 Problem statement

The input of the APSP problem is a directed graph $G = (V, E)$ with a set $V = \{v_1, \ldots, v_n\}$ of n nodes and a set $E \subseteq V \times V$ of directed edges. Each edge $e \in E$ is annotated with a weight $w(e) \in \mathbb{N}$. In the following, we assume that the weights are stored in a matrix $A \in \mathbb{R}^{n \times n}$ where an entry $A_{i,j}$, $i, j = 1, \ldots, n$ is defined as follows

$$A_{i,j} = \begin{cases} w(e) & \text{if } e = (v_i, v_j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

The APSP problem consists of the determination of the path with the minimum accumulated weight between each pair of nodes $u, v \in V$. For this problem, it is reasonable to assume that there are no cycles with a negative accumulated weight in G because traversing a negative weight cycle multiple times will lead to paths with an arbitrary low accumulated weight. Furthermore, we assume that the output is produced in form of two matrices $D, P \in \mathbb{R}^{n \times n}$ where an entry $D_{i,j}$ of the *distance matrix* D denotes the accumulated weight of the shortest path from v_i to v_j and an entry $P_{i,j}$ of the *path matrix* P denotes the index of the first node on the shortest path from v_i to v_j with $i, j = 1, \ldots, n$. The entire shortest path between two nodes can be reconstructed by a suitable traversal of matrix P.

3.2 The Min-plus algorithm

Several algorithms for the computation of the APSP problem have been proposed. Examples are the Floyd-Warshall algorithm with a complexity of $\mathcal{O}(n^3)$ and the Min-plus algorithm [15] with a complexity of $\mathcal{O}(n^3 \cdot \log(n))$. In the following, we focus on the Min-plus algorithm, which is especially suited for a GPU implementation due to its similarity with matrix multiplication.

Algorithm 1: Host Program for CUDA-APSP.

```
1 Init<<< GridWidth × GridWidth, BlockWidth × BlockWidth >>>(n, A, D, P);
2 for (step = 1, ..., ⌈log2(n − 1)⌉) do
3     APSP<<< GridWidth × GridWidth, BlockWidth × BlockWidth >>>(n, D, P);
```

The idea of the Min-plus algorithm is to compute a series of auxiliary matrices $D^{(2*m)}, m = 1, 2, \ldots$ where $D^{(k)}$ contains the length of the shortest paths between all pairs of nodes subject to the constraint that each path contains at most k edges. These matrices are computed by

$$\begin{aligned} D^{(1)} &= A \\ D_{i,j}^{(2*m)} &= \min_{k=1,\ldots,n} \{D_{i,k}^{(m)} + D_{k,j}^{(m)}\} \qquad (m \geq 1 \text{ and } i, j = 1, \ldots, n). \end{aligned}$$

The computation of $D^{(2*m)}$ can also be written as $D^{(m)} \otimes D^{(m)}$ where the operator $\otimes$ denotes a modified form of matrix multiplication with addition replacing multiplication and the minimum operation replacing the addition in the original matrix multiplication. Since each shortest path in G contains at most $n-1$ edges (assuming no negative weight cycles), the final distance matrix D equals $D^{(k)}$ where $k = 2^{\lceil \log_2(n-1) \rceil}$, i.e., $\lceil \log_2(n-1) \rceil$ steps are required to compute the final result. For the path matrix P, a similar sequence of auxiliary matrices can be computed by keeping track of the positions at which a new shortest path is discovered, i.e.,

$$\begin{aligned} P_{i,j}^{(1)} &= \begin{cases} j & \text{if } (v_i, v_j) \in E \\ \infty & \text{otherwise} \end{cases} \\ P_{i,j}^{(2*m)} &= P_{i,k_{min}}^{(m)} \qquad \text{with } k_{min} = \operatorname*{argmin}_{k=1,\ldots,n, k \neq i} \{D_{i,k}^{(m)} + D_{k,j}^{(m)}\} \\ &\qquad (m \geq 1 \text{ and } i, j = 1, \ldots, n). \end{aligned}$$

3.3 Parallel CUDA implementation

Several GPU implementations have been proposed for the APSP problem, see e.g. [3,6,24]. In the following, we present an implementation that exploits the similarity of the Min-Plus algorithm with matrix multiplication [24]. The CUDA implementation consists of a host program shown in Alg. 1 and two CUDA kernels called *Init* and *APSP*. The host program first invokes the embarrassingly parallel kernel *Init*, which initializes the matrices D and P with data from the input matrix A. Afterwards, the kernel *APSP* is executed $\lceil \log_2(n-1) \rceil$ times, where each kernel call applies the Min-plus operator $\otimes$ to matrix D and updates matrix P accordingly. The kernel works in-place, i.e., the updated values are written to the given input matrices immediately.

Kernel *APSP* is executed by two-dimensional thread blocks of size $BlockSize \times BlockSize$ where $BlockSize$ is a configurable parameter. The entire kernel grid consists of $GridWidth \times GridWidth$ thread blocks where $GridWidth = \lceil n/BlockSize \rceil$. Table 2 summarizes the application-specific parameters introduced in this section. The matrices D and P are partitioned into tiles of size $BlockSize \times BlockSize$ where each thread block is responsible for the updates for one of these tiles. The pseudo code of kernel *APSP* is shown in Alg. 2.

Algorithm 2: Pseudo code of kernel *APSP*.

```
1  let (gx, gy) be the global thread coordinates;
2  let (tx, ty) be the local thread coordinates;
3  wmin = D(gx, gy); kmin = ∞;
4  for (b = 0, ..., GridWidth − 1) do
5  |   collectively load 2 tiles of matrix D into shared memory locations D1 and D2;
6  |   synchronize();
7  |   km = argmin_{k=0,...,BlockSize−1}{D1(tx, k) + D2(k, ty)};
8  |   wk = D1(tx, km) + D2(km, ty);
9  |   if wk < wmin then wmin = wk; kmin = b ∗ BlockSize + km;
10 |   synchronize();
11 if (kmin ≠ ∞) then D(gx, gy) = wmin; P(gx, gy) = P(gx, kmin);
```

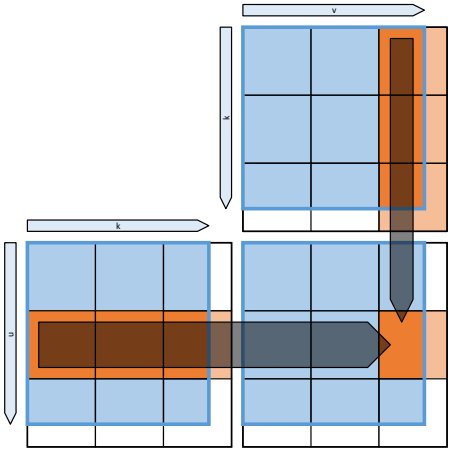


Fig. 1 Illustration of the tiles processed by a specific thread block. The bottom right matrix highlights the tile processed by the thread block. The top matrix and the bottom left matrix show the tiles that have to be loaded into shared memory for the computations of the output tile.

First, each thread determines its local and global coordinates within the thread block and the kernel grid, respectively (lines 1 and 2). Line 3 determines the current length *wmin* of the path for the assigned pair of nodes. Afterwards, the kernel iterates over the tiles in matrix D (line 4). In each step, one tile located on the same row and the corresponding tile located on the same column as the output tile of the thread block are loaded into shared memory using a coalesced access pattern (line 5), see Fig. 1 for an illustration. After synchronizing all threads from a thread block (line 6), each thread performs a Min-plus operation using data from shared memory (line 7-8). If applicable, the current shortest path *wmin* and the coordinate *kmin* of the minimum are updated (line 9). Finally, the global data structures are updated, if a new shortest path has been found (line 11).

Table 2 Overview of the application-specific model parameters defined in Sect. 3.

Parameter	Description
$BlockWidth$	Size of a tile in the computed matrices.
$GridWidth$	Number of tiles per dimension of the computed matrices.
n	Number of nodes in the input graph.

Algorithm 3: Host Program for CUDA-APSP-FW.

```
1 Init<<< GridWidth × GridWidth, BlockWidth × BlockWidth >>>(n, A, D, P);
2 for (k = 1, ..., n) do
3     APSP-FW<<< GridWidth × GridWidth, BlockWidth × BlockWidth >>>(n, k, D, P);
```

Algorithm 4: Pseudo code of kernel *APSP-FW*.

```
1 let (gx, gy) be the global thread coordinates;
2 let k be the step number provided by the host code;
3 if (D(gx, k) + D(k, gy) < D(gx, gy)) then
4     D(gx, gy) = D(gx, k) + D(k, gy);
5     P(gx, gy) = P(k, gy);
```

3.4 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an alternative approach to compute the APSP problem. In its sequential version, it consists of three nested loops each of which running over all nodes of the input graph leading to a runtime complexity of $\mathcal{O}(n^3)$. All data dependencies of the loop nest are carried by the outermost loop. Thus, in a parallel implementation of this algorithm, the iterations of the outermost loop have to be executed one after another, but the iterations of two innermost loops are pairwise independent of each other and can be executed concurrently.

The GPU implementation of the Floyd-Warshall algorithm follows this parallelization strategy. The host program shown in Alg. 3 is responsible for the initialization (similar to the Min-plus algorithm) and the sequential execution of the outermost loop. Each loop iteration corresponds to one invocation of the parallel kernel $APSP - FW$, which is shown in Alg. 4. The kernel is executed with $n \times n$ threads organized in thread blocks of size $BlockWidth \times BlockWidth$, see Tab. 2 for the parameters. Each thread is responsible for the update of one element of the distance and the path matrix.

In principle, a GPU implementation of the Floyd-Warshall algorithm can also employ shared memory to store elements of the distance matrix required by multiple threads of the kernel, see e.g. [24]. Our experiments on Fermi and Kepler GPUs have shown a slowdown caused by the additional memory copy operations. Thus, our implementation accesses all data directly from global memory.

4 Performance Model

This section proposes a performance prediction model for the parallel CUDA implementations of the APSP problem shown in Sect. 3. The approach used in this section is purely analytical, the translation of the obtained values to actual time units in milliseconds is discussed in Sect. 5. The model focuses on the update operations performed by the CUDA kernels *APSP* (see Alg. 2) and *APSP-FW* (see Alg. 4),

Table 3 Overview of the parameters and variables defined in Sect. 4.

Parameter	Description
T_{App}	Total predicted execution time, computed according to (1).
T_{APSP}	Predicted execution time of a single kernel execution, computed according to (2).
T_{Launch}	Overhead for launching a CUDA kernel (hardware parameter).
T_{Warp}	Predicted execution time of a single warp, computed according to (3).
T_{Instr}	Average execution time of an arithmetic operation (hardware parameter).
T_{Read}	Average execution time for a read memory access (hardware parameter).
T_{Write}	Average execution time for a write memory access (hardware parameter).
$\#InstrPerVertex$	Average number of instructions executed per graph node (hardware parameter).
$\#MemAccessRead$	Total number of read accesses per kernel launch, computed according to (4).
$\#MemAccessWrite$	Total number of write accesses per kernel launch, computed according to (5).
$\#WarpsPerSM$	Maximum number of active warps assigned to an SM, computed by either model (*Min*, *Sorted*, *Full*, or *Buckets*).
$\#Warps$	Total number of active warps per kernel launch, computed according to (6).
$\#Warps_{Full/Right/Bottom/Last}$	Total number of warps in full, right, bottom, or last thread blocks, computed by (8), (10), (12), and (13), respectively.
$\#WarpsPerBlock_{Full/Right/Bottom}$	Number of warps in a thread block of type full, right, or bottom, computed by (7), (9), and (11), respectively.
$\#WarpsPerSM_{Min/Full/Buckets}$	Maximum number of active warps per SM according to model *Min*, *Full*, or *Buckets*, computed by (14),(15), and (18), respectively.

respectively. The model ignores the data transfer operations to copy the input/output data to/from the device and the time required to initialize the output data structures (kernel *Init*). An overview of all symbols defined in this section is given in Tab. 3. We first present the performance model for the Min-plus algorithm in detail. Subsect. 4.5 shows the application of the model to the Floyd-Warshall implementation.

4.1 Modeling the application execution time

The total execution costs T_{App} for the entire Min-plus algorithm are computed by

$$T_{App} = \lceil \log_2(n-1) \rceil * T_{APSP} \tag{1}$$

where T_{APSP} is the time required to execute kernel *APSP* once. The kernel execution time is computed by

$$T_{APSP} = T_{Launch} + T_{Warp} * \#WarpsPerSM * \frac{\#ThreadsPerWarp_{Dev}}{\#CoresPerSM_{Dev}} \tag{2}$$

where T_{Launch} is the overhead to start a CUDA kernel, T_{Warp} is the time to execute a single warp (see Subsect. 4.2 for its calculation) and $\#WarpsPerSM$ is the maximum number of warps assigned to an SM (see Subsect. 4.3 and 4.4 for computing this value). The factor $\#ThreadsPerWarp_{Dev}/\#CoresPerSM_{Dev}$ determines how many of the $\#WarpsPerSM$ are executed one after another on each SM.

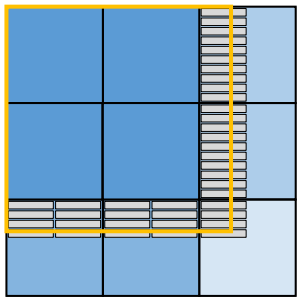


Fig. 2 Illustration of the relationship between the input matrix size (yellow frame) and the kernel grid created for the execution of kernel *APSP*. The blue squares symbolize thread blocks and the grey rectangles represent warps with active threads in thread blocks that partially overlap the input matrix.

4.2 Modeling the warp execution time

The time T_{Warp} denotes the time required for a single warp to execute Alg. 2, which is computed by

$$\begin{aligned} T_{Warp} =& n * \#InstrPerVertex * T_{Instr} + \\ & \#MemAccessRead * T_{Read} + \#MemAccessWrite * T_{Write}. \end{aligned} \tag{3}$$

The first line of equation (3) captures the time required to perform computations. This time depends on the number $\#InstrPerVertex$ of GPU instructions executed for each graph node in lines 7-9 of Alg. 2 and the average execution time of an instruction T_{Instr}. Both, $\#InstrPerVertex$ and T_{Instr} depend on the hardware specifications of the GPU, see also Sect. 5.

The time required to copy data from or to global memory depends on the number of read and write accesses per warp ($\#MemAccessRead$ and $\#MemAccessWrite$) and the hardware-specific time for a memory access (T_{Read} and T_{Write}). We only consider accesses to the global memory, which are all performed using a coalesced access pattern. The exact number of memory accesses depends on whether the matrices D and P have to be updated in line 11 of Alg. 2. In the following, we use the parameter $P_{min} \in [0, 1]$ to denote the probability of an update operation for a warp. Thus, each warp performs

$$\#MemAccessRead = \left\lceil \frac{(1 + 2 * GridWidth + P_{min}) * \#ThreadsPerWarp_{Dev}}{\#CoalescableMemAccesses_{Dev}} \right\rceil \tag{4}$$

read accesses (lines 3,5,11 of Alg. 2) and

$$\#MemAccessWrite = \left\lceil \frac{2 * P_{min} * \#ThreadsPerWarp_{Dev}}{\#CoalescableMemAccesses_{Dev}} \right\rceil \tag{5}$$

write accesses (line 11 of Alg. 2).

4.3 Computing the total number of warps

Figure 2 shows the kernel grid for kernel *APSP* where the blue squares represent thread blocks of size $BlockSize \times BlockSize$ and the yellow frame represents the input matrix of size $n \times n$. If n is not a

multiple of $BlockSize$, some threads have coordinates outside of the matrix and, thus, do not participate in computations. As a consequence, some warps are immediately terminated by the GPU. For an accurate modeling of the execution time especially for small graphs, we have to take this behavior into account. Thus, we distinguish four different types of thread blocks:

- Full thread blocks where all threads perform computations (dark blue squares in Fig. 2).
- Right thread blocks located on the rightmost column of the kernel grid. In these blocks, only a subset of the warps on each row perform computations.
- Bottom thread blocks located on the bottom row of the kernel grid. In these blocks, only a subset of the rows performs computations and some rows have no assigned work.
- The last thread block located on the bottom right of the kernel grid. In this block, only a subset of the warps on some of the rows perform computations.

The total number of warps is the sum of the warps of each category, i.e.,

$$\#Warps = \#Warps_{Full} + \#Warps_{Right} + \#Warps_{Bottom} + \#Warps_{Last}. \tag{6}$$

In the following, we give formulas to compute the number of warps of each category assuming that each row of a thread block starts with a new warp.

4.3.1 Computing the number of warps in full thread blocks

In full thread blocks all included threads are active. Thus, a single thread block of type full includes

$$\#WarpsPerBlock_{Full} = \#WarpsPerBlock_{Dev}\left(\left\lceil \frac{BlockSize}{\#ThreadsPerWarp_{Dev}} \right\rceil * BlockSize\right) \tag{7}$$

warps. The total number of active warps in full thread blocks is then computed by

$$\#Warps_{Full} = \left\lfloor \frac{n}{BlockSize} \right\rfloor^2 * \#WarpsPerBlock_{Full}. \tag{8}$$

4.3.2 Computing the number of warps in right thread blocks

In a right thread block, (n mod $BlockSize$) threads on each of the $BlockSize$ rows perform work leading to

$$\#WarpsPerBlock_{Right} = \#WarpsPerBlock_{Dev}\left(\left\lceil \frac{n \bmod BlockSize}{\#ThreadsPerWarp_{Dev}} \right\rceil * BlockSize\right) \tag{9}$$

active warps per thread block and

$$\#Warps_{Right} = \left(\left\lceil \frac{n}{BlockSize} \right\rceil - 1\right) * \#WarpsPerBlock_{Right} \tag{10}$$

total active warps in right thread blocks.

4.3.3 Computing the number of warps in bottom thread blocks

Each bottom thread block has (n mod $BlockSize$) rows each of which containing $BlockSize$ active threads. Thus, the number of warps in such a thread block is

$$\#WarpsPerBlock_{Bottom} = \#WarpsPerBlock_{Dev} \left(\left\lceil \frac{BlockSize}{\#ThreadsPerWarp_{Dev}} \right\rceil * (n \bmod BlockSize) \right) \tag{11}$$

and the total number of warps in all bottom thread blocks is

$$\#Warps_{Bottom} = \left(\left\lceil \frac{n}{BlockSize} \right\rceil - 1 \right) * \#WarpsPerBlock_{Bottom}. \tag{12}$$

4.3.4 Computing the number of warps in the last thread block

The last thread block contains (n mod $BlockSize$) rows with (n mod $BlockSize$) active threads each. Thus, the total number of warps in this thread block is

$$\#Warps_{Last} = \#WarpsPerBlock_{Dev} \left(\left\lceil \frac{n \bmod BlockSize}{\#ThreadsPerWarp_{Dev}} \right\rceil * (n \bmod BlockSize) \right). \tag{13}$$

4.4 Modeling the number of warps per SM

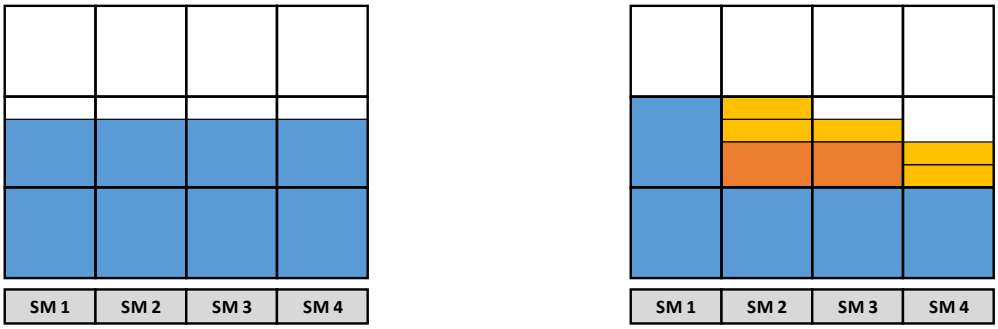
The execution time of kernel *APSP* depends on the assignment of the $\#Warps$ active warps to the SMs of the GPU. Since this mapping decision is made by the internal scheduler of the GPU and not visible for the application program, we consider 4 different approaches to estimate the maximum number of warps per SM ($\#WarpsPerSM$).

4.4.1 Model Min

The model *Min* assumes that warps are assigned to SMs in a round robin fashion, i.e., the maximum number of warps per SM is given by

$$\#WarpsPerSM_{Min} = \left\lceil \frac{\#Warps}{\#SMs_{Dev}} \right\rceil. \tag{14}$$

This model ignores that warps of the same thread block have to executed on the same SM. Furthermore, it assumes a perfect load balance between the SMs. As a consequence, the computed value $\#WarpsPerSM_{Min}$ is an optimistic approximation of the actual number of warps executed by an SM. Figure 3 (a) illustrates the workload distribution for 4 SMs and 7 thread blocks according to model *Min*.



(a) Model *Min*. (b) Model *Sorted*.

Fig. 3 (a) Assumed distribution of 7 full thread blocks onto 4 SMs according to model *Min*. Each SM gets approximately the same number of warps possibly splitting thread blocks between SMs. (b) Assumed distribution of 4 full thread blocks, 2 half-full thread blocks and 5 quarter-full thread blocks onto 4 SMs according to model *Sorted*. The thread blocks are assigned to SMs by a list-scheduling algorithm that considers the thread blocks in order of decreasing number of active warps.

4.4.2 Model Sorted

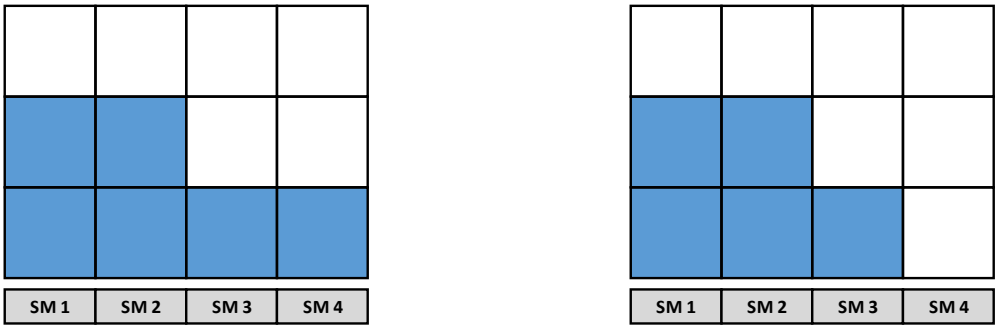
Model *Sorted* assumes that thread blocks are assigned to SMs by a static list-scheduling approach, i.e., the thread blocks are considered one after another starting with the block with the most active warps and each block is assigned to the SM currently having the smallest accumulated number of active warps. This leads to an optimistic estimate of the actual number of warps per SM, since the GPU hardware does not know the number of active warps before actually executing the corresponding thread block. Thus, a less optimal block distribution results in practice. Figure 3 (b) illustrates the mapping for 3 different types of thread blocks onto 4 SMs. The formula to compute $\#WarpsPerSM_{Sorted}$ is quite lengthy and thus omitted for the sake of a clear presentation.

4.4.3 Model Full

Model *Full* assumes that all created thread blocks are of type full, i.e., all created threads are assumed to be active. The assignment of thread blocks to SMs is performed in a round robin fashion. Thus, the maximum number of warps of an SM is computed by

$$\#WarpsPerSM_{Full} = \left\lceil \frac{\lceil \frac{n}{BlockSize} \rceil^2}{\#SMs_{Dev}} \right\rceil * \#WarpsPerBlock_{Full}. \tag{15}$$

An illustration for 6 thread blocks and 4 SMs is given in Fig. 4 (a). Model *Full* overestimates the actual workload leading to a pessimistic cost prediction especially for small input problem sizes where a large fraction of the warps may be inactive. But for large input problem sizes, i.e. $n \gg BlockSize$, the vast majority of blocks are of type full and the model is expected to give a realistic estimate.



(a) Model *Full*. (b) Model *Buckets*.

Fig. 4 (a) Assumed distribution of 6 thread blocks onto 4 SMs according to model *Full*. All thread blocks contain the maximum number of warps and the mapping of thread blocks onto SMs is performed by a round robin algorithm. (b) Assumed distribution of 5 thread blocks onto 4 SMs according to model *Buckets* using a bucket size of 2. The resulting buckets with at most 2 thread blocks are scheduled in a round robin way.

4.4.4 Model Buckets

Model *Buckets* assumes that multiple thread blocks are combined into a larger entity called bucket and that in each scheduling step an entire bucket of thread blocks is assigned to an SM. Similar to model *Full*, *Buckets* assumes that all thread blocks are of type full and that the resulting buckets assigned to SMs in a round robin fashion. The rationale behind the model is that the GPU scheduler may decide to assign an additional thread block to an SM when all currently assigned warps are blocked, e.g. when waiting for a data transfer from global memory. The size of a bucket is set to $\#BlocksPerSM_{Occ}$, i.e., the maximum number of thread blocks that may be active on an SM at the same time. An illustration of the workload distribution for bucket size 2 and 5 thread blocks is given in Fig. 4 (b). The maximum number of warps in an SM for this model is computed by

$$\#Blocks_1 = \left\lceil \frac{\lceil n/BlockSize \rceil^2}{\#BlocksPerSM_{Occ} * \#SMs_{Dev}} \right\rceil * \#BlocksPerSM_{Occ} \tag{16}$$

$$\#Blocks_2 = \min\left\{ \lceil n/BlockSize \rceil^2 - \#Blocks_1 * \#SMs_{Dev}, \#BlocksPerSM_{Occ} \right\} \tag{17}$$

$$\#WarpsPerSM_{Bucket} = (\#Blocks_1 + \#Blocks_2) * \#WarpsPerBlock_{Full} \tag{18}$$

4.5 Performance Model for the Floyd-Warshall algorithm

In the following, we discuss the application of the performance model developed for the Min-plus algorithm to the Floyd-Warshall algorithm. Several minor changes are required. The total execution time T_{App-FW} of the algorithm is computed by

$$T_{App-FW} = n * T_{APSP-FW} \tag{19}$$

where the kernel execution time

$$T_{APSP-FW} = T_{Launch} + T_{Warp-FW} * \#WarpsPerSM * \frac{\#ThreadsPerWarp_{Dev}}{\#CoresPerSM_{Dev}} \tag{20}$$

is computed similar to the Min-plus algorithm, see Eq. (2). The warp execution time $T_{Warp-FW}$ is computed by

$$\begin{aligned} T_{Warp-FW} =& \#InstrPerVertex - FW * T_{Instr} + \\ & \#MemAccessRead - FW * T_{Read} + \#MemAccessWrite - FW * T_{Write}. \end{aligned} \tag{21}$$

where the parameters

$$\#InstrPerVertex - FW = 2 \tag{22}$$

$$\#MemAccessRead - FW = \lceil \frac{(2 + P_{min}) * \#ThreadsPerWarp_{Dev}}{\#CoalescableMemAccesses_{Dev}} \rceil \tag{23}$$

$$\#MemAccessWrite - FW = \lceil \frac{2 * P_{min} * \#ThreadsPerWarp_{Dev}}{\#CoalescableMemAccesses_{Dev}} \rceil \tag{24}$$

$$\tag{25}$$

come from Alg. 4. The number of warps per SM (parameter $\#WarpsPerSM$) of the Floyd-Warshall algorithm is identical to the Min-plus algorithm, since both implementations employ an identical thread block grid.

5 Experimental Evaluation

The proposed performance prediction model is evaluated using two Nvidia GPUs: a Tesla C2075 (Fermi architecture) and a GTX 780 (Kepler architecture). The hardware parameters are shown in Tab. 4. For the remaining model parameters, i.e., the memory access and arithmetic computation times, we do not use the hardware parameters directly, since the cost model does not account for the overlapping of computation and memory accesses within an SM. Instead, we use a two step approach. In the first step, we use symbolic platform-independent values for these parameters that reflect their relative magnitude to each other, see Tab. 5 for the values used in this article. The result is a runtime prediction in symbolic time units. In the second step, we use the measured execution time on a specific GPU using a fixed input problem size n_0 to determine a platform-specific adjustment factor f, which is computed by

$$f = \frac{4 * T_{meas}(n_0)}{T'_{Min}(n_0) + T'_{Sorted}(n_0) + T'_{Full}(n_0) + T'_{Buckets}(n_0)}$$

where T_{meas} is the measured execution time and $T'_{Min}, T'_{Sorted}, T'_{Full}, T'_{Buckets}$ are the runtime predictions in symbolic units provided by the 4 models presented in the previous section. The final cost value in actual time units is then computed by multiplying the symbolic time with factor f, e.g. by

$$T_{Min}(n) = f \cdot T'_{Min}(n)$$

for model *Min* and an arbitrary $n \in \mathbb{N}$. For the results presented in the following, we use $n_0 = 500$ and consider small ($n \in [50, 100]$) and large synthetic graphs ($n \in [4000, 4050]$).

Table 4 Hardware parameters of the GPUs used in the benchmark test.

Parameter	C2075	GTX 780
$\#SMs_{Dev}$	14	12
$\#CoresPerSM_{Dev}$	32	192
$\#ThreadsPerWarp_{Dev}$	32	32
$WarpGranularity_{Dev}$	2	4
$\#CoalescableMemAccesses_{Dev}$	4	4

Table 5 Model parameters used in the experimental evaluation.

Parameter	Value
T_{Launch}	5000
$\#InstrPerVertex$	1000
T_{Instr}	10
T_{Read}	2000
T_{Write}	4000
P_{Min}	0.5

5.1 Experiments on the Tesla C2075

The experiments on the Tesla C2075 have been performed using thread blocks of size 8×8, 16×16, and 32×32, i.e., $BlockSize \in \{8, 16, 32\}$. Figure 5 shows the measured and predicted execution times for $BlockSize = 8$. Using this configuration, each SM can hold 8 blocks at the same time, i.e., $\#BlocksPerSM_{Occ} = 8$. As a consequence, the entire GPU with 14 SMs has a maximum of 112 active blocks. The measured execution times for small graphs show two major jumps: at $n = 65$ and at $n = 80$. The first jump results from an additional execution of kernel *APSP* (see line 2 of Alg. 1). The second jump is caused by an increase of the number of thread blocks from 100 (10×10 thread blocks for $n = 80$) to 121 (11×11 thread blocks for $n = 81$), which includes the maximum number of active blocks, i.e., some blocks have to wait for other blocks to finish their execution. Model *Buckets* (using bucket size 8) correctly predicts both jumps and provides a very good approximation of the observed execution behavior. The other models predict minor jumps with a step size of 8 (equal to $BlockSize$), but fail to account for the major jump at $n = 80$. For large graphs, we only show the results of the models *Min* and *Bucket*, because they represent the minimum and the maximum of the prediction interval, respectively. Both models underestimate the measured execution time by less than 1% on average. This may be caused by a factor f that is slightly too small.

The results for 16×16 thread blocks are shown in Fig. 6. The measured execution times for small graphs show jumps at $n = 65$ and $n = 96$, which originate in an increase of the number of kernel executions and in an increase in the number of thread blocks from 36 (6×6 thread blocks for $n = 96$) to 49 (7×7 thread blocks for $n = 97$), respectively. All models correctly predict the jump at $n = 65$, but only model *Min* foresees the jump at $n = 96$. For large graphs, we obtain a very good prediction of the actual execution time by all models. The maximum observed deviation is below 1%.

Using thread blocks of size 32×32 (see Fig. 7 for the results) delivers identical prediction for models *Full* and *Buckets*, since a bucket size of 1 is used for the latter ($\#BlocksPerSM_{Occ} = 1$). For small graphs, the runtime predictions are very accurate for all models (except *Min*) up to $n = 96$. The jump at $n = 96$ results from an increase of the number of thread blocks from 9 (3×3 thread blocks for $n = 96$) to 16 (4×4 thread blocks for $n = 97$). This jump is predicted by all models except *Sorted*. Model *Min*

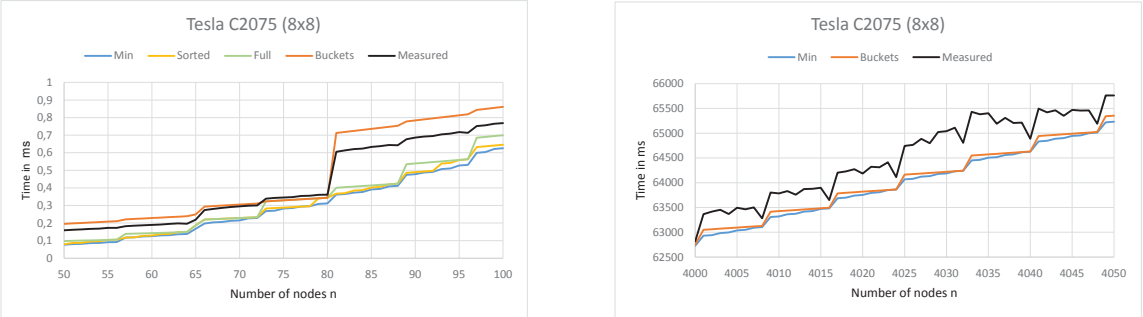


Fig. 5 Comparison of the measured and predicted execution times for the Tesla C2075 using thread blocks of size 8×8 for small graphs (left) and large graphs (right).

Fig. 6 Comparison of the measured and predicted execution times for the Tesla C2075 using thread blocks of size 16×16 for small graphs (left) and large graphs (right).

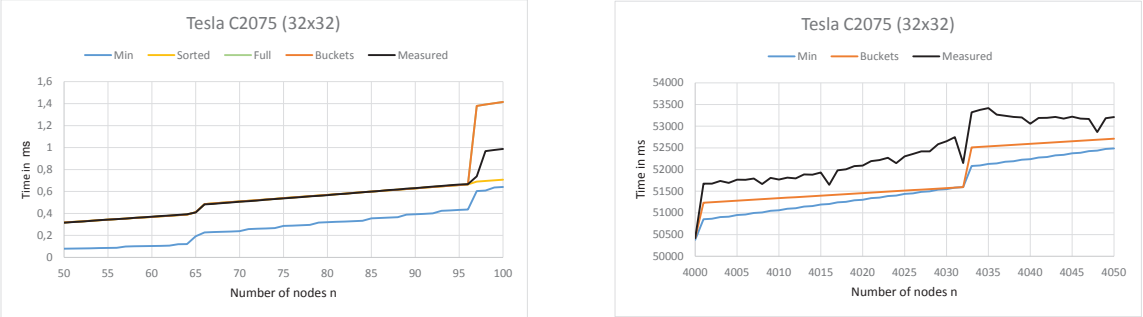


Fig. 7 Comparison of the measured and predicted execution times for the Tesla C2075 using thread blocks of size 32×32 for small graphs (left) and large graphs (right).

leads to the best prediction of the height of this jump. For large graphs, we observe a deviation of up to 2.5% from the measured runtime. The average deviation is between 1.1% (models *Full* and *Buckets*) and 1.6% (models *Min* and *Sorted*).

In summary, the results for small graphs show that the measured execution times are always within the interval spanned by the 4 proposed models. There are several jumps in the execution times that are predicted by some but not all models. Further investigations are required to analyze these jumps and to

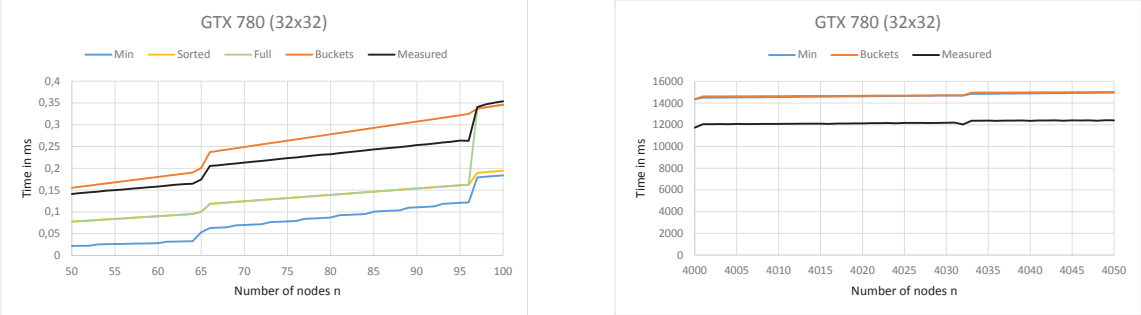


Fig. 8 Comparison of the measured and predicted execution times for the GTX 780 using thread blocks of size 32×32 for small graphs (left) and large graphs (right).

incorporate the observations into the prediction models. Considering the large graphs, all models slightly underestimate the measured runtime. Nevertheless, the runtime predictions have a very high accuracy with a deviation below 1% in most cases.

5.2 Experiments on the GTX 780

For the GTX 780, we only show the results for thread blocks of size 32×32 (see Fig. 8); thread blocks of size 8×8 and 16×16 lead to similar observations. For small graphs, the measured execution times exhibit two jumps (at $n = 65$ and at $n = 96$) that are predicted by all 4 models. The origin of these jumps is similar to the 32×32 scenario for the C2075. For $n \in [96, 100]$, models *Full* and *Buckets* predict the execution time almost perfectly. For large graphs, all models overestimate the execution time by about 20%. Nevertheless, the curve progression is modeled quite accurately, indicating that a smaller conversion factor f is required. Thus, future investigations may include the use of different values for f for different graph sizes to increase prediction accuracy.

6 Related Work

There are three major approaches to performance modeling for GPUs[16]:

- analytical models, which deliver a set of equations that describes the performance of an application depending on hardware and software parameters;
- machine learning approaches, which uses empirical data to train a model or a classifier;
- simulator-based approaches that simulate the target system step by step and extract performance data on the fly.

A further classification can be made into application-specific approaches that are designed for a single class of applications and general-purpose approaches that do not make any assumptions on a specific application structure. The model presented in this article is an application-specific analytical model whose final parameters are determined by empirical data. In the following, we give a brief overview of analytical and empirical models proposed by other research groups.

$MWP{-}CWP$ [8] is an analytical general-purpose model based on two metrics: MWP (memory warp parallelism) representing the maximum number of warps per SM that can overlap memory accesses and

CWP (computation warp parallelism) representing the number of warps that can perform their computations during one memory waiting period. These metrics determine whether a kernel is compute-bound, memory-bound, or does not offer enough parallelism. Depending on this classification, an appropriate formula to predict the kernel execution time is derived. The $MWP - CWP$ model has been used as basis for several other performance prediction frameworks. An example is GROPHECY[18], that obtains a suitable implementation variant for a GPU kernel based on an abstract code skeleton, which defines the computations and memory accesses performed by the kernel. Several extensions to the $MWP - CWP$ model, e.g., the inclusion of caches and special function units, have been presented in [22].

The MAX and SUM models[13] are general-purpose analytical models that use ideas of the models BSP and QRQW, the latter being an extension of the PRAM model. The total execution cost is either predicted as the maximum (MAX model) or the sum (SUM model) of the predicted computation and memory access costs. [1] presents a compiler-based approach, which constructs a workflow graph to represent the dependencies between memory accesses of a given kernel. The weight of the graph edges depends on the number of compute operations and on an adjustment for branch divergence. The total computation and memory latency of a warp is obtained by a suitable graph reduction. The approach presented in [9] is similar in spirit to the $MWP{-}CWP$ model, but with a focus on kernel throughput. The memory accesses are modeled in form of a pipeline. The execution time of a kernel is predicted depending on its classification as memory-bound or compute-bound. An application-independent analytical model for compute-intensive tasks has been proposed in [7]. This model ignores memory traffic and all thread blocks are assumed to have an identical workload that is evenly distributed over the available SMs.

The approach described in [17] combines an analytical model with a calibrated application-specific part, which provides information about, for example, cache utilization and thread-block layout. The model coefficients are fitted using measured execution times. The focus is on the correct prediction of the influence of the individual model variables. As an example application, an APSP implementation similar to the one in this article is used. In contrast to the proposed model, different input problem sizes are not taken into account and only full thread blocks are considered.

Application-specific performance models for GPUs have been proposed for sparse matrix-vector multiplication [5] and matrix-matrix multiplication [14]. The approach in [5] uses profiling information obtained for a set of benchmark matrices to compute a conversion factor to translate execution times from a reference architecture to an architecture with unknown execution times. The approach in [14] works on the warp level, i.e., thread block configurations are not considered. It computes a performance upper-bound by making optimistic assumptions, such as the absence of bank conflicts, and using microbenchmarks for different machine instructions occurring in the considered code.

Examples for models using empirical data are [11,21,4,10]. The approach discussed in [11] uses the Ocelot framework to collect various parameters like the number of instructions and memory transfers from the PTX code of a given kernel. Statistic analysis is used to group the parameters into principle components. The performance model is obtained by linear regression applied to the identified components and benchmark results obtained from a training set. [21] presents a history-based model for OpenCL kernels. The framework collects profiling data for kernel launches with different input data and execution configurations. The obtained results are used to identify parameters that correlate with the execution time. The strongly correlating parameters are used to build a linear prediction model. [4] presents an approach for cross-platform performance prediction that uses various profiling information for a given kernel that may be gathered by measurements or by using a simulator like GPGPU-Sim. Using the obtained information, the distance in the multi-dimensional parameter space between the considered kernel and different predefined training kernels is computed. The known performance of the training kernels is used to predict the execution time of the considered kernel on different hardware. [10] uses

a multiple regression model to predict the execution time of OpenCL kernels with the goal to identify bottlenecks. The model is built using profiling data from actual kernel executions, which are grouped into principal components. The above mentioned approaches collect many different parameters from program runs and may require the repeated execution of kernels with different execution configurations. In contrast, the approach presented in this article only requires the measured execution time for a single input problem size.

A quantitative performance model for the Nvidia GeForce 200 series has been proposed in [25]. The model combines information about the executed instructions obtained by the Barra GPU simulator with the results of microbenchmarks for the instruction pipeline, the shared memory access time, and the global memory access time. The goal is to optimize real-world applications and to suggest architectural improvements. [23] presents a set of microbenchmarks that measure the execution time depending on the memory access pattern and the amount of computation performed. The goal is to provide a guideline on how to choose an appropriate thread block size. For this purpose, the impact of different thread block configurations on the resulting performance is evaluated.

In contrast to the approach proposed in this article, none of the models listed above distinguishes explicitly between active and inactive warps and none of these models focuses on the correct reproduction of the curve shape obtained for different input sizes.

7 Conclusions

In this article, we have presented a cost model that provides four different approximations for the execution time of an all-pairs-shortest-path implementation on GPUs. Two of these approximations are based on optimistic assumptions. The model *Min* assumes that warps of a thread block can be scheduled independently and model *Sorted* assumes that the GPU hardware knows the number of active warps of a thread block before actually executing it. The other two approximations are pessimistic by ignoring any inactive threads that may exist if the kernel block size is not a multiple of the input graph size. Experiments show that we get a high prediction accuracy especially on the Fermi architecture. The underlying approach of the presented cost model may prove useful for predicting execution time of applications with a similar compute structure, such as for example the multiplication of two matrices.

References

1. Baghsorkhi, S., Delahaye, M., Patel, S., Gropp, W., m.W. Hwu, W.: An Adaptive Performance Modeling Tool for GPU Architectures. SIGPLAN Not. **45**(5), 105–114 (2010).
2. Brodtkorb, A., Hagen, T., Sætra, M.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing **73**(1), 4–13 (2013).
3. Buluç, A., Gilbert, J., Budak, C.: Solving Path Problems on the GPU. Parallel Comput. **36**(5-6), 241–253 (2010).
4. Che, S., Skadron, K.: BenchFriend: Correlating the performance of GPU benchmarks. Int. Journal of High Performance Computing Applications **28**(2), 238–250 (2014).
5. Guo, P., Wang, L.: Accurate cross–architecture performance modeling for sparse matrix–vector multiplication (SpMV) on GPUs. Concurrency and Computation: Practice and Experience (2014).
6. Harish, P., Narayanan, P.: Accelerating Large Graph Algorithms on the GPU Using CUDA. In: Proc. of the 14th Int. Conf. on High Performance Computing (HiPC '07), pp. 197–208. Springer-Verlag, Berlin, Heidelberg (2007).
7. Hasan, K., Chatterjee, A., Radhakrishnan, S., Antonio, J.: Performance Prediction Model and Analysis for Compute-Intensive Tasks on GPUs. In: C.H. Hsu, X. Shi, V. Salapura (eds.) Proc. of the 11th IFIP Int. Conf. on Network and Parallel Computing (NPC'14), *Lecture Notes in Computer Science*, vol. 8707, pp. 612–617. Springer Berlin Heidelberg (2014).

8. Hong, S., Kim, H.: An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. SIGARCH Comput. Archit. News **37**(3), 152–163 (2009).
9. Hu, Z., Liu, G., Dong, W.: A Throughput-Aware Analytical Performance Model for GPU Applications. In: Proc. of the 10th Annual Conf. Advanced Computer Architecture (ACA '14), pp. 98–112. Springer Berlin Heidelberg (2014).
10. Karami, A., Mirsoleimani, S., Khunjush, F.: A Statistical Performance Prediction Model for OpenCL Kernels on NVIDIA GPUs. In: Proc. of the 17th CSI Int. Symp. on Computer Architecture and Digital Systems (CADS '13), pp. 15–22. IEEE (2013).
11. Kerr, A., Diamos, G., Yalamanchili, S.: Modeling GPU-CPU Workloads and Systems. In: Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10), pp. 31–42. ACM, New York, NY, USA (2010).
12. Khronos OpenCL Working Group: The OpenCL Specification, Version 2.0 (2013). `http://www.khronos.org/opencl`
13. Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P., Srinathan, K.: A Performance Prediction Model for the CUDA GPGPU Platform. In: Proc. of the 2009 Int. Conf. on High Performance Computing (HiPC '09), pp. 463–472. IEEE (2009).
14. Lai, J., Seznec, A.: Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. In: Proc. of the 2013 IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO '13), pp. 1–10. IEEE Computer Society, Washington, DC, USA (2013).
15. Lawler, E.: Combinatorial Optimization: Networks and Matroids. Holt, Rinehart and Winston (1976).
16. Lopez-Novoa, U., Mendiburu, A., Miguel-Alonso, J.: A Survey of Performance Modeling and Simulation Techniques for Accelerator-based Computing. IEEE Transactions on Parallel and Distributed Systems **PP**(99), 1–1 (2014).
17. Ma, L., Chamberlain, R., Agrawal, K.: Performance Modeling for Highly-threaded Many-core GPUs. In: Proc. of the 25th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP '14), pp. 84–91. IEEE (2014).
18. Meng, J., Morozov, V., Kumaran, K., Vishwanath, V., Uram, T.: GROPHECY: GPU Performance Projection from CPU Code Skeletons. In: Proc. of 2011 Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '11), pp. 14:1–14:11. ACM, New York, NY, USA (2011).
19. Nvidia: CUDA Occupancy calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`
20. Nvidia: CUDA Toolkit Documentation, Version 6.5 (2014). `http://docs.nvidia.com/cuda`
21. Sato, K., Komatsu, K., Takizawa, H., Kobayashi, H.: A History-Based Performance Prediction Model with Profile Data Classification for Automatic Task Allocation in Heterogeneous Computing Systems. In: Proc. of the 9th Int. Symp. on Parallel and Distributed Processing with Applications (ISPA '11), pp. 135–142. IEEE Computer Society, Washington, DC, USA (2011).
22. Sim, J., Dasgupta, A., Kim, H., Vuduc, R.: A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. SIGPLAN Not. **47**(8), 11–22 (2012).
23. Torres, Y., Gonzalez-Escribano, A., Llanos, D.: uBench: Exposing the Impact of CUDA Block Geometry in Terms of Performance. J. Supercomput. **65**(3), 1150–1163 (2013).
24. Tran, Q.N.: Designing Efficient Many-Core Parallel Algorithms for All-Pairs Shortest-Paths Using CUDA. In: Proc. of the 7th Int. Conf. on Information Technology: New Generations (ITNG '10), pp. 7–12. IEEE Computer Society, Washington, DC, USA (2010).
25. Zhang, Y., Owens, J.: A Quantitative Performance Analysis Model for GPU Architectures. In: Proc. of the 17th IEEE Int. Symp. on High Performance Computer Architecture (HPCA '11), pp. 382–393. IEEE Computer Society, Washington, DC, USA (2011).