# GPU computations and memory access model based on Petri nets*

Anna Gogolińska, Łukasz Mikulski, and Marcin Piątkowski

Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University, Toruń, Poland
{anna.gogolinska,lukasz.mikulski,marcin.piatkowski}@mat.umk.pl

**Abstract.** In modern systems CPUs as well as GPUs are equipped with multi-level memory architectures, where different levels of the hierarchy vary in latency and capacity. Therefore, various memory access models were studied. Such a model can be seen as an interface abstracting the user from the physical architecture details. In this paper we present a general and uniform GPU computation and memory access model based on bounded inhibitor Petri nets (PNs). Its effectiveness is demonstrated by comparing its throughputs to practical computational experiments performed with the usage of Nvidia GPU with CUDA architecture.
Our PN model is consistent with the workflow of multithreaded GPU streaming multiprocessors. It models a selection and execution of instructions for each warp. The three types of instructions included in the model are: the arithmetic operation, the access to the shared memory and the access to the global memory. For a given algorithm the model allows to check how efficient the parallelization is, and whether a different organization of threads will improve performance.
The accuracy of our model was tested with different kernels. As the preliminary experiments we used the matrix multiplication program and stability example created by Nvidia, and as the main experiment a binary version of the least significant digit radix sort algorithm. We created three implementations of the algorithm using CUDA architecture, differing in the usage of shared and global memory as well as organization of calculations. For each implementation the PN model was used and the results of experiments are presented in the work.

**Keywords:** Petri nets, CUDA architecture, GPU, memory model

## 1 Introduction

The inter-process communication over a common part of the memory shared by processes is a usual performance bottleneck in multiprocessor environments. In modern systems CPUs as well as GPUs are equipped with multi-level memory architectures, where different levels of the hierarchy vary in latency and capacity.

* This research has been supported by the Polish National Science Center through grant No.2013/09/D/ST6/03928

By considering different local views of the processes on the common part of the memory one can try to improve the processor utilization. Therefore, various memory access models were studied, see for instance [8, 11, 15]. Such a model can be seen as an interface abstracting the user from the physical architecture details. It allows to specify, without a reference to processors, the local views that are possible in concurrent task executions and maintain its consistency.

Another important issue is the task distribution between threads and CPU/GPU cores and the instruction scheduling, which can have a significant impact on the efficiency. Consider for instance running the three threads on a single processor depicted in Figure 1. Each of them performs a list of arithmetic operations interleaved by memory reads/writes consisting of a short preprocessing and then a longer period of waiting for the memory access (which is a usual situation in parallel computing). In the initial part of the computation each thread realizes its preprocessing for the memory access and then starts to wait for the access itself. This causes the processor idle period when all threads are waiting (marked as I). After that the arithmetic operations of all threads are executed simultaneously. They can be scheduled in such a way that one thread waits for the memory access while other threads perform their computations and there is no idle period (marked as II). Thanks to that the waiting period of a thread can be hidden behind the active computations of other threads.

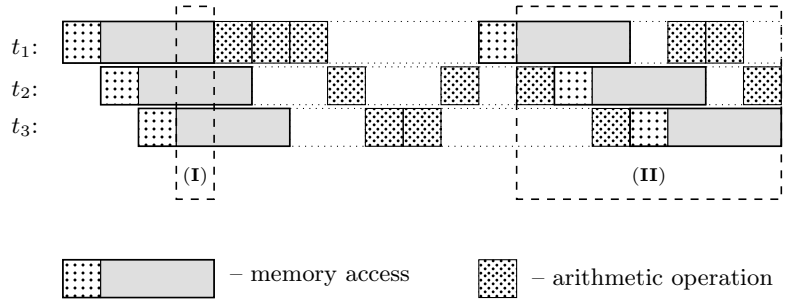


**Fig. 1.** The example run of the threads $t_1$, $t_2$ and $t_3$ on a single processor. The area marked with (I) represents the idle period when no computation is done, and the area marked with (II) represents the period when the waiting for the memory access is hidden behind arithmetic operations.

The main contribution of this paper is a general and uniform GPU computation and memory access model based on bounded Petri nets [14] together with the application simulating its execution. For a given algorithm pseudocode (or the source code) one can use our model to count the number of operations performed (taking into account their simultaneous execution). The main advantage of the model over the basic arithmetic counting and other models described below is the possibility of reorganization of parts of the code, the potential ability to predict a duration of GPU calculations and to handle the aforementioned

computation scheduling. Such an approach might be used to improve the algorithm code organization and to partition the computation tasks between the threads to maximally increase the efficiency.

The effectiveness of our model is demonstrated by comparing its throughputs to practical computational experiments performed with the usage of Nvidia GPU. We study the impact on the complexity of various parameters such as the number of concurrent processes, and the level of memory used (shared memory vs. global memory). The successful application of our model is discussed in details, as a proof of concept, on a single example of digit radix sorting algorithm. We do not want to discuss methods of parallelization, nor the most efficient way of using different types of the memory. It is beyond the scope of this paper. Those problems are very complex, and cannot be explained in such a short publication. More about those topics can be found for instance in [6, 20].

Our PN model is not the only one GPU efficiency model. There are quite a few other GPU performance models, which can be divided into two groups (according to [12]):

(1) Calibrated performance models that make specific predictions and include many lower levels of details. They usually contain many specialized parameters, some of which may be difficult to obtain or calculate. The first example is a model presented in [7]. It is the first analytical model of the GPU efficiency. The most important values used there are MWP (number of memory requestes that can be executed concurrently) and CWP (number of warps, which can be computed while one warp is waiting for memory values). The model consists of 14 equations, which contains 21 parameters. Other example is a model presented in [19]. The authors created not only performance GPU model but also power consumption model. In the performance part they estimate execution time for individual GPU architecture components (e.g. shared, global memory) separately. This way they easily identify potential performance bottlenecks. The model has a form of equations and contains 32 parameters.

(2) Asymptotic models for algorithm analysis at a high level of abstraction that capture only the essential features of GPU architecture. One of the models from this group is the model described in [13]. He used elements of PRAM, BSP and QRQW approaches. The model calculates the number of cycles required for the whole kernel, taking into account the number of blocks, warps and threads, the maximal number of cycles required by a single thread to perform calculations and the number of threads which can be executed in parallel. However the model is quite simple and some important elements are neglected (like hiding memory latency in computations). The other asymptotic model is Thread Multi-Core Memory (TMM) model presented in [11]. Basing on the TMM, GPUs can be presented as abstract core groups, each containing a number of cores and fast local memory. A large and slow global memory is shared by all cores. The running time of the algorithm is calculated basing on suitable equation with basic GPU parameters. One of the most popular GPU efficiency models is the roofline model introduced in [21]. It can be used to obtain performance estimates of GPU computations, and requires two parameters: the number of operations

performed by a kernel and the number of bytes transferred from/to the memory, which can be used to calculate the arithmetic intensity $I$. In the naive roofline model $I$ can be handily presented as a point in two-dimensional space restricted by two ceilings lines: the memory bandwidth and the processor's peak efficiency. The resulting performance is a bound under which the arithmetic intensity appeared: the memory bandwidth bound or the peak performance bound. In the extended versions of the model, additional ceilings can be added. They related for example to software prefetching or task level parallelism. The roofline model can determine the type of kernel limitation and show, how optimal the program is. However, such a simple arithmetic operation cannot precisely represent the complex processes of kernel execution, like for example hiding the waiting period in calculations (see Figure 1). Similar problem occurs in other purely arithmetical models. More complex tools are necessary for such a purpose. Our model belongs to the asymptotic group, and to the best of our knowledge it is the first one utilizing Petri nets.

The paper is organized as follows. In the next section we describe Graphical Processing Units and CUDA Toolkit, focusing in particular on memory types (and organization). In Section 3 we recall some standard notions and notations related to Petri nets. Then we introduce our memory access and computation model followed by the description of radix sort algorithm. We also present the results of experiments conducted on the base of our implementations of the radix sort algorithm. We conclude the paper and give some directions of further research in the last section.

## 2 CUDA

An intensive development of Graphical Processing Units (GPU in short) resulted in construction of high performance computational devices, which besides the graphical display management, allow also the execution of parallel general purpose algorithms (not necessarily related to computer graphics). As opposed to CPU consisting of a few cores optimized for sequential serial processing, GPU contains thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. The most popular ones are GPU's produced by Nvidia Corporation, supplied with Nvidia CUDA Toolkit (see [2]).

A program running in heterogeneous environment equipped with GPU can be split into so-called *host* parts, which are executed by CPU, and so-called *kernel* parts, which are executed by GPU. The host part specify the kernel execution context and manages the data transfer between the host and the GPU memory. The kernel functions create a big number of threads allowing a highly parallel computation (where each thread runs the same kernel code). GPU threads, as opposed to CPU threads, are much lighter, hence their creation and controlling requires less CPU cycles.

All threads executed on GPU are organized into several equally-sized thread blocks, which in turn are organized into a grid.

A thread block is the set of concurrently executed threads. Such an execution and the thread cooperation can be coordinated by a barrier synchronization. Moreover, the data can be exchanged between threads using a shared memory. The size of a block is limited by the capacity of resources accessible on a single processor core. On currently available GPU's a single block can contain up to 1024 threads. Each thread block within a grid is uniquely identified by its block ID, and each thread within a block – by its thread ID.

A grid is an array of thread blocks executing the same kernel. The number of blocks in a grid is specified by the amount of data to be processed and the number of available processors. The exchange of data between threads within a grid requires the usage of the global memory. Moreover, while all threads within a single block run simultaneously, different blocks can be executed in any order.

The physical Nvidia GPU architecture consists of several multithreaded streaming multiprocessors (SM in short). Thread blocks are distributed between SM's in such a way that all threads within a single block are concurrently executed on the same SM (different blocks may, but not necessarily have to, be executed on the same SM). A streaming multiprocessor organizes threads into so-called *warps* consisting of 32 threads each. The partition is done according to increasing thread ID. Each SM works utilizing SIMT (single-instruction, multiple-threads) architecture, which means that all threads within a single warp execute one common instruction at a time. Any divergence (e.g. caused by a data processed) leads to a serial execution of a single computation path until possible convergence to the same execution path.

A single SM serves multiple warps. It is equipped with a number of warp schedulers and instruction dispatch units (currently 2 or 4 depending on the device used). A scheduler selects a warp, which is ready to be executed and issues it to the physical cores of GPU. If the currently active warp needs to wait for the memory read/write operation it is replaced by another ready warp. While the replaced warp is waiting for the memory access, other warps perform their computations, therefore the SM is busy as often as possible. The waiting period of a single warp is hidden behind the computations of the others (if there are only enough active warps available) and is not seen outside the SM. The simulation of such a behavior is the main part of our model.

The above mentioned heterogeneous environment is equipped with the *host memory* managed by CPU and the GPU *device memory*. The latter is significantly more complex than the former. Due to a necessary compromise between the data transfer/access speed and the possible capacity, the GPU device memory consists of various types of data storage, such as global, constant, local and shared memory.

The global memory is the largest and at the same time the slowest type of GPU memory (with hundreds of cycles latency). Together with the constant memory it is the only type of GPU memory, which can be accessed by the host. It is available for reading and writing for all running threads, however the data exchange and result sharing are possible only after a kernel-wide global synchronization.

The content of the global memory is accessed in blocks of size 32, 64 or 128 bytes (depending on the device used). Every time an element within a block is accessed, the whole block have to be transferred. Therefore, the concurrent (among the threads within a single warp) global memory read and write operations are grouped into transactions, the number of which depends on the cache lines required to serve all threads within a warp. However, if different threads in a warp refer to different memory blocks, all such blocks have to be transferred to cache sequentially. Such a situation cause the necessity of repeated global memory accesses.

The shared memory is a fast memory physically placed inside a multiprocessor. It consists of blocks, each of which is available for all threads within a single thread block. Moreover, each such block is divided into several so-called memory banks. The access of different threads to different banks is realized simultaneously, while the access of different threads to the same bank is realized sequentially. Such a situation is called the *bank conflict*. The shared memory can be used for data exchange between threads within the same thread block after block-wide thread synchronization.

The local memory of a single thread consists of a number of registers, which are the fastest type of memory available (with almost negligible access time). It is used to store local thread variables. Due to large number of threads the capacity of each thread local memory is strongly limited.

To complete the picture we have to mention also the constant and texture memory – dedicated parts of the GPU device memory (usually buffered). Both of them are optimized for access speed within the device, but are available for threads in read-only mode. Neither of them is considered in our model.

## 3 Petri Nets

The set of non-negative integers is denoted by $\mathbb{N}$. Given a set X, the cardinality (number of elements) of $X$ is denoted by $|X|$, the powerset (set of all subsets) by $2^X$ – the cardinality of the powerset is $2^{|X|}$. Multisets over $X$ are members of $\mathbb{N}^X$, i.e., functions from $X$ into $\mathbb{N}$. For convenience and readability, if the set $X$ is finite, multisets in $\mathbb{N}^X$ will be represented by vectors of $\mathbb{N}^{|X|}$ (assuming a fixed ordering of the set $X$). The addition and the partial order $\leq$ on $\mathbb{N}^X$ are understood componentwise, while $<$ means $\leq$ and $\neq$.

Let us now recall basic definitions and facts concerning inhibitor Petri nets [1, 16].

**Definition 1.** *An inhibitor*[1] *place/transition net (p/t-net) is a quintuple* $S = (P, T, W, I, M_0)$*, where:*

- *$P$ and $T$ are finite disjoint sets, of places and transitions (actions), respectively;*

[1] Note that in the case of bounded nets the use of inhibitors is not necessary, one can provide an equivalent (with more complex structure) net without inhibitors.

- $W : P \times T \cup T \times P \to \mathbb{N}$ *is an arc weight function;*
- $I \subseteq P \times T$ *is an inhibition relation;*
- $M_0 \in \mathbb{N}^P$ *is a multiset of places, named the initial marking.*

For all $a \in T$ we use the following denotations:

$^\bullet a = \{p \in P \mid W(p,a) > 0\}$ – the set of *entries* to $a$
$a^\bullet = \{p \in P \mid W(a,p) > 0\}$ – the set of *exits* from $a$
$^\circ a = \{p \in P \mid (p,a) \in I\}$ – the set of *inhibitor places* for $a$.

Petri nets admit a natural graphical representation. Nodes represent places and transitions, arcs with classical arrow heads represent the weight function, while arcs with small circles as arrowheads represent inhibition relation. Places are indicated by circles, and transitions by boxes. Markings are depicted by tokens inside the circles, the capacity of places is unlimited. However, Petri nets used in our model are bounded (which means that there exist a common bound for all the numbers of tokens appearing during the computation in a single place).

The set of all finite strings of transitions is denoted by $T^*$, the empty string is denoted by $\varepsilon$, the length of $w \in T^*$ is denoted by $|w|$, number of occurrences of a transition $a$ in a string $w$ is denoted by $|w|_a$.

Multisets of places are called markings. In the context of p/t-nets, they are typically represented by nonnegative integer vectors of dimension $|P|$, assuming that $P$ is totally ordered.

A transition $a \in T$ is enabled at a marking $M$ whenever $^\bullet a \leq M$ (all its entries are marked) and $\forall_{p \in {}^\circ a}\, M(p) = 0$ (all inhibitor places are empty). If $a$ is enabled at $M$, then it can be executed. A marking $M$ is called a *dead marking* if no transition is enabled at $M$ (which means that $\forall_{a \in T} \exists_{p \in P} \big(W(p,a) > M(p) \vee ((p,a) \in I \wedge M(p) > 0)\big)$). The execution of an enabled transition $a$ is not forced and changes the current marking $M$ to the new marking $M' = (M - {}^\bullet a) + a^\bullet$ (tokens are removed from entries, then put to exits). We shall denote $Ma$ for the predicate "$a$ is enabled at $M$" and $MaM'$ for the predicate "$a$ is enabled at $M$ and $M'$ is the resulting marking".

In this paper however, we use the maximal concurrent semantics and in every marking we execute one of the maximal sets of enabled transitions (i.e. a step, which is maximally concurrent at this marking). Formally, a set of transitions $A \subseteq T$ is called *step* and is enabled if $\big(\sum_{a \in A} {}^\bullet a\big) \leq M$ and $\forall_{p \in (\bigcup_{a \in A} {}^\circ a)}\, M(p) = 0$. The execution of a step $A$ changes the current marking $M$ to the new marking $M' = \big(M - \sum_{a \in A} {}^\bullet a\big) + \sum_{a \in A} a^\bullet$. We say that a step is *maximally concurrent at marking* $M$ if $A$ is enabled at $M$ and $\forall_{a \notin A} A \cup \{a\}$ is not enabled at $M$.

The notions of enabledness and execution we extend, in a natural way, to strings of steps (*computations*): the empty string $\varepsilon$ is enabled at any marking, a string $w = Av$ is enabled at a marking $M$ whenever $MAM'$ and $v$ is enabled at $M'$. The predicates $MA$, $Mw$, $MAM'$ and $MwM'$ are defined like for single transitions.

## 4 Memory access and computations model

The Petri net model of GPU calculations is consistent with the workflow of multithreaded streaming multiprocessors (SMs). The model represents the way one SM operates. It models each warp assigned to the SM, selection of the next instruction for the SM, accesses to the global and shared memory, and arithmetic operations. It does not represent threads hierarchy (blocks, grid), repeated accesses to the global memory nor bank conflicts. The model allows simultaneous accesses to the global memory, but the number of warps, which can use the global memory, at the same time, is limited by the number of SM warp schedulers (2 or 4). Each element of the model was created basing on [3, 4].

The size of the considered Petri net depends on the number of warps. The two elements: the place $p_0$ – *SM* and the transition $t_0$ – *waiting* are the constant part of the model, other are generated for every warp. The place $p_0$ represents the streaming multiprocessor and its initial marking should correspond to the number of warp schedulers. For the modern graphical cards it should be 2 or 4. This place is connected by a loop with the transition $t_0$. The transition $t_0$ can be executed only when the warp schedulers cannot schedule any instruction, i.e. there is no instruction ready to be executed. The *waiting* transition is added to the model to gain control over how many steps of the calculations on the SM is idle. The minimization of that number is crucial for optimization of GPU programming. Beside those two elements, the PN model contains 18 places and 17 transitions for every warp required by the analysed algorithm. That part is called a warp part of PN model (WPNM). The detailed description of the most important places and transitions of WPNM is presented in Table 1.

The WPNM together with *SM* place and *waiting* transition are depicted in Figure 2. The place $p_1$ represents the activation of the warp. It is marked when the warp is active. The place $p_2$ is marked when the warp finishes the execution of its previous instruction. The warp can be scheduled for the execution only when the places $p_2$ and $p_4$ are marked. A token in $p_4$ means that the next instruction is selected and ready. The places from $p_3$ to $p_{10}$ and the transitions $t_2, t_3, t_4$ (the frame I part in Figure 2) are responsible for controlling and selecting the instructions. The three types of instructions are allowed in the model: an arithmetic calculation, an access to the shared memory and an access to the global memory. The initial marking of the place $p_5$ corresponds to the number $x_a$ of arithmetic operations required by the analyzed algorithm. Similarly, the initial markings of the places $p_7$ and $p_9$ represent respectively the numbers $x_s$ and $x_g$ of read/write operations from/to the shared and global memory. If the place $p_3$ is marked and at least one of the places: $p_5$, $p_7$, $p_9$ is not empty, the next instruction can be selected. The selection is random. When the instruction is chosen, according to its type (arithmetic calculation, shared memory access, global memory access) the marking of the corresponding place is decreased and a token is added to $p_4$. Moreover, (according to the instruction type) one of the places: $p_6$, $p_8$ or $p_{10}$ is marked. Now the selected instruction is ready to be executed and the warp can be processed by SM. The execution of the instruction

is represented by the three parts of the net marked in frame II, frame III and frame IV (see Figure 2). They correspond to the type of the selected instruction: frame II for an arithmetic calculation, frame III – an access to the shared memory and frame IV – an access to the global memory. The arithmetic calculation is simply represented by one place and two transitions. When the calculation is done, tokens are put in places: $p_2$ and $p_3$ (which means that the instruction is finished and the next one can be selected), and in the place $p_0$ (which means that SM is ready to execute the next instruction). The same situation is obtained when the access to the memory (shared or global) is finished, but those parts of the model contain more places and transitions. Those additional elements are used to model the memory access latencies. The shared memory latency and the global memory latency are the parameters of the model and are denoted by $l_1$ for the global memory and by $l_2$ for the shared memory. The transition $t_9$ ($t_{12}$ respectively) may be executed only after $l_1$ ($l_2$ respectively) executions of $t_8$ ($t_{17}$ respectively). For testing, their default values were 20 for $l_1$ and 2 for $l_2$, which is consistent with [4].

The transition $t_{16}$ is connected by inhibitor arcs with the places $p_5$, $p_7$ and $p_9$ (the frame I in Figure 2) and can be executed only when those places are empty, i.e. there is no instruction left for execution. The transition is also connected by a regular arc with $p_2$, Moreover, $t_{16}$ is the only one able to take the token from the place $p_1$ and its execution is equivalent to the termination of a given warp.

As it was mentioned above, the WPNM (places from $p_1$ to $p_{18}$ and transitions from $t_1$ to $t_{17}$) is generated for a single warp. In the case of multiple warps a separate WPNM should be generated for each of them. To distinguish between distinct WPNMs one can either increase the numeration of places and transitions accordingly or assign to them two-part labels consisting of the original place/transition number together with the warp id. It should be noticed that each place corresponding to $p_2$ and representing the readiness of the given warp should be connected by an inhibitor arc with the transition $t_0$. Moreover, each transition corresponding to $t_1$ should be connected with the place $p_0$ ($SM$). The same goes for transitions corresponding to $t_5$, $t_{14}$ and $t_{15}$. Note that the control is returned by $t_5$ not $t_{11}$ in the case of part responsible for the access to the global memory. Thanks to that, SM can process the next ready warp while the current one is waiting for the memory access.

Our PN model of GPU computation and memory access may by adapted for any algorithm. Instantiations of the model for different kernels may differ in the number of warps and the marking of places responsible for instructions counting (i.e. $p_5$, $p_7$ and $p_9$). For the chosen number of required warps, a new model containing sufficient number of WPNMs can be generated. The other possibility is the generation of one big model with the maximal possible number of WPNMs (i.e. 64 for modern graphical cards [4]), and then the necessary number of places representing active warps should be marked. The number of WPNMs can be calculated basing on the number of threads and blocks, which are parameters of the kernel. Notice that the model provides no controlling mechanism for the maximum number of WPNMs. It is up to the user to be aware

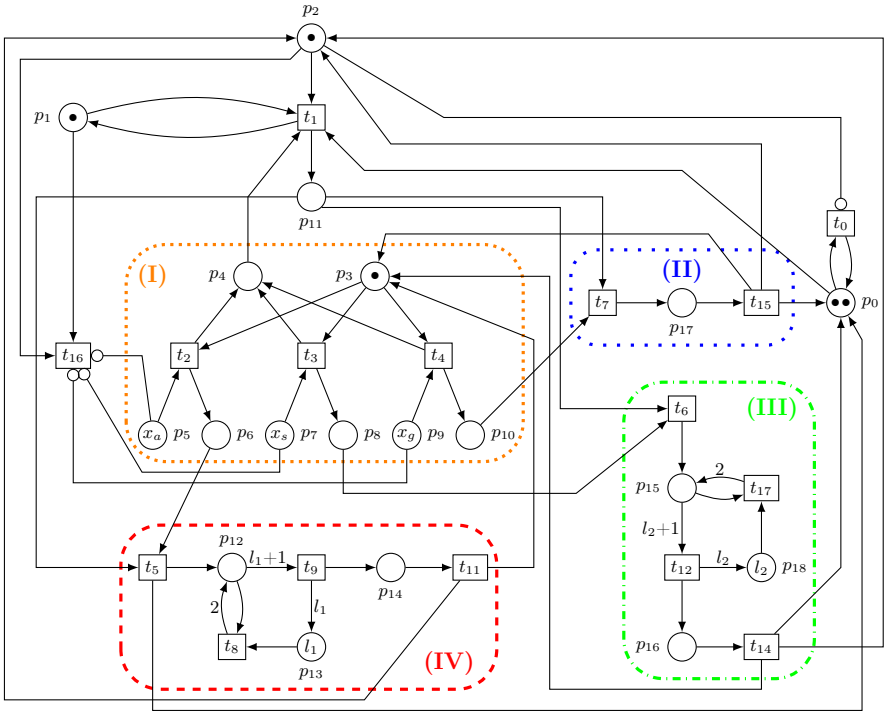


**Fig. 2.** The example of a warp PN model (WPNM). In frame I part the next instruction is selected. Sections: frame II, frame III and frame IV represent the execution of different types of instructions: arithmetic calculation, access to the shared memory and access to the global memory (respectively).

| Place/Transition name and description | | | |
|---|---|---|---|
| $p_0$ | Streaming Multiprocessor (SM) | $t_0$ | Waiting |
| $p_1$ | Active warp | $t_1$ | The warp is executed on SM |
| $p_2$ | The previous instruction is finished and the warp is ready for the next one | $t_2$ | Selection of a calculation for the next instruction |
| $p_3$ | Check the next instruction | $t_3$ | Selection of an access to the shared memory for the next instruction |
| $p_4$ | Next instruction is ready | $t_4$ | Selection of an access to the global memory for the next instruction |
| $p_5$ | Arithmetic operations | $t_5$ | Execution of the access to the global memory |
| $p_6$ | Instruction – arithmetic calculation | $t_6$ | Execution of the access to the shared memory |
| $p_7$ | Access to the shared memory | $t_7$ | Execution of arithmetic operation |
| $p_8$ | Instruction – shared memory access | $t_8$ | Waiting for the global memory access |
| $p_9$ | Access to the global memory | $t_9$ | Access to the global memory |
| $p_{10}$ | Instruction – global memory access | $t_{11}$ | Access to the global memory is finished |
| | | $t_{12}$ | Access to the shared memory |
| | | $t_{14}$ | Access to the shared memory is finished |
| | | $t_{15}$ | Calculation is finished |
| | | $t_{16}$ | Termination of the warp |
| | | $t_{17}$ | Waiting for the shared memory access |

**Table 1.** The description of the places and transitions depicted in the Figure 2.

that the maximal number of WPNMs is limited to 64 in modern GPUs. The number of arithmetic operations and accesses to the shared and global memory

need to be calculated for the considered algorithm. Those numbers should be used as the initial marking of places $p_5$, $p_7$ and $p_9$. If the model is constructed according to the description above, it is ready to be used out of the box. The usage of the model involves the execution of computations according to the maximal concurrent semantics (i.e. concurrent execution of all transitions that are enabled) starting from the initial marking until reaching the dead marking. The latter is obtained only when all places corresponding to $p_1$ (for each WPNM) become empty, which is equivalent to the termination of all warps. The number of steps of the computation is returned by the model and corresponds to the GPU execution time.

The maximal concurrent semantics requires execution of all enabled transitions. However, some of the enabled transitions may be in a conflict (i.e. execution of one transition makes disables another transition). In our implementation of the model, in every step, a permutation of the enabled transitions is randomly generated (using uniform distribution). Transitions are executed according to an order determined by the generated permutation. If two (or more) transitions are in a conflict, the transition appearing earlier in the considered order is executed.

The initial tests of the PN model were performed using the matrix multiplication kernel from [4] and the *stability* example from [20]. The PN models were generated for both kernels. The matrix multiplication program was executed many times with different sizes of matrices. Similarly, the *stability* example was executed with different values of *Time Step* and *Final Time* parameters. For the same data, the computations of the PN model were executed. The execution times of kernels and the numbers of steps of PN computations were compared. The results were consistent in both cases.

## 5 Experimental results

In the main experiment we used a binary version of the least significant digit radix sort algorithm [17, 18]. The idea of this method is to sort a list of positive $n$-bit integers using their binary representation. We make $n$ runs rearranging the list in such a manner that in $i$-th run all the integers having 0 on $i$-th bit are arranged in the first part of the array, while those having 1 – in the second part. An important requirement is to preserve the order of elements which do not differ on the processed bit. In other words, the sorting subroutine need to be stable. As a side effect the whole sorting procedure is also stable.

In the parallel version we made $n$ runs (one for every bit), each run consisting of three phases. At the beginning of each run, we partition the dataset equally between $m$ nodes. During the first phase, $j$-th node counts $zeros[i, j]$ – the number of elements containing 0 on $i$-th bit (consequently, we know $ones[i, j]$ – the number of elements with 1 on $i$-th bit for this node). In the next phase, we need to compute the positions for the set of elements assigned to each node. Namely, $j$-th node should place all the elements having 0 on $i$-th bit between $\sum_{k<j} zeros[i, k]$ and $\left(\sum_{k\leq j} zeros[i, k]\right) - 1$, while those with 1 on $i$-th in the

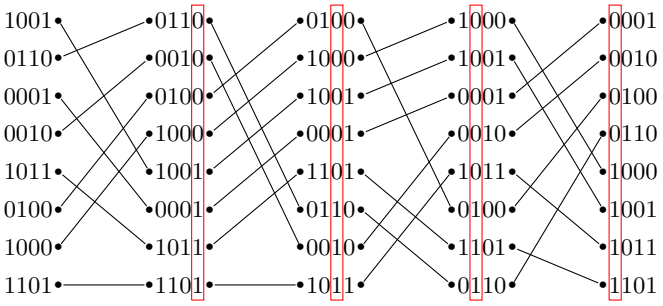


**Fig. 3.** Example of the use of radix sort procedure for four-bit integers. In consecutive columns we present the lists after each run of sorting subroutine. The rectangles emphasize columns with freshly sorted bits.

range

$$\Big[\sum_{k\le m} zeros[i,k] + \sum_{k<j} ones[i,k], \sum_{k\le m} zeros[i,k] + \big(\sum_{k\le j} ones[i,k]\big) - 1\Big].$$

The last, third phase, is the rearrangement of the list of integers. Each node traverse assigned part of the data splitting it into two parts (containing only 0 on $i$-th bit and only 1 on $i$-th bit) with the use of the positions computed in the second phase and in a stable manner. Since the output space for the nodes is partitioned into disjoint blocks, this phase may be realized using either shared or global memory.

We consider three CUDA implementations of the algorithm described above. In all versions, the array of integers $A$ to be sorted is stored in the global memory. During each kernel execution, one block of threads (with different number of threads) is created. Each thread has its own part of the array $A$ assigned. Its size is the parameter and is denoted by $memsize$. The product: $threadsNumber * memsize$ should be equal to the size of $A$. At the beginning of each run, every thread copies the assigned part of the array $A$ from the global memory to its local registers, then calculates number of 0 and 1 bits. At the end of the run, the content of the global memory array is rearranged – each thread moves the elements from its part of $A$.

Each of the three implementations of the radix sort algorithm was tested on a randomly generated array of 65536 integers, with five combinations of $threadsNumber$ and $memsize$ parameters. For the given implementation and the value of parameters, the numbers of arithmetic operations and accesses to the global and shared memory were calculated and used in the PN model (as the initial marking of the places $p_5$, $p_7$ and $p_9$). As an arithmetic operation we count every assignment, addition, subtraction, multiplication, division, relational operation, logical operation and array subscript (arrays in registers). Every access (read or write) to data stored in the global or shared memory is counted as single memory operation.

The numbers of steps of the model calculations were compared to the execution times of kernels. The tests were performed on NVIDIA GeForce GTX 960M graphical card with CUDA Toolkit 8.0. The execution times of kernels are averages of one hundred runs. The results for the PN model were calculated as averages of ten computations of the models.

In the first implementation only the global memory is used. The second phase of the algorithm is performed by thread with id *0*. The results of the tests are depicted in Figure 4 – the dotted line.

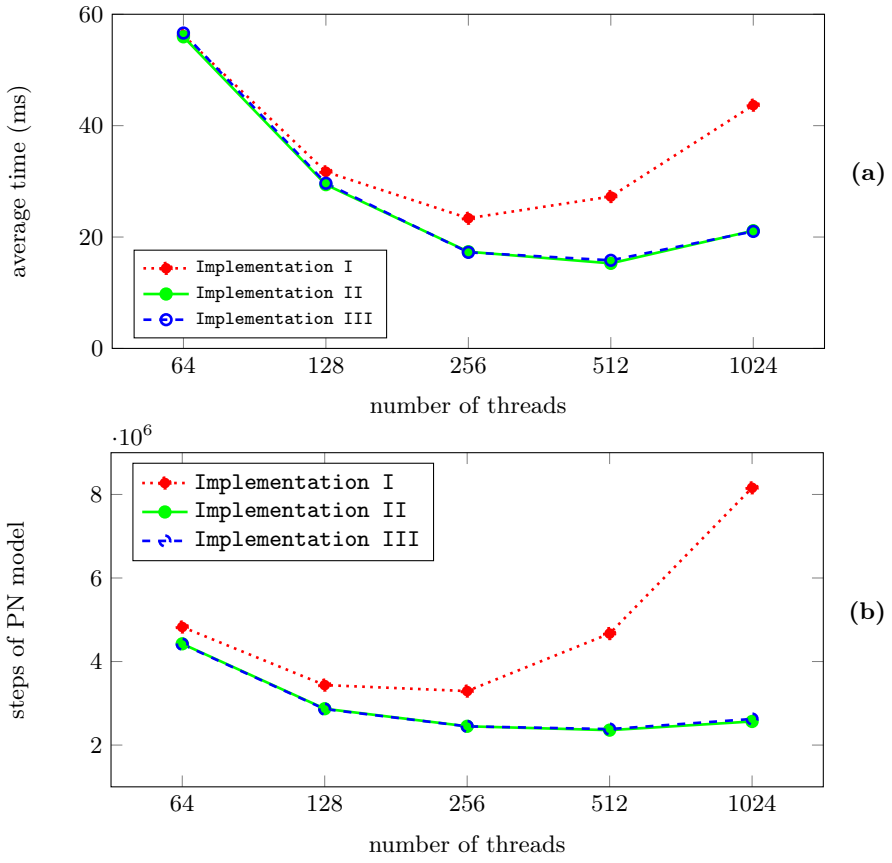


**Fig. 4.** The results for the radix sort algorithm tests with 65536 element arrays: (a) execution times of kernels (ms), (b) steps of the PN model.

In the second implementation the realization of the second phase is organized in a more efficient way. Instead of computing all sums incrementally, we compute all partial sums (for indexes between $p \cdot 2^q$ and $(p+1) \cdot 2^q$, where $p$ and $q$ are

suitable non-negative integers) and use them as input components for other sums.

Having $m = 2^r$ nodes we can compute $zeros[i, j]$ and $ones[i, j]$ for all $j \leq m$ in $r+1$ cycles with full system load (using all nodes in every cycle). To do it, we compute in $c-th$ cycle specific partial sums of lengths between $2^{c-1}$ and $2^c - 1$, an example for $r = 2$ is depicted on Figure 5. In this sample case $z[x..y]$ denotes $\sum_{x \leq t \leq y} zeros[i, t]$, while $o[x..y]$ – $\sum_{x \leq t \leq y} ones[i, t]$, each row corresponds to a single element in table *zeros* or *ones*, while in subsequent columns the values of partial sums stored in those elements are given. Each arc between $x - th$ and $y - th$ row denotes the addition of value kept in $x - th$ element to the value kept in $y - th$ element, the result is stored in $y - th$ element.

More specifically, in the first cycle we compute

$$zeros[i, 2k] + zeros[i, 2k + 1] \quad \text{and} \quad ones[i, 2k] + ones[i, 2k + 1]$$

placing results in $zeros[i, 2k + 1]$ and $ones[i, 2k + 1]$ respectively. The number of all operations made in this cycle (the number of arcs between first and second column in example) is $m/2 + m/2 = m$, hence we can utilize all available nodes to do it at once.

In subsequent cycles we compute longer partial sums using already precomputed ones. This way in $c - th$ cycle we compute

$$\sum_{0 \leq t \leq u} zeros[i, 2^c k + t] \quad \text{and} \quad \sum_{0 \leq t \leq u} ones[i, 2^c k + t],$$

where $2^{c-1} \leq u < 2^c - 1$, while $0 \leq k < 2^{r-c} - 1$, and store the results in $zeros[i, 2^c k+t]$ and $ones[i, 2^c k+t]$, respectively. Since after the previous cycle all values $\sum_{0 \leq t \leq u} zeros[i, 2^c k+t]$ and $\sum_{0 \leq t \leq u} ones[i, 2^c k+t]$ are stored in memory, where $2^{c-2} \leq u < 2^{c-1}$, while $0 \leq k < 2^{r-c+1}$, we need to add only two elements for each longer partial sum computed in $c - th$ cycle. Note that the number of such operations equals to the size of the range of $u$ multiplied by the size of the range of $k$ and doubled (we need to compute both ones and zeros), i.e.

$$|\{u, 2^{c-1} \leq u < 2^c\}| \cdot |\{k, 0 \leq k < 2^{r-c}\}| \cdot 2 = 2^{c-1} \cdot 2^{r-c} \cdot 2 = 2^r = m.$$

Finally, in the last cycle we add the computed so far $\sum_{k \leq m} zeros[i, k]$ to all $\sum_{t \leq k} ones[i, t]$ for each $k \leq m$. The execution times of kernels and the results from the PN model are depicted in Figure 4 – the solid line.

In the third version of the implementation the arrays *zeros* and *ones* are stored in the shared memory instead of global. The results of those tests are also presented in Figure 4 – the dashed line.

The results of the PN model calculations for the first implementation (Figure 4 (a)) clearly show that this parallelization is not very efficient as compared to the others, especially for larger numbers of threads. This is confirmed by the execution times of kernels (Figure 4 (b)). One can easily observe that in both plots the number of PN model steps and execution times of kernels for this implementation initially decrease with the increasing number of threads, however

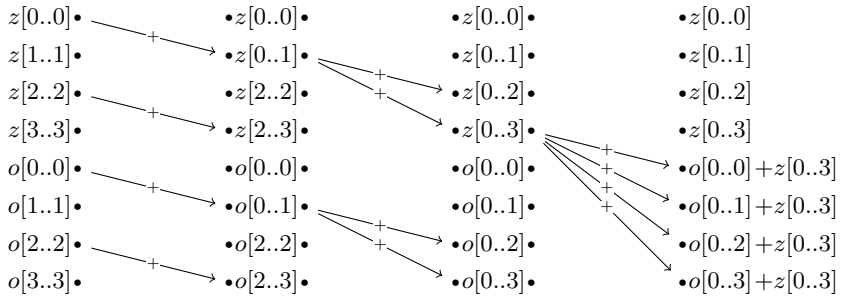


**Fig. 5.** The organization of the second phase of the algorithm for $m = 4$ (as $m = 2^r$, $r = 2$) nodes in $r + 1 = 3$ cycles.

for more than 256 threads both of them increase. The similar situation can be also noticed for other implementations, but here the growth is more significant. It can also be observed that for the largest amounts of threads the number of PN model steps increases faster than execution times of kernels. In this case the general direction of changes predicted by our model is consistent with the kernels executions (despite of the lack of the exact match of the plots). The model predicted correctly the most efficient choice of the number of threads, which in this case is 256.

The predictions of the PN model for the 2nd and 3rd implementations are more consistent with the execution times of kernels. In both cases differences between implementations are very small. It is probably caused by a relatively small number of shared memory operations in comparison to accesses to the global memory. For the larger number of threads, the increase in execution times of kernels is more significant than in the results from the PN model. The reliable explanation of such a difference is an overhead for communication between threads. It is clear that such overhead will not be observed in the PN model. However, the general characterization of the results is the same both for the model and the kernels. The PN model predicted correctly also the most effective choice of the number of threads for both implementations, which is 512 threads.

## 6 Conclusions and future work

The purpose of our PN based GPU computations and memory access model is to help in the analysis and optimization parallel algorithms, which are designed to be implemented on CUDA graphical cards. We do not require the source code to be given as an input, however the algorithm description should be detailed enough to estimate the number of arithmetic operations and accesses to the global and shared memory. Note that the other tools (e.g. the roofline model) also require those information. The expected speedup of the computation from a parallelization can be predicted and compared to other algorithms, even with-

out using of any physical GPU device. The model can help to predict which algorithm is the fastest, how much its modifications can affect the speedup of the computation and whether they are significant enough to include them in the source code.

Any inaccurate results demonstrated by our model might be interpreted as a premise that the algorithm should be improved. As an example recall the presented results for the radix sort algorithm. The predictions of the PN model for the first implementation were not satisfying, and the model clearly showed that a different organization of the second phase of the algorithm improved the time of the computation. On the other hand, using the shared memory in this case was not so beneficial. That was confirmed by the GPU kernels execution times.

Another important advantage of the PN model is the possibility to check how different values of parameters and the level of parallelization can affect the final efficiency. As it can be observed in Figure 4, it is not only a theoretical discussion, the problem is substantial and can result in very different execution times of kernels. With the appropriate number of threads, the GPU calculations were even three times faster. It should be also noticed that for all three implementations presented above, different numbers of threads were the most efficient, hence the selection of the one, universal number of threads is not possible. Our model easily allows to check different number of threads (warps) and its predictions seem to be very accurate. Various values of parameters may also result in different numbers of arithmetic operations and accesses to the memory, and it can be also easily introduced and analyzed by the model.

Our model can freely swap instructions of various types. It allows to check whether different order of the instructions may improve the algorithm efficiency, for example by allowing to hide thread waiting periods in calculations. If the results of the PN model are significantly better than the results from GPU kernels execution, that possibility should be considered. Naturally, the swap of the instructions is not always possible because of the nature of calculations. One of the most important improvements of the model would be the introduction of a partial order over the set of instructions. This can be achieved by defining dependence of instructions basing on the access to the same variable (see [9]). The partial order would make predictions of the model more accurate.

In the model, the bounded inhibitor Petri nets are used. In the future we would like also to consider other PN semantics like colored PNs, timed PNs, etc.. Their nature seems to be suitable for our model and they may help to simplify it or introduce new features.

The PN results are expressed as the number of steps performed, while for the GPU computations we use the execution times of kernels in milliseconds. To compare them directly the special coefficient is required to align one result with the other. However, different nature of various algorithms as well as the lack of research on atomic and comparable in terms of time consumption operations makes the issue of finding the universal coefficient a very hard task.

Although designing of the efficient sorting algorithm was not the aim of this paper, we described the process of improving the parallel version of the considered radix sort algorithm. Nevertheless, it is worth to note that providing its further improvements is possible. The radix sort is quite popular, both in the most significant digit (MSD), normally together with merge sort subroutine [5], and the least significant digit (LSD), as suggested in [10], version.

Note that for different GPU devices the execution time of a fixed kernel may differ, while the number of steps performed by the model stays the same. It would be useful to prepare a set of benchmarks, which for a given device compute the universal scaling coefficient for this device and the model.

## References


1. Giovanni Chiola, Susanna Donatelli, and Giuliana Franceschinis. Priorities, inhibitor arcs and concurrency in p/t nets. In *Proc. of ICATPN*, volume 91, pages 182–205, 1991.
2. Nvidia Corporation. CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.
3. Nvidia Corporation. CUDA. Best practice guide version 8.0.61, 2017.
4. Nvidia Corporation. CUDA. C programming guide version 7.5, 2017.
5. Victor J. Duvanenko. Algorithm improvement through performance measurement. `http://www.drdobbs.com/architecture-and-design/algorithm-improvement-through-performanc/220000504`, 2009.
6. Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.
7. Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.
8. Kai Hwang and Naresh Jotwani. *Advanced Computer Architecture, 3e*. McGraw-Hill Education, 2011.
9. Ryszard Janicki and Maciej Koutny. Structure of concurrency. *Theoretical Computer Science*, 112(1):5–52, 1993.
10. David Luebke, John Owens, Mike Roberts, and Cheng-Han Lee. Intro to parallel programming – online course. Nvidia Corporation.
11. Lin Ma, Kunal Agrawal, and Roger D Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, 2014.
12. Lin Ma, Roger D Chamberlain, and Kunal Agrawal. Performance modeling for highly-threaded many-core gpus. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 84–91. IEEE, 2014.
13. Rishabh Mukherjee. *A Performance Prediction Model for the CUDA GPGPU Platform*. PhD thesis, International Institute of Information Technology Hyderabad, India, 2010.
14. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
15. David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
16. Wolfgang Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012.

17. Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.
18. Harold E Seward. Information sorting in the application of electronic digital computers to business operations, Master Thesis, MIT, 1954.
19. S Shuaiwen, S Chunyi, R Barry, and C Kirk. A simplified and accurate model of power-performance efficiency on emergent gpu architecure. In *Parallel &Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 673–686, 2013.
20. Duane Storti and Mete Yurtoglu. *CUDA for Engineers: An Introduction to High-performance Parallel Computing*. Addison-Wesley Professional, 2015.
21. Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.