Computing Optimal Cycle Mean in Parallel on CUDA[*]

Jiří Barnat

Faculty of Informatics
Masaryk University
Brno, Czech Republic

barnat@fi.muni.cz

Petr Bauch

Faculty of Informatics
Masaryk University
Brno, Czech Republic

xbauch@fi.muni.cz

Luboš Brim

Faculty of Informatics
Masaryk University
Brno, Czech Republic

brim@fi.muni.cz

Milan Češka

Faculty of Informatics
Masaryk University
Brno, Czech Republic

xceska@fi.muni.cz

Computation of optimal cycle mean in a directed weighted graph has many applications in program analysis, performance verification in particular. In this paper we propose a data-parallel algorithmic solution to the problem and show how the computation of optimal cycle mean can be efficiently accelerated by means of CUDA technology. We show how the problem of computation of optimal cycle mean is decomposed into a sequence of data-parallel graph computation primitives and show how these primitives can be implemented and optimized for CUDA computation. Finally, we report a fivefold experimental speed up on graphs representing models of distributed systems when compared to best sequential algorithms.

1 Introduction

High quality implementation of complex computer systems, e.g. complex embedded systems, is a major challenge today and the computer industry struggles with how to efficiently engineer these systems. Implementations of these systems raise complex parallelism and scheduling issues, which are in practice solved by hand or, at best, by using emerging tools that address only a limited set of applications with favourable properties, such as static nests of loops. One way to tackle this challenge is to use model-driven engineering.

Model-driven engineering is a very active academic domain, driving many studies and prototype tools, and there is an emerging industrial market with an expected growth greater than 10% per year (cf Forrester Consulting). Model-driven performance analysis introduces performance analysis in the early design phases and leading thus to design of more reliable and optimal system.

Inspections of graph cycles is one of the possible means to deal with performance prediction. For example assume that transitions of a system are labelled with resource consumption that the actions these transitions model impose on the system. Then by finding the *maximal cycle mean* of a graph representing the system it is possible to approximate the worst sustainable load—the amount of resources consumed— under which the system will operate. Unlike using Queueing networks [16] or stochastic Petri nets [20] the computation of optimal cycle mean (OCM) allows to measure the worst expectable performance over infinite runs.

The potential use of the OCM computation for performance analysis should be more apparent with our running example of a client/server distributed application. Provided we are able to create a model of this application with edge-labels representing consumption of CPU resources we intend to compute how many clients will consume how much of the CPU of the server. Analogously, the inspection of properties of critical cycles, and especially the computation of OCM, allows to analyse performance of a large number of systems.

It has been shown that the system to be analysed can be modelled as a Petri net [22], Process graph [19] or e.g. a Data flow graph [15]. When an appropriate modelling formalism for the given real-world system is used, the enumeration of a specific cycle property facilitates performance evaluation of asynchronous systems [4], delay intensive and latency-insensitive systems [18]; rate analysis and scheduling of embedded real-time systems [19]; time-separation analysis of concurrent systems [5] and many others. The performance evaluation measures provided by OCM computation include resource consumption (CPU, memory, bandwidth, etc.), worst expectable latency and other measures in specific systems, e.g. cycle period in synchronous systems.

Many different approaches to compute OCM have been taken so far. Dasdan et al. [11] gave a comprehensive list of algorithms. However, it is imperative that the process of finding the cycles is not overly expensive since many of these applications require the critical cycle to be found repeatedly [10]. To further emphasize the necessity of having efficient algorithmic solution to this problem we present two practical observations. The graphs representing even a small system can be exceedingly large, containing millions of vertices. Moreover, the number of cycles can be exponential with respect to the number of vertices thus making the trivial inspection of all cycles in the graph impractical. Yet the asymptotic complexity of even the best sequential algorithms is very high, which renders the applicability of OCM-based performance analysis limited to small systems. We intend to improve the run-time of OCM computation and consequently the applicability of OCM-based performance analysis by employment of SIMD parallelism.

The potential of SIMD parallelism has been recently rediscovered (first efficiently employed in the Connection Machines [13]) with the entry of affordable graphics processing units (GPU) computation. The GPUs possess off-the-shelf data-parallel computation capability, which was soon realized by the academic community and has led to acceleration of various scientific computations. Among these applications of SIMD parallelism were also examples of acceleration of graph algorithms: several can be found in [12], together with thorough experimental evaluation on large sparse graphs.

Since the distributed computation of OCM algorithms has led to rather moderate results [6], we attempt to utilize the massive parallelism of modern GPUs to accelerate the OCM computation. In order to give a lucid description of the procedure we first provide details on what limitations are imposed on the algorithmic solution by the target architecture, i.e. the advantages and shortcomings of modern general purpose GPUs. Subsequently, we choose among the existing algorithms the one most appropriate for data-parallelization through careful inspection of the relative extend of the underlying graph operations. Then we describe the translation of this algorithm and finally conduct an experimental study, comparing our new data-parallel version with the sequential version and also with other sequential algorithms.

2 Preliminaries

In this section we briefly introduce the optimal cycle mean problem and the corresponding terminology within the context of graph theory. As promised in the introduction, the graph theoretical exposition of OCM is followed by an exemplary use of the formalism as a mean for performance analysis, allowing computation of the worst expectable resource consumption. We also describe CUDA computation technology and we detail on which OCM algorithmic approach would be most suitable for data-parallel implementation. The sequential version of the selected algorithm is then thoroughly expounded.
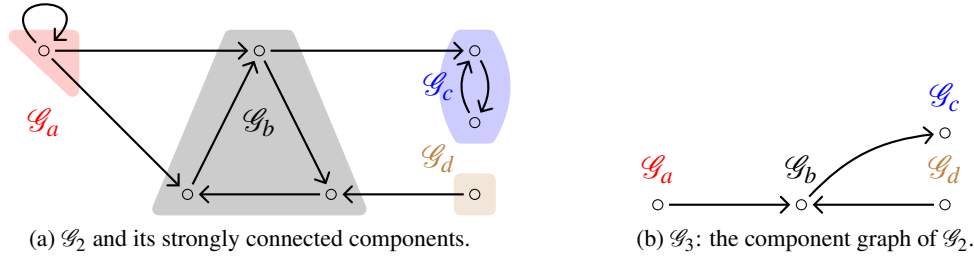
(a) $\mathscr{G}_2$ and its strongly connected components.

(b) $\mathscr{G}_3$: the component graph of $\mathscr{G}_2$.

Figure 1: Graph with strongly connected components and the corresponding component graph.

2.1   Related Graph Theory Definitions

A *graph* is a tuple $\mathscr{G} = (V, E)$, where V is a set of *vertices* and $E \subseteq V \times V$. As usual, cardinalities of sets $|V|$ and $|E|$ are denoted with n and m, respectively. A *path* in $\mathscr{G}$ is a non-empty sequence of edges $\pi = \langle e_1, \ldots, e_n \rangle$ such that $\forall 1 \leq i \leq n : e_i = (v_i - 1, v_i) \in E$. The length of path π is denoted as $|\pi|$ and for $\pi = \langle e_1, \ldots, e_n \rangle$ equals to n. A path for which $v_0 = v_n$ is called a *cycle*. The set of all cycles of graph $\mathscr{G}$ is denoted with $Z_{\mathscr{G}}$.

We say that a graph is *cyclic* if and only if every edge is part of some cycle. Furthermore, we say that a graph is rooted if there is a root vertex s of the graph such that all the other vertices are *reachable* from it, i.e. $\forall v \in V$ there is a path from s to v. Should there be no such a vertex in the graph, we *augment* the graph be adding a new vertex $s \notin V$ and edges $\{(s, v) | \forall v \in V\}$.

Let $\mathscr{G} = (V, E)$ be a graph. A *weight* function is a function $w : E \to \mathbb{R}$ that assigns a real weight to every edge of $\mathscr{G}$. We speak of weighted graph if $\mathscr{G}$ and w are given. Weight function naturally extends to paths as a sum of the weights of all the edges on the path, i.e. $w(\pi) \stackrel{df}{=} \sum_{i=1}^{n} w(e_i)$, where $\pi = \langle e_1, \ldots, e_n \rangle$. In the case of augmented graphs we put $w(e) = 0$ for every newly added edge e. Finally, let π be a cycle in a graph $\mathscr{G}$ weighted with a weight function w. We define *cycle mean* of cycle π as $\mu(\pi) \stackrel{df}{=} \frac{w(\pi)}{|\pi|}$. *Minimal cycle mean* for a given graph $\mathscr{G}$ and weight function w is then denoted with $\mu^*(\mathscr{G}, w)$, where $\mu^*(\mathscr{G}, w) = min\{\mu(\pi) \mid \pi \in Z_{\mathscr{G}}\}$. Henceforward, we will safely drop the graph and weight function from the notation of minimal cycle mean and will refer to minimal cycle mean simply as to *optimal cycle mean* that will be denoted by μ^*.

Definition 2.1. OCM problem*: For a given graph $\mathscr{G}$ and weight function w find the minimal cycle mean.*

In order to describe further details of some OCM algorithms we introduce the notion of parametric weight functions and strongly connected components. Given a weight function w and a real number Λ, we define *parametric weight function* $w_\Lambda \stackrel{df}{=} w - \Lambda$. We say that Λ is *feasible* for a graph $\mathscr{G}$ if no cycle in the graph has negative weight with respect to w_Λ. Directed graph $\mathscr{G} = (V, E)$ is *strongly connected* if $\forall u, v \in V$ there is a path from u to v. Graph $\mathscr{G}_s = (V_s, E_s)$ is a maximal *strongly connected component* of $\mathscr{G}$ if $V_s \subseteq V$, $E_s \subseteq E$ induced on V_s, and V_s is a maximal in V such that $\mathscr{G}_s$ is strongly connected. Graph $\mathscr{G}_c = (\mathfrak{V}_c, E_c)$ is a *component graph* of $\mathscr{G}$ if $\mathfrak{V}_c$ is the set of all strongly connected components of $\mathscr{G}$ and $e = (\mathscr{G}_1, \mathscr{G}_2)$ is in $E_c \subseteq \mathfrak{V}_c \times \mathfrak{V}_c$ if there is an edge in $\mathscr{G}$ between a vertex from $\mathscr{G}_1$ and a vertex from $\mathscr{G}_2$. An exemplary illustration of strongly connected components of a graph and its corresponding component graph see Figure 1.

Proposition 2.2. Λ *is feasible* $\Leftrightarrow \Lambda \leq \mu^*$.

Proposition 2.3. *Minimal cycle mean of $\mathscr{G}$ is equal to the smallest of minimal cycle means among the strongly connected components of $\mathscr{G}$.*

2.2   OCM as a Performance Analysis Measure

In order for us to be able to validate the applicability of the OCM computation as a performance analysis measure we have extended the DIVINE model checking tool with the possibility of OCM computation. Originally, the explicit state space, parallel model checker DIVINE has in its core an algorithm for computation of accepting cycle. Preserving most of the code, especially the part generating the transition system, we only needed to perform a few changes in the modelling language and to replace accepting cycle detection algorithms with OCM algorithms.

DVE, the modelling language of DIVINE, allows specification of communicating *processes* in a very intuitive fashion. From the general point of view the processes consist of *states* and we use *transitions* to move from one state to another. The communication is achieved by *synchronization* of two transitions from different processes: we require these two transition to be executed concurrently in the resulting transition system. To allow performance evaluation we have added specification of *cost* and *time* to every transition. The cost can represent consumption of some resource of computation (CPU utilization, memory requirements, etc.) and the addition of time allows to compute the more general optimal cycle ratio [19].

The primary motivation behind the extension of DIVINE was performance analysis of an online PDF editor that was, in the time of writing, being developed by the Normex company. The OCM-based analysis was used with the intention to measure the worst expectable utilization of the CPU server. For the analysis to be as precise as possible we have devised several scenarios of the behaviour of the clients and constructed the model as a synchronous composition of these scenarios. Specifics of the scenarios were determined based on results of a case study among users of a similar editor and on preliminary beta testing. The concrete findings of our analysis will be detailed in Section 4.

2.3   CUDA Computation

The Compute Unified Device Architectures (CUDA) [9], developed by NVIDIA, is a parallel programming model and a software environment providing general purpose programming on Graphics Processing Units. At the hardware level, GPU device is a collection of multiprocessors each consisting of eight scalar processor cores, instruction unit, on-chip shared memory, and texture and constant memory caches. Every core has a large set of local 32-bit registers but no or a very small cache (L1 cache has configurable size of 16-48KB). The multiprocessors follow the SIMD architecture, i.e. they concurrently execute the same program instruction on different data. Communication among multiprocessors is realized through the shared device memory that is accessible for every processor core.

On the software side, the CUDA programming model extends the standard C/C++ programming language with a set of parallel programming supporting primitives. A CUDA program consists of a *host* code running on the CPU and a *device* code running on the GPU. The device code is structured into the so called *kernels*. A kernel executes the same scalar sequential program in many *independent data-parallel threads*.

Each multiprocessor has several fine-grain hardware thread contexts, and at any given moment, a group of threads called a *warp* executes their instructions on the multiprocessors in a lock-step manner. When several warps are scheduled on multiprocessors, memory latencies and pipeline stalls are hidden primarily by switching to another warp. Overall the combination of out-of-order CPU and data-parallel processing GPU allows for heterogeneous computation as illustrated in Figure 2a, where sequential host code and parallel device code are executed in turns.

Data structures used for CUDA accelerated computation must be designed with care. First, they
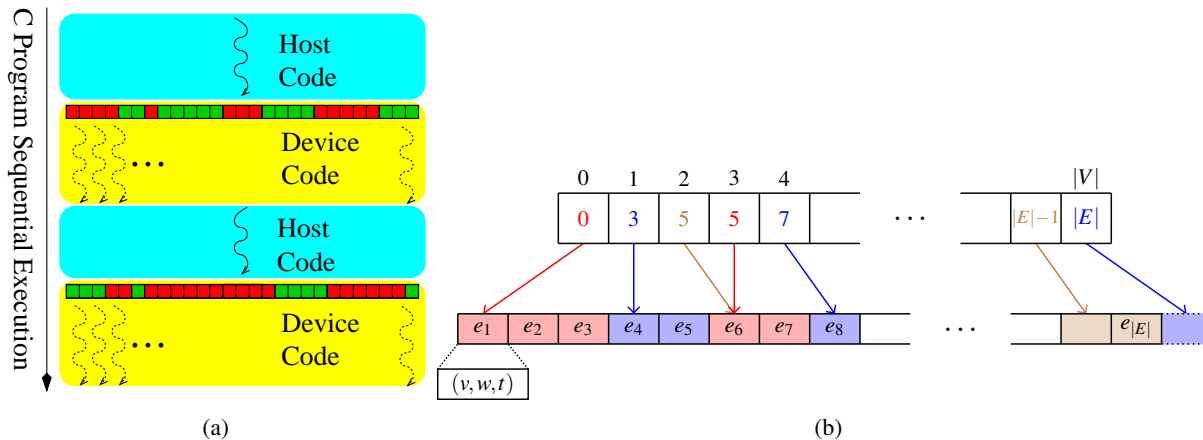
Figure 2: a) Sequential heterogeneous computation work-flow with CUDA. b) Adjacency list representation: a graph $G = (V, E)$ is stored as two arrays: A_i of size $|V| + 1$ and A_t of size $|E|$.

have to allow independent thread-local data processing so that the CUDA hardware can fully utilize its massive parallelism. And second, they have to be small so that the high latency device-memory access and limited device-memory bandwidth are not large performance bottlenecks (also the regularity of structures in the memory is of great importance). In our case, it is the representation of graph $\mathscr{G}$ to be encoded appropriately in the first place. Note that uncompressed matrix or dynamically linked adjacency lists violate these requirements and as such they are inappropriate for CUDA computation.

An efficient CUDA-aware computation of a graph algorithm requires the graph to be represented in a compact, preferably vector-like, fashion. We encode the graph as an adjacency list that is represented as two one-dimensional arrays A_t and A_i, similarly as in [12]. The array A_t keeps target vertices of all the edges of the graph. The target vertices stored in the array are ordered according to the source vertices of the corresponding edges. The second A_i array then keeps an index to the first array for every vertex in the graph. Every index points to the position of the first edge (represented as a target vertex) emanating from the corresponding vertex. See Figure 2b. If other data associated to a vertex are needed by a CUDA kernel algorithm, then they are organized in vectors as well.

Most of known OCM algorithms require to access predecessors of a given vertex in order to perform a kind of backward reachability. Storing backward edges together with their forward versions causes additional nontrivial memory requirements, which might be a problem as the size of CUDA memory is limited. We have shown how to carry out the backward reachability using only forward edges with minor time overhead [2]. However, OCM algorithms often require the reachability procedure to perform only on a given subgraph of the whole graph (such that it contains a single outgoing edge for every vertex). Unfortunately, augmenting the original procedure proposed in [2] in order to follow the selected edges only resulted in considerable slow down in computation. For that reason we were forced to explicitly store both forward and backward edges of the graph.

As mentioned before the amount of memory on a CUDA device may limit the applicability of CUDA accelerated algorithms to graphs with representation that would fit into the memory of the GPU. Multiple CUDA-aware GPUs can be used to effectively extend available memory [1] for the price of extensive modification of the source code and a certain slow down. Fortunately, the memory limitation is not that restricting in the case of OCM algorithms as the high asymptotic complexity of individual algorithms results in practical issues dealing with long run-times rather than a lack of memory space.

Algorithm 1: Scanning Method of SPF Algorithms

Input: A *found* vertex u

1 **foreach** $e = (u, v) \in E$ **do**
2   **if** $\pi(u) + w(e) < \pi(v)$ **then**
3          $\pi(v), p(v), S(v) \leftarrow \pi(u) + w(e), u, found$
4 $S(u) \leftarrow scanned$

2.4 OCM Algorithms

Through the course of study of optimal properties of graph cycles a plethora of algorithms emerged. These algorithms, while not sharing similar concepts, can be divided into several groups according to what graph property they use to find critical cycles. Since our goal in this section is to choose one of the algorithms which would be most suitable for data-parallel implementation we will describe all three categories of OCM algorithms and consider their aptness for our purpose.

There are various limitations imposed on the potential algorithms should they be even considered for SIMD acceleration. First of all, most of the data structures used should be to a very large extend in form of vectors: stacks, queues or heaps are not possible to be effectively data-parallelized. Then the kernels should prevalently address the whole set of vertices (or edges): limiting the computation to a insufficiently small subset would prevent utilization of the computational power. Yet the vector-wide operations are rather costly even on many-core architectures and the complexity of algorithms is practically measurable in their number. Finally, there is also a non-negligible overhead of kernel calls and thus even a very fast kernel should not be run excessively.

A strong relation can be observed between the OCM and the *Shortest Path Feasibility* (SPF) problems [10]. This relation should be much more apparent once we formulate the OCM problem as a linear programming problem: μ^* is the optimal solution of

$$\textbf{max } r \textbf{ subject to}$$
$$d(v) \leq d((u) + (w((u, v)) - r) \tag{1}$$
$$\forall e = (u, v) \in E,$$

where d stands for *distance*, i.e. minimal-cost path from the source node to v. According to the previous formulation, we can equivalently search for the maximal parameter r such that $\mathscr{G}$ with w_r contains no negative cycle. Following Proposition 2.2 such r is exactly the optimal cycle mean. As a result, existence of an efficient implementation of the SPF algorithm enables an efficient solution to the OCM problem. Furthermore, any future improvements to the SPF procedure automatically translates to improvements in this OCM problem solution.

All SPF algorithms have a common basic step: the *scanning method* [8]. This method assumes we maintain for every vertex u its potential $\pi(u)$, parent $p(u)$, and a label $S(u) \in \{unreached, found, scanned\}$. Initially, only the root vertex is labelled *found*, all other vertices are *unreached*, and potential of all vertices is set to zero. A single *found* vertex is then repeatedly scanned using Algorithm 1.

The scanning method repeats until there are no *found* vertices or until the algorithm finishes *n passes*. Recall that $n = |V|$. Passes of an SPF algorithm are defined inductively:

0-th pass is the initialization,

i-th pass scans all vertices labelled *found* during the $(i-1)$st pass.

If all n passes are performed, the graph inevitably contains a negative cycle.

2.4.1 Cycle-Based

The arguably most straightforward application of shortest path feasibility solution to the OCM problem is the *cycle-based* approach. The idea is to maintain an upper bound Λ of the minimal cycle mean, i.e. $\Lambda \geq \mu^*$, and a cycle C such that $\Lambda = \mu(\mathsf{C})$. If $\Lambda > \mu^*$ then new better upper bound Λ' of minimal mean cycle can be detected with the SPF algorithm provided that it uses parametric weight function w_Λ. The newly computed upper bound Λ' is used in another iteration of the algorithm as Λ. The whole procedure repeats until no improvement of upper bound can be found, which indicates that $\Lambda = \mu^*$.

A classical implementation of the cycle-based approach is the Howard's algorithm [14], which further improves the approach by altering the SPF procedure. At the end of each pass it checks the parent graph, induced by edges $(p(v), v)$, for cycles. Existence of a cycle allows to restate Λ to a value that sets to zero the weight of the most negative cycle. Should there be no negative cycle the improved SPF algorithm either terminates, if all reduced weights are non-negative, or it continues with the next pass.

The cycle-based approach seems to be fairly compatible with the SIMD computation. Namely, the SPF subroutine is a vector-wide propagation of values, the minimal cycle location on the parent graph is also feasible to parallelize, and most importantly sequential experiments suggest that the algorithm typically performs only very few passes of the underlying SPF subroutine.

2.4.2 Binary Search

The *binary search* approach is slightly more involved. It maintains both upper and lower bound $\Lambda_1 \leq \mu* \leq \Lambda_2$ together with a cycle C such that $\mu(\mathsf{C}) = \Lambda_2$. The SPF subroutine is repeatedly called with parametric weight function w_Λ, where Λ, as the name suggests, is set to $\frac{\Lambda_1 + \Lambda_2}{2}$. In case a negative cycle ζ is found, we set C and Λ_2 to ζ and $\mu(\zeta)$, respectively. Since we did not use Λ_2 as a parameter we cannot be certain of the value of the optimal cycle mean in either of SPF answers, and thus if no negative cycle is found we set Λ_1 to Λ. The termination criterion for binary search approach is $\Lambda_2 - \Lambda_1 < \varepsilon$. If ε is chosen sufficiently small, C will be the critical cycle.

The well-known implementation of binary search is due to Lawler [17], who also proved the run-time of his algorithm to be in $\mathcal{O}(nm \lg(W/\varepsilon))$, where W is the maximal edge weight. Structurally, there is no apparent reason why the Lawler's algorithm should be inappropriate for data-parallelization, yet the fact that the SPF subroutine requires up to n passes (and given the idea of binary search it often is necessary to carry out all n passes) renders this particular approach unusable. This hypothesis was experimentally confirmed once we have implemented the Lawler's algorithms and executed preliminary tests.

2.4.3 Tree-Based

While the previous two approaches used full SPF, the *tree-based* approach uses the shortest path feasibility subroutine only partially. Here only the lower bound Λ is maintained, initially small enough to guarantee that all edges have positive weight under w_Λ. Λ is progressively increased throughout the algorithm in correctly chosen increments, until a cycle ζ is found such that $w_\Lambda(\zeta) = 0$. Apart from Λ we also maintain the shortest path tree **T**, with respect to the current w_Λ. As we are working with the augmented graph we may initiate **T** to consist of the edges from s to every other vertex.

The increments of Λ must be chosen with care, otherwise minimal mean cycle could be missed. A safe strategy is to set new value of Λ to the smallest $\lambda \geq \Lambda$ such that there is a different **T** for w_λ. To this

end we assign to every vertex *u* a *threshold*, the smallest λ that would force *u* to change parent. Finding the smallest among all thresholds is facilitated by a priority queue.

Again this approach is unsuitable for SIMD acceleration for several reasons. The usage of priority queues (either heaps or Fibonacci heaps) is particularly problematic and would most likely be implemented as a simple vector, with the minimum operation as parallel reduction. Also the span of operation updating the shortest path tree is in many cases relatively small and would not fully utilize the number of GPU cores. Much larger problem was found during experiments with sequential version demonstrating that there are simply too many iteration of the algorithm that must be carried out one after another.

2.5   Howard's Algorithm

Since it is the Howard's algorithm that appears to be the one most suitable for parallelization, we will now provide its detailed description. First, we should stress that the algorithm works on strongly connected graphs only. There exist two approaches how to overcome this restriction. First, we can decompose the given graph to its strongly connected components and then process the graph one component at a time. In the sequential case we can use the Tarjan's algorithm [23] based on the depth-first traversal procedure which outputs the list of all strongly connected component in $\mathcal{O}(n+m)$ time. Hence there is asymptotically no difference in complexity of the algorithm, although practically the difference can be quite substantial. The second approach suggests to modify the underlying graph by adding a Hamiltonian cycle $\zeta_H = <(v_0,v_1),\dots(v_{n-1},v_0)>$ to the graph. With this modification the graph becomes strongly connected and provided that weights of newly added edges are sufficiently large, the optimal cycle mean of the graph remains unchanged.

As stated in the description of the cycle-based approach, the Howard's algorithm (see Algorithm 2, adopted from [6]) extends the shortest path feasibility algorithm. Indeed the main cycle on lines 7–27 up to line 14 is in fact a scanning step of the SPF algorithm. Altough the output of the cycle on lines 9–13 is not yet the shortest path tree, since we are approaching the optimal cycle mean from top and hence there are cycles in our shortest path graph. To remain consistent with the established notation we will call the graph induced on edges $(v, \pi(v))$ the *policy graph*.

Apart from the successor in the policy graph $\pi(v)$ that must exist since we are working with strongly-connected graphs only, we also store two values $\mathsf{val}_0(v)$ and $\mathsf{val}_1(v)$ with every vertex. In these two values we keep information about the current and the following parametric length for a given vertex. In every iteration of the out-most cycle we check whether there was a change in the policy graph, and if not we interrupt the main cycle and return λ as the optimal cycle mean. Each λ that is used as a parameter for the feasibility computation, is actually the mean weight of a specific cycle in both the original and the policy graph. Hence, after every iteration of the SPF algorithm part we inspect the policy graph, locate all cycles inside it and choose the one with minimal mean weight (line 16). Upon finding the minimal cycle (or after choosing one of the minimal cycles), we modify the policy graph in such a way that every vertex has a path to the minimal cycle (line 18).

Lines 16 and 18 would perhaps require more detailed explanation. From a property of the policy graph (that every vertex has exactly one outgoing edge), we know that each of its connected components consists of one cycle and potentially several paths leading to this cycle. Finding all cycles can thus be done in linear time simply by following the successor path, marking all visited vertices. Next we need to rebuild the policy graph so that the selected cycle would be the only cycle there and every vertex has a path to that cycle. Moreover, it is required for the component of the minimal cycle to remain unchanged, otherwise the SPF subprocedure would always detect an improvement. This can be achieved by two consecutive backward reachabilities: one to demarcate the component of the minimal cycle and the other

---

Algorithm 2: Howard's Algorithm

Input : A directed, strongly-connected graph $\mathcal{G} = (V, E, w), w : E \to \mathbb{Q}$
Output: $\lambda \in \mathbb{Q} : \lambda = \mu^*(\mathcal{G})$

1 **foreach** $v \in V$ **do**
2      $\mathsf{val}_0(v) \leftarrow 0$
3      $\pi(v) \leftarrow nil$

4 improved $\leftarrow$ true
5 i $\leftarrow 0$
6 $\lambda \leftarrow 0$
7 **while** improved **do**
8   improved $\leftarrow$ false
9   **foreach** $v \in V$ **do**
10          $\mathsf{val}_{((i+1) \bmod 2)}(v) \leftarrow \min_{u \in \mathtt{Succ}(v)} \{\mathsf{val}_{(i \bmod 2)}(u) + w(v, u) - \lambda\}$
11   **if** $\pi(v) = nil \vee (\mathsf{val}_{(i \bmod 2)}(\pi(v)) + w(v, \pi(v)) - \lambda > \mathsf{val}_{((i+1) \bmod 2)}(v))$ **then**
12              $\pi(v) \leftarrow u | \mathsf{val}_{(i \bmod 2)}(u) + w(v, u) - \lambda = \mathsf{val}_{((i+1) \bmod 2)}(v)$
13   improved $\leftarrow$ true

14   i $\leftarrow$ i $+ 1$
15   **if** improved **then**
16   c $\leftarrow$ MinMeanWeightCycle$(\mathcal{G}_\pi)$ // $\mathcal{G}_\pi.|E| = |V|, \forall v \in V : deg(v) = 1$
17          $\lambda \leftarrow \mu_{\mathcal{G}(\mathsf{c})}$
18   break all other cycles than c in $\mathcal{G}_\pi$ so that all vertices have path to c
19   s $\leftarrow$ MinVertex(c)
20          $\mathsf{val}_{(i \bmod 2)}(\mathsf{s}) \leftarrow 0$
21          $q.push(\mathsf{s})$
22   **while** $\neg q.empty()$ **do**
23   v $\leftarrow q.pop()$
24   **foreach** $u \in \mathtt{Pred}(\mathsf{v})$ **do**
25   **if** $u \neq s \wedge \pi(u) = \mathsf{v}$ **then**
26                  $\mathsf{val}_{(i \bmod 2)}(u) \leftarrow \mathsf{val}_{(i \bmod 2)}(\mathsf{v}) + w(u, \mathsf{v}) - \lambda$
27                  $q.push(u)$

28 **return** λ

---

one to connect also the remaining vertices to the minimal cycle.

Subsequently, we choose one vertex (line 19) on the minimal cycle and set its val to zero. It is necessary that we always select the same vertex, assuming the same cycle is found minimal. After the modification of the policy graph and selecting new λ, it is also necessary to modify the val values for other vertices accordingly. This process is started by setting the val of *s* to zero on line 20 and carried out by the cycle on lines 22–27, performing backward reachability from *s* along the edges of the policy graph. In the next iteration of the out-most cycle we again find the policy graph (using the updated val values. This process is iteratively applied until two consecutive policy graphs are found to be the same.

Algorithm 3: Howard's Algorithm – GPU (host code)

Input : A directed, strongly-connected graph $\mathscr{G} = (V, E, w), w : E \to \mathbb{Q}$
Output: $\lambda \in \mathbb{Q} : \lambda = \mu^*(\mathscr{G})$

1 **while** true **do**
2   **if** gTerminate $\leftarrow$ SPFPassIter$(\mathscr{G}, val, \lambda, \mathscr{G}_\pi, \text{it})$ **then break**
3   it$++$
4   GpiPreprocess$(\mathscr{G}_\pi, gPredInfo)$
5   elimination$^*(\mathscr{G}_\pi, gPredInfo)$
6   cycleIdentification$(\mathscr{G}_\pi, gCycles)$
7   reduce$_{\min}(gCycles, \text{minCycle})$
8     $\lambda \leftarrow$ minCycle.mean
9   setMinCycle$(\mathscr{G}_\pi, gCycles, \text{minCycle})$
10   markMinComponent$^*(\mathscr{G}_\pi, gPredInfo)$
11   connectGpi$^*(\mathscr{G}, \mathscr{G}_\pi)$
12     $val[\text{it}\&1][\text{minCycle.minIndex}] \leftarrow 0$
13   GpiPreprocess$(\mathscr{G}_\pi, gPredInfo)$
14   valuePropagate$^*(\mathscr{G}^{-1}, \mathscr{G}_\pi, val[\text{it}\&1], \lambda, \text{minCycle.minIndex}, gPredInfo)$
15 **return**

---

3 Data-Parallel Version of Howard's Algorithm

The actual description of our data-parallel implementation of Howard's algorithm will be conducted in several steps. We start by proposing a high-level work flow, where we attempt to preserve the provably correct layout. Concurrently proposing graph primitive operations that would perform actions functionally equivalent to those of the original algorithm, but, wherever possible, addressing the whole vector of values at a time. CUDA-specific implementation of these graph primitives will be detailed extensively in the following section. Finally, we propose an extension to Howard's algorithm which prepends a parallel decomposition to strongly connected components to the algorithm. Then we let the algorithm perform the OCM computation on all components concurrently.

3.1 High-Level Description

The proposed host code of our implementation is listed as Algorithm 3. It is apparent that lines 16 and 18 of Algorithm 2 that rebuild the policy graph, require much more attention in the SIMD environment as it is the place most susceptible to inefficient processing. These two lines of CPU pseudo-code span from line 4 to line 11 in our GPU implementation. We first describe this part of the algorithm postponing the SPF subroutine for later discussion.

There are two calls to the GpiPreprocess kernel (on lines 4 and 13) and they both serve the same purpose to gather information about predecessors in the policy graph. This step is merely an optimization speeding up the kernels that perform backward reachability (or its modification) on the policy graph. It would be possible to omit this kernel for the same reason it is possible to perform backward reachability using only forward edges [2]. Yet the speedup gained from employing this kernel is quite considerable even though we have to call it twice as the graph is rebuild in kernel connectGpi. The first call is

required because of the `elimination` and `markMinComponent` kernel, the second call is because of `valuePropagate` kernel.

From the description of sequential Howard's algorithm we know that the policy graph consists of *weakly connected* components, each containing a cycle and several paths leading to this cycle. In order to be able to find all cycles in the policy graph in as few parallel steps as possible, we first apply `elimination` to remove all vertices that do not lie on any cycle, i.e. those that are on the paths leading to a cycle (*path vertices*). This kernel is called iteratively (every call removes the vertices with no predecessors) until a fixpoint is found; in other words until there are no such vertices. There are more fixpoint kernels in Algorithm 3 all marked with an asterisk. The elimination allows localization of cycles and computation of their means in a straightforward manner (line 6): we simply follow the propagation of edges starting from any not eliminated vertex. The minimal among them can subsequently be found by employing parallel reduction [7] with `min` operation.

Upon finding the cycle with minimal mean (which for technical reasons has to be agreed on by all vertices: line 9) we can actually start rebuilding the graph. With the first backward reachability (line 10) we undo the elimination of the path vertices within the component of the minimal cycle, hence the second backward reachability (line 11) is started from this component and is iteratively applied until all vertices are connected, one breadth-first search layer at a time.

Finally, the SPF subprocedure is easy to parallelize. Using two *val* vectors, alternating the two in odd and even iterations, allows us to perform all updates of values in a single parallel step (as there is no danger of race between threads, see lines 2, 3 and 14). Also realization of the kernel `valuePropagate` on line 14 that propagates the change of *val* from the vertex on minimal cycle with smallest index (*source*), was only a minor modification of backward reachability procedure.

3.2   Graph Primitives and Data Structures

We first focus on the data structures that are used during the computation. The graph representation itself has been described in Section 2.3. In order to keep low space profile, we store the policy graph $\mathscr{G}_\pi$ in a vector of n elements containing indices of the M_c array that uniquely specifies what edge leads to the successor of a given vertex. Also the first few bits of every element are reserved to flags. For example there is a flag marking what vertices have been removed during elimination. Cycles are stored in *gCycles* using two 32-bit values, one for the index of its source vertex and second for the mean weight. Finally, the vector *gPredInfo* is used to store a partial information about the predecessors within one 32-bit value. The first 16-bits are for the number of predecessors and the last 16-bits for the *local* index of the first predecessor.

While most of our kernels are only minor modifications of previously published data-parallel graph primitives (see e.g. [2]), they are crucial for the overall efficiency of our data-parallel implementation, and therefore, we describe some of those modifications in detail. The `GpiPreprocess` primitive was devised for a simple reason: there is no efficient way to propagate along backward edges in the policy graph. Using forward edges would require deployment of as many threads as there are unseen vertices. Searching among backward edges those in the policy graph, on the other hand, suffers from the fact that there are often as many edges that do not belong to the policy graph. Hence both these approaches are approximately equally inefficient. The improvement we have proposed first passes the whole graph $\mathscr{G}$ storing correctly the information about predecessors within the policy graph into *gPredInfo*. This preprocessing allows to virtually skip inspection of vertices with no predecessors in the policy graph and also to jump at the first edge that belongs to the policy graph as can be observed from the pseudo-code of `valuePropagate` in Algorithm 4. There we store in the local variable `counter` the number of predecessor

---

Algorithm 4: `valuePropagate`$(\mathscr{G} = (M_n, M_c), \mathscr{G}_\pi = M_\pi, val, \lambda, \text{source}, gPredInfo)$

---

1 index $\leftarrow$ threadId
2 prop $\leftarrow$ `false`
3 counter $\leftarrow$ $gPredInfo[\text{index}].\texttt{getNum}()$
4 pred $\leftarrow$ $M_n[\text{index} + gPredInfo[\text{index}].\texttt{getFirst}()]$
5 **while** counter > 0 **do**
6   edge $\leftarrow$ $M_c[\text{pred}]$
7   **if** index $= M_c[M_\pi[\text{edge.to}]]$ **then**
8   counter $--$
9   **if** edge.to $\neq$ source **then**
10 $val[\text{edge.to}] \leftarrow val[\text{index}] + \text{edge.weight} - \lambda$
11   prop $\leftarrow$ `true`
12   pred $++$
13 **if** prop **then**
14   fixPoint $\leftarrow$ `false`

---

of the vertex assigned to this thread (line 3) and can skip the cycle if counter is zero. Together with the jump to the first actual predecessor (see line 4) this improvement alone has led to fivefold speedup of `valuePropagate` including the cost of preprocessing.

Details on remaining kernels are as follows. `elimination` kernel is actually the *trimming* primitive (see [2]) augmented similarly as the `valuePropagate` with the information about predecessors. A simple while loop (it is executed only on vertices on some cycle) for identification of cycle source and its mean is implemented in the `cycleIdentification` kernel. And finally the `connectGpi` performs backward reachability from the component of the minimal cycle, and it utilizes a flag `propagate` which marks the currently active breath-first layer. Only the threads that have a vertex with the `propagate` flag do propagate and so less threads needs to be dispatched.

3.3 SCC Decomposition Extension

There are several reasons why to prepend SCC decomposition before a CUDA accelerated OCM algorithm. First of all, the algorithm requires the input graph to be strongly connected and thus we have to add the Hamiltonian cycle. Not only is this operation costly, it also adds more edges into the graph, further prolonging the computation. Furthermore, even though the parallel algorithms for SCC decomposition have rather high asymptotic complexity ($\mathscr{O}(n(n+m))$), we were able to implement data-parallel SCC decomposition which is considerable faster than the optimal sequential algorithm [2]. And most importantly, it would allow the computation to be executed concurrently on all SCC components, which further improves the running time provided that the components are much smaller than the whole graph.

The technique to run a kernel on multiple *regions* within a graph and to restrict its effect to respective regions was thoroughly described in [2] and we will thus concentrate on the parts related to computation over decomposed graphs specific for our OCM algorithm. First of all the Proposition 2.3 states that the optimal cycle mean of the whole graph needs to be found as the minimal among all components. This observation raises two problems, first, how to choose the minimum, and second, where to store the

---

Algorithm 5: Region-specific minimum voting

---

1 myCycle ← (source, mean = (weight/length))
2 **while** `true` **do**
3   comCycle ← $cycles$[myRegion]
4   **if** comCycle.mean ≤ myCycle.mean **then break**
5   `atomicCAS`(&($cycles$[myRegion]), comCycle, myCycle)

---

component-specific λ values during the computation (which also has to be agreed on).

Selecting the minimal cycle mean during the computation on multiple regions is particularly problematic as the vertices of one component are not clustered together. It would actually require first to *split* [21] the vector according to the region identification and then perform *segmented* reduction [7], both complicated and expensive operations. Fortunately, we have observed that often very few cycles are found and consequently only a few values are candidates for the minimum. Thus we were able to use the `atomicCAS` operation as shown in Algorithm 5 without any significant time expense.

Also the termination needs to be modified to work on two levels. The global termination occurs when computation is finished on all strongly connected components. But it is also important to prevent execution of kernels on components where we have already found the OCM. Since then less threads needs to be deployed and less candidates compete in the minimum voting. For that purpose we have inserted a kernel that unsets the flag *work* for all vertices in inactive components between lines 2 and 3 of Algorithm 3. Finally, we have optimized the overall amount of work by unsetting the *work* flag of all single vertex components prior to the actual OCM computation.

4 Experimental Evaluation

Since the prime objective of our research was to accelerate performance analysis based on computation of optimal cycle mean, we have compared sequential and data-parallel algorithms mainly on models of communicating distributed systems. The state space of all possible configurations of a given system forms a graph with cost function (representing for example resource consumption) labelling edges of that graph. Both modelling of the system and generation of its state space was facilitated by the enumerative model checker DiVinE [3], extended with the capability to analyse performance of input models.

The sequential OCM algorithms Howard's and YTO were also implemented within DiVinE. Thus the evaluation of our GPU Howard's algorithm is conducted by comparing its running time against these two sequential algorithms (which are often considered to be the fastest [10]). The graph representation, while primarily targeting the vector processing architecture of GPU, is also particularly suitable for CPU due to its cache-efficient characteristics.

All experiments were executed on a CUDA-equipped Linux workstation with an AMD Phenom II X4 940 Processor @ 3GHz, 8 GB DDR2 @ 1066MHz RAM and NVIDIA GeForce GTX 480 GPU with 1.5 GB of GDDR5 memory. All codes were compiled with `-O3` optimization using gcc version 4.3.2 and nvcc version 3.1 for CPU and GPU code, respectively.

Our approach to performance analysis allows us to compute quantitative characteristics of distributed systems where clients comply to a distinct sets of behavioural patterns (scenarios of expected behaviour). Furthermore, we can state how many clients of that particular scenario appear in the system and thus we can estimate the load of a part of the system (a server for example) in an execution based solely on the
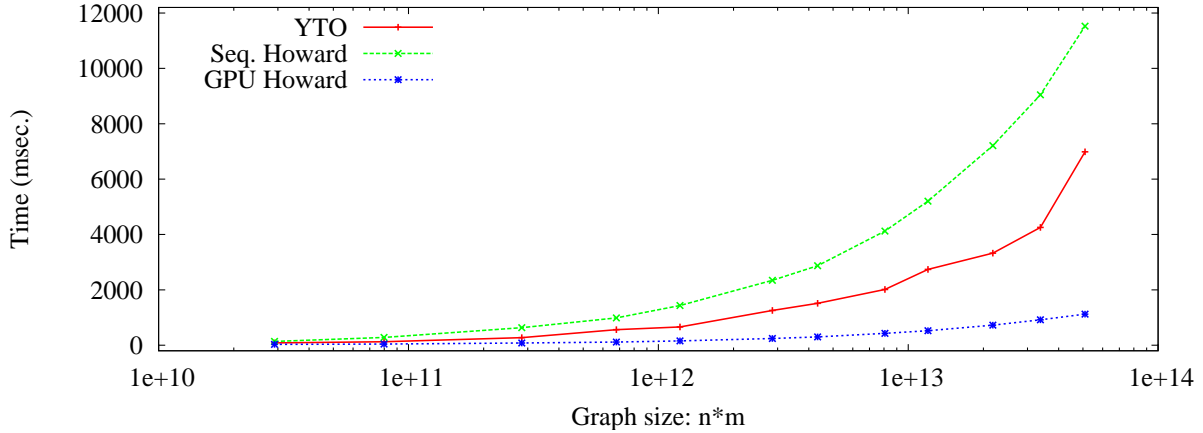
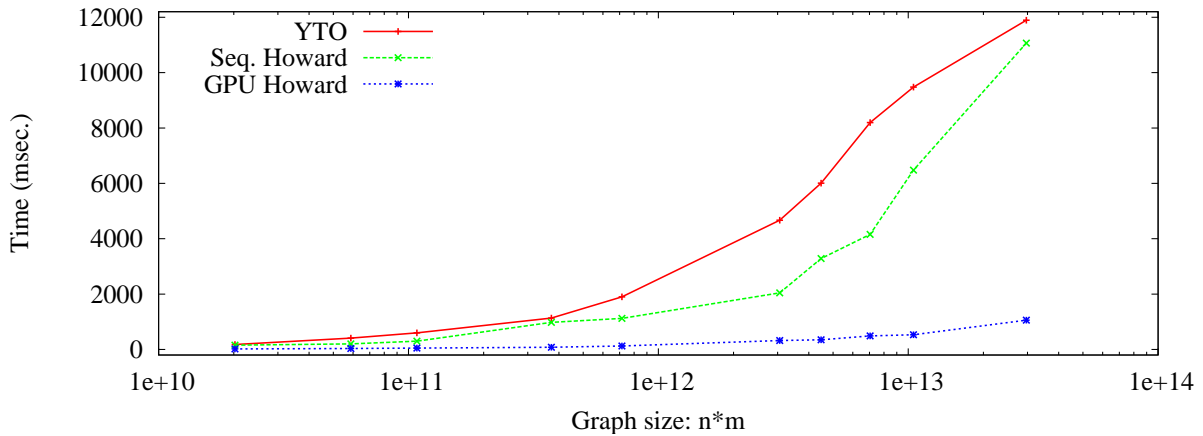Figure 3: Plot for the server-free system.

Figure 4: Plot for the system with a server.

information of how many clients of what scenario there are. As stated in Section 2.2 we have used a similar approach in our running example for analysis of CPU utilization in an online PDF editor (as we could for any distributed client/server application in general). After modelling the system using the client scenarios we have computed the maximal cycle mean in the transition system generated from the synchronous composition of those clients. This value was equal to the worst sustainable CPU utilization during any infinite run of the modelled system. Thus we were able to estimate the number of clients that would be able to fully utilize the server.

In order to measure scalability of the GPU algorithm we have constructed two distributed client/server system *templates* from which a user can generate system models by deciding on the number of client for each scenario. In the first system, there is no server present and the actions of clients are left to be interleaved nondeterministically. The second systems contain a server and the clients have to communicate with the server (and only one client can access the server at a time). The OCM of such systems then represents the average system load inflicted on the environment.

We have performed various tests on both templates to measure scalability of all the algorithms and plotted the findings in two figures. In Figure 3 there are the results for system without a server and in Figure 4 the results for system with a server. The *x* axis are in logarithmic scale and represents the

size of the graph in $n * m$ as all the sequential OCM algorithms are asymptotically in $\mathcal{O}(nm)$ or worse. Actual sizes of the analysed graphs range from 50 thousand vertices and 400 thousand edges to approximately 2 million vertices and 25 million edges. The plots show very clearly that while both YTO and the sequential Howard's algorithm struggle with preserving their running times as the graphs grow bigger, our GPU implementation is capable of performing the OCM computation in reasonably small time. On larger instances the GPU rarely fails to provide a fivefold speed up compared to the better of the two sequential algorithms. It is worth noting that while the CPU algorithms were deterministic and in different runs exhibited indistinguishable behaviour, the GPU implementation behaves nondeterministically due to the nature of the massive parallelism. For that reason we executed every test ten times and the result displayed in the plots is the median of all trials.

Although primarily targeting acceleration of OCM computation for performance analysis we feel obliged to admit that on graphs from other applications was our data-parallel implementation much less successful. We have conducted several experiments with US traffic network graphs, random graphs and various graphs from the DIMACS challenge and were never able to outperform the YTO algorithm. On these graphs the YTO performed only a very few iterations which we attach to the fact that the OCM of these algorithms was often very close to the minimal edge weight.

5   Conclusion

We have proposed data-parallel acceleration of an OCM algorithm within several consecutive steps. First we have evaluated all existing classes of OCM algorithms with respect to their predisposition for vector processing. Subsequently, we have described thoroughly the Howard's algorithm which was found most appropriate for GPU acceleration and devised its data-parallel version. Specifics of the implementation together with selected data-parallel graph primitives were then detailed, e.g. the incorporation of SCC decomposition and the concurrent execution of the OCM algorithm on all strongly connected components.

The primary motivation behind GPU implementation of OCM algorithms was the acceleration of performance analysis of distributed communication systems. That we have evaluated experimentally by constructing two scalable client/server systems based on distinct scenarios of the clients finding our data-parallel algorithm capable of providing performance analysis in negligible time. Although competitiveness of the GPU algorithm on other types of graphs is questionable, we have reported a steady fivefold speed up on performance analysis graph against all other algorithms.

References

[1] J. Barnat, P. Bauch, L. Brim & M. Češka (2010): *Employing Multiple CUDA Devices to Accelerate LTL Model Checking*. In: *Proceedings of the 16th International Conference on Parallel and Distributed Systems*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 259–266, doi:10.1109/ICPADS.2010.82.

[2] J. Barnat, P. Bauch, L. Brim & M. Češka (2011): *Computing Strongly Connected Components in Parallel on CUDA*. In: *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*, IEEE Computer Society, pp. 541–552.

[3] J. Barnat, L. Brim, M. Češka & P. Ročkai (2010): *DiVinE: Parallel Distributed Model Checker (Tool paper)*. In: *Proceedings of joint HiBi/PDMC workshop*, HiBi/PDMC 2010, IEEE, pp. 4–7.

[4] S. M. Burns (1991): *Performance Analysis and Optimization of Asynchronous Circuits*. Ph.D. thesis, California Intitute of Technology, Pasadena.

[5] S. M. Burns, H. Hulgaard, T. Amon & G. Borriello (1995): *An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems*. IEEE Transaction on Computers 44(11), pp. 1306–1317, doi:10.1109/12.475126.

[6] J. Chaloupka (2006): *Distributed Algorithms for the Minimum Mean Weight Cycle Problem*. Master's thesis, Masaryk University, Faculty of Informatics, Brno.

[7] S. Chatterjee, G. E. Blelloch & M. Zagha (1990): *Scan Primitives for Vector Computers*. In: *Proceedings of the 2nd International Conference for High Performance Computing, Networking, Storage and Analysis (SC '90)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 666–675.

[8] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan & R. F. Werneck (2010): *Shortest Path Feasibility Algorithms: An Experimental Evaluation*. Journal of Experimental Algorithmics 14, pp. 118–132.

[9] (April 2011): *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 4.0.*

[10] A. Dasdan & R. K. Gupta (1997): *Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17, pp. 889–899, doi:10.1109/43.728912.

[11] A. Dasdan, S. S. Irani & R. K. Gupta (1999): *Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems*. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, ACM, pp. 37–42, doi:10.1145/309847.309862.

[12] P. Harish & P. Narayanan (2007): *Accelerating Large Graph Algorithms on the GPU Using CUDA*. In: *High Performance Computing (HiPC'07)*, Lecture Notes in Computer Science 4873, Springer Berlin / Heidelberg, pp. 197–208, doi:10.1007/978-3-540-77220-0_21.

[13] W. D. Hillis (1987): *The Connection Machine*. Scientific American 256(6), pp. 108–115, doi:10.1038/scientificamerican0687-108.

[14] R. A. Howard (1960): *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

[15] K. Ito & K. K. Parhi (1995): *Determining the Minimum Iteration Period of an Algorithm*. The Journal of VLSI Signal Processing 11(3), pp. 229–244, doi:10.1007/BF02107055.

[16] L. Kleinrock (1975): *Queueing Systems, Volume 1: Theory*. Wiley-Interscience, New York.

[17] E. L. Lawler (1976): *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart, and Winston, New York, NY.

[18] R. Lu & C. Koh (2006): *Performance Analysis of Latency-Insensitive Systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25(3), pp. 469–483, doi:10.1109/TCAD.2005.854636.

[19] A. Mathur, A. Dasdan & R. K. Gupta (1998): *Rate Analysis for Embedded Systems*. ACM Transaction on Design Automation of Electronic Systems 3(3), pp. 408–436, doi:10.1145/293625.293631.

[20] M. K. Molloy (1982): *Performance Analysis Using Stochastic Petri Nets*. IEEE Transactions on Computers 31(9), pp. 913–917, doi:10.1109/TC.1982.1676110.

[21] S. Patidar & P. J. Narayanan (2009): Scalable Split and Gather Primitives for the GPU. Technical Report IIT/TR/2009/99, Centre for Visual Information Technology, Hyderabad, INDIA.

[22] C. V. Ramamoorthy & G. S. Ho (1980): *Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets*. IEEE Transaction of Software Engineering 6(5), pp. 440–449, doi:10.1109/TSE.1980.230492.

[23] R. Tarjan (1971): *Depth-First Search and Linear Graph Algorithms*. In: *Proceedings of the 12th Annual Symposium on Switching and Automata Theory*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 114–121, doi:10.1109/SWAT.1971.10.