Executing PRAM Programs on GPUs

Jürgen Brenner, Jörg Keller and Christoph Kessler

Original Publication:

Jürgen Brenner, Jörg Keller and Christoph Kessler, Executing PRAM Programs on GPUs, 2012, Procedia Computer Science, (9), 1799-1806.
http://dx.doi.org/10.1016/j.procs.2012.04.198

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1799 – 1806

Procedia Computer Science

International Conference on Computational Science, ICCS 2012, PAPP Workshop

Executing PRAM Programs on GPUs

Jürgen Brenner[a], Jörg Keller[a,*], Christoph Kessler[b]

[a] *Fac. Mathematics and Computer Science, FernUniversität in Hagen, Germany*
[b] *Dept. Computer and Information Science (IDA), Linköping University, Sweden*

Abstract

We present a framework to transform PRAM programs from the PRAM programming language Fork to CUDA C, so that they can be compiled and executed on a Graphics Processor (GPU). This allows to explore parallel algorithmics on a scale beyond toy problems, to which the previous, sequential PRAM simulator restricted practical use. We explain the design decisions and evaluate a prototype implementation consisting of a runtime library and a set of rules to transform simple Fork programs which we for now apply by hand. The resulting CUDA code is almost 100 times faster than the previous simulator for compiled Fork programs and allows to handle larger data sizes. Compared to a sequential program for the same problem, the GPU code might be faster or slower, depending on the Fork program structure, i.e. on the overhead incurred. We also give an outlook how future GPUs might notably reduce the overhead.

Keywords: PRAM programming, GPU computing

1. Introduction

The Parallel Random Access Machine (PRAM) [1] is a well-established model to study the paradigms and complexities of parallel algorithms. In order to extend this study from asymptotic complexities to constant factors and to allow practical exercises in classes, the high-level programming language Fork has been formulated [2]. A key concept in Fork are processor groups that execute code blocks synchronously, and the possibility to hierarchically and recursively partition those groups [3]. To execute Fork programs, a complete tool chain for the SB-PRAM instruction set has been developed [3], the SB-PRAM being a prototype hardware realization of a PRAM. The tool chain includes a compiler, a simulator that emulates the SB-PRAM on a sequential computer, and some debugging and performance tools. As the SB-PRAM never left prototype stage and the hardware is now defunct, running Fork programs means to use the simulator. However, for larger problem instances, the simulator runtimes are very long, so that efforts for a parallel simulator have been attempted, with modest success [4, 5].

Newer graphics processing units (GPUs) are massively parallel processors and can also be programmed for general purpose computing (GPGPU). In particular, Nvidia GPUs can be programmed in CUDA, where a warp, i.e. a group of currently 32 threads, can work (virtually) synchronously on the fast but small on-chip shared memory [6, 7] like in a PRAM. Thus, GPUs might be a target for practical PRAM programming. This is also acknowledged in [8, 9], where a manual adaptation of PRAM algorithms for use with GPUs is proposed. In [10], a source-to-source transformation

*Corresponding author
Email addresses: joerg.keller@fernuni-hagen.de (Jörg Keller), christoph.kessler@liu.se (Christoph Kessler)

doi:10.1016/j.procs.2012.04.198

from OpenMP to CUDA is proposed where techniques similar to ours, such as split of parallel regions into different kernels, are discussed.

In this paper, we investigate whether Fork programs can be (semi-)automatically transformed into CUDA programs and how much overhead is incurred by arbitrarily large group sizes, dynamic group partitioning, and use of large memories. Our conclusions are that automatic transformation is possible, but that further work might be necessary to tune performance. Our preliminary experiments indicate that the CUDA program on a GPU achieves a notable speedup of almost 100 over the sequential PRAM simulator running on the CPU, and depending on the application even over a sequential program solving the same problem on the CPU. We analyze the overhead incurred, mainly kernel synchronizations, and conclude that future GPUs with tighter integration to the CPU could largely reduce this overhead, so that beyond classroom use, Fork programming might also become more competitive in real life.

The remainder of this article is structured as follows. In Section 2, we briefly recall relevant facts about PRAMs, GPUs, and their programming. In Section 3, we describe major issues in the transformation of Fork programs to CUDA. In Section 4, we provide preliminary experimental results, while Section 5 provides a conclusion with an outlook on future work.

2. PRAMs and GPUs

PRAM. A *Parallel Random Access Machine (PRAM)* [1, 3] is a parallel computer with p processing units working synchronously on a shared memory with small constant access time. Each processor has its own program control (MIMD principle) but the whole machine is clocked globally and works synchronously such that each PRAM processor executes exactly one PRAM operation (arithmetic, memory access, branch etc.) per clock cycle. In principle, the hardware thus performs an implicit global barrier synchronization and memory fence after every instruction.

Different PRAM variants have been defined to model constraints on concurrent access to shared memory (such as exclusive vs. concurrent read or write). The possible resolutions of concurrent write access conflicts (such as arbitrary, priority or combining concurrent write), have implications for how fast certain problems can be solved.

Thanks to the instruction-level synchronous execution, PRAM computations are deterministic as long as the mechanism for resolving concurrent write access conflicts is deterministic. Also, PRAMs offer strict memory consistency, the strongest memory consistency model possible. In many cases, the PRAM's natural synchronization makes explicit user-programmed low-level synchronization, such as mutual exclusion or barrier synchronization, unnecessary for protecting cross-processor data dependencies from race conditions, in contrast to the common asynchronous thread-based shared memory programming model as provided by Pthreads or OpenMP. Thus, the PRAM model abstracts from many difficulties that decrease parallel performance in real parallel machines, and serves to study algorithmic efficiency. Several projects investigated the realization of the PRAM model in hardware, see e.g. [3, 11, 12, 13].

Fork. Fork is a C-based high-level programming language that was created to allow for experimentation with PRAM algorithms, e.g. to study scalability or to get an impression about the constant factors in their time complexity.

In a Fork program, an (active) *group* of PRAM processors executes a piece of code synchronously; initially all processors form one large group. Variables can be declared as (group-wide) *shared* or *private*, and thus are either created once for the whole (declarating) group or once for each group member, respectively. In case of statements where control flow depends on a private variable, such as a two-sided IF-statement where the processor ID (expressed by the $ symbol) is compared to some constant

```
if ($ < 2) function1(); else function2();
```

the group is split and temporarily deactivated while the subgroups synchronously execute their different program subpaths. After the statement, the subgroups are automatically synchronized and again form the previous group, which resumes execution. Thus, at any point in time, the group hierarchy forms a tree, where the leaves are the currently active groups. A code example is depicted in Fig. 1. There, a group of processors is first reading in a shared array and then splits into two subgroups, each writing a different value. Finally, the two subgroups merge again to form the previous group and execute a final assignment.

Fork also provides an asynchronous mode of execution where the group structure is read-only and the automatic group splitting and synchronization are temporarily disabled. This mode is appropriate for code parts where synchronous execution is not needed for correctness or where the higher flexibility of asynchronous execution can be

```
sh int a[N]; // shared array
pr int loc; // private var
pr int myID = $; // proc.  ID
...
loc = a[myID];
if(expr(myID)) a[myID] = myID; else a[myID]=loc+1;
loc = a[myID] + a[(myID+1)%N];
```

Figure 1: Fork code example.

advantageous, e.g. for load balancing purposes, for sequential code, and for I/O operations. The execution mode (synchronous or asynchronous) is statically associated with code blocks in Fork programs, and there are special language constructs to mark these blocks and switch execution mode.

For further details, we refer the interested reader to [3].

GPUs. A *Graphics Processing Unit (GPU)* is a massively parallel processor on a chip. It mainly serves for rendering images, but can also be used for general purpose computations (GPGPU computing). Figure 2 provides a very simplified view of a GPU architecture. Several *Streaming Multiprocessors (SM)* form the processing hardware. Each SM is equipped with a number of hardware-multithreaded processing units called *scalar processors (SP)* that work synchronously on a small, fast shared memory (16 to 48 kByte).

All SMs can access the large but slow global memory of the GPU, which is also used for data transfer to and from the host, i.e. the CPU. The global memory (together with a constant memory and a so-called local memory, which we will not discuss further) are realized by dynamic random access memory (DRAM) on the graphics card, external to the GPU chip.

Not all Nvidia processor architectures provide a stack, so not all support recursion in hardware. Also, not all Nvidia architectures provide caching of the global memory. If they do, there is an L1-cache for each SM, but those caches are not coherent. Furthermore, there is a single L2-cache which is shared by all SMs. The Fermi architecture e.g. provides caches and stack, while the Tesla architecture does not.

CUDA. Nvidia provides the CUDA C programming language for its GPUs, where programs are executed in master-slave style. A host program is started on the CPU, then a so-called *kernel* is spawned on the GPU, which provides a return value to the host program after completion. This is repeated until the host program terminates. A kernel is executed on the GPU by a so-called *grid*, i.e. by one or several blocks of threads. The number of threads per block (at most 512) and the number of blocks can be chosen by the programmer.

A set of (up to 32) threads that are executed simultaneously and synchronously on the processing units of a SM is called a *warp*. Warps are executed like SIMD operations; where control flow diverges in a warp, the warp executions along the different paths are serialized, i.e. processors not participating in the current path are masked out. Each block is scheduled for execution on an SM. So if the block comprises more threads than fit into one warp, the block's threads are executed warp by warp; the dynamic scheduling of warps is done by a hardware scheduler and not under user control.

If a grid comprises more blocks than SMs are available, the blocks are scheduled in arbitrary order. As soon as an SM gets available by completion of its current block, it executes another block.

For further details, the interested reader is referred to [6, 7].

3. Fork-to-CUDA Transformation

In order to derive guidelines for the transformation, we start by considering a Fork program that only consists of one function, does not perform recursion, and where the initial group is not split. All processors of the group e.g. synchronously access a shared array with read, write, and another read, cf. left half of Fig. 3. One would think that this function can be directly transformed into one kernel consisting of a grid with one block, and where all threads work on the shared memory of an SM. However, we already run into problems here. First, the number of processors

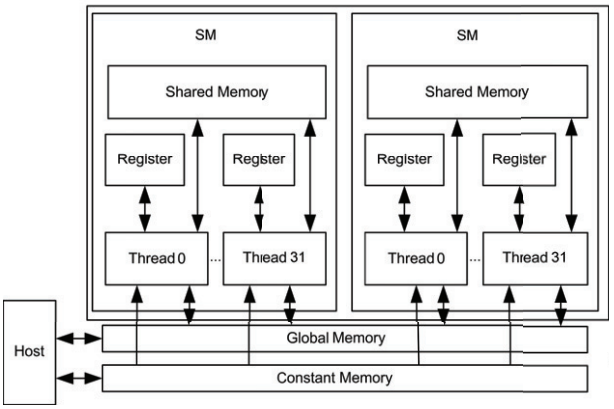

Figure 2: Memory Architecture of an NVidia GPU.

might be larger than 512, the largest possible size of one block. This means that the grid must consist of more than one block, with the possibility that two blocks are executed simultaneously on two different SMs. Thus, placing the shared array in the shared memory of an SM is not possible, and we have to place all shared variables in the global memory. We will also do this with private variables, on the one hand because the shared memory is very small, on the other hand because values of private variables sometimes must be transferred from one kernel to the next, see below.

```
void forkfunc(void) {
  sh int a[N]; // shared array
  pr int loc; // private counter var
  ...
  loc = a[myID];
  a[($+1)%N] = loc + $;
  loc = a[myID];
  ...
}
```

```
void Hostfunc(...)  {
  Allocatevars(...);
  Devicefunc1<<<1,N>>>(...);
  Sync(...);
  Devicefunc2<<<1,N>>>(...);
}
```

Figure 3: Fork code transformation example.

For groups with more than 32 processors, we also run into the problem that not all threads are running synchronously, either because a block comprises more than 32 threads, or because more than one block is used, and the blocks are scheduled one by one to the same SM. Therefore, the threads must be synchronized after the write access to the shared array, and before the following read access. However, synchronization cannot be done within the kernel, as synchronization between blocks may lead to a deadlock if one block has not yet started because it is scheduled to the same SM that is currently busy with the block that tries to synchronize. Also it is not possible to call a synchronization by the host program from the slave program. Therefore, the only possibility is to split this Fork function into two parts that are executed in separate kernels by the host program, cf. right part of Fig. 3. The first kernel `Devicefunc1` comprises the read and write access, and the second kernel `Devicefunc2` comprises the second write access. The synchronization between them is necessary to ensure that the first kernel has completely terminated and all caches have been flushed, i.e. that all writes are written to the global memory, before any of the reads from the second kernel is executed.

Note that in [14], a similar approach of splitting a function is used. However, it seems that they refrain from using a synchronization in-between, which seems necessary if several kernels can be active simultaneously, as is the case in the Fermi architecture. Also, most of the further problems discussed here are not of interest there, as a global cellular automaton comprises a set of finite-state automata, whereas a Fork program also has to deal with recursion and the

like.

Splitting a function into several kernels also explains why local variables are placed in the global memory. If a function is split, then the values of local variables must be transferred from one kernel to the next. This can only occur via the global memory, which is the only memory still valid after termination of a kernel. Even if we would assume that shared memory contents are not altered, we have no influence on the scheduling of blocks to SMs, and so we could not ensure that the PRAM processors with the same IDs are mapped to the same SM in both kernels.

A similar situation occurs when a function call occurs in a Fork function. As not all GPUs support a stack, and as CUDA device function visibility is restricted to its source file, the function call cannot be implemented within a device function. Also in this case, the Fork function is split, and the host program calls two kernels: one that contains the code prior to the function call, the other contains the code after the function call. The function call itself is implemented via a call in the host program (at the place where the synchronization occured above) to the host function implementing the called function.

In the situations considered so far, all processors have formed a single group. If we now consider the code from Fig. 1, the group of processors is split into two subgroups, because the condition in the IF-statement depends on a private variable, in this case on the processor ID. The different subgroups execute the `then` and `else` branches of the IF-statement, respectively, and then are merged again into the previous group, and perform the final assignment. The code of the resulting host program can be seen in Fig. 4. The kernel `Devicefunc1` contains the first assignment to variable `loc` and the evaluation of the condition. Then, function `ComputeSubgroups` of the host program computes how many of the N PRAM processors take the then path ($N1$) and how many take the else path ($N - N1$). The synchronization after `Devicefunc1` is necessary to ensure that all condition results have been written before they are evaluated by the host program. The host function `ComputeSubgroups` also updates the group structure in global memory, i.e. transfers data to the global memory. For each subgroup, the participating processors must be stored, their new group-local IDs have to be assigned, and so on. Now the kernels `Devicefunc2then` and `Devicefunc2else` are started with the respective number of threads, and as the subgroups are independent, they can be executed in parallel. After the kernels, a synchronization is necessary to ensure that both kernels have completed and all data values are written to the global memory. The group structure is updated in host function `RestoreGroup` to restore the previous group, and the final assignment to variable `loc` is performed in the kernel `Devicefunc3`.

```
void Hostfunc(...)  {
  Devicefunc1<<<1,N>>>(...); // includes condition evaluation
  Sync(...);
  ComputeSubgroups(N,&N1,...);
  Devicefunc2then<<<1,N1>>>(...);
  Devicefunc2else<<<1,N-N1>>>(...);
  Sync(...);
  RestoreGroup(...);
  Devicefunc3<<<1,N>>>(...);
}
```

Figure 4: Fork group split example.

Note that if the `then` or `else` branches would contain further statements, then further kernels would be necessary, as the statements in both branches write to shared variables, so that a synchronization in the subgroups would be necessary prior to follow-up statements.

The `fork` statement is used in Fork to explicitly split a group into several subgroups, by having each processor evaluate two private expressions that return its new group and member index. When a `fork` statement is reached, the control flow is handled similarly to an IF-statement. The difference is that in a `fork` statement, more than two subgroups can be formed, so that the updates to the group structure are more involved.

As a last example, we will consider a loop. If the loop body can be implemented in a single kernel (shared exit condition, no writes to shared variables, no `if` or `fork` statements), it does not need special treatment. However, if one of the above features occurs, then the loop body cannot be implemented by a single kernel. In this case, the loop

control is realized in the host program, which executes the kernels of the loop body as described above. In addition, the exit condition is computed by a kernel and evaluated by the host program, which updates the group structure in the sense that processors that exit the loop are disabled from the active group. Consider the example in Fig. 5. Here, the write to a shared array splits the loop body in two kernels. Therefore the while loop is executed by the host program. The first kernel `Devicefunc1` contains the assignment to the shared variable. The second kernel `Devicefunc2` contains the increment of the loop counter and the evaluation of the loop exit condition. The function `Updategroup` disables all processors for which the condition $i < 10$ is false. The function `lastexit` returns a non-zero value when all processors of the group have exited. Then the loop is left and the group restored. Generally, all host functions working on the group structure are somewhat expensive because they require a synchronization, and they transfer data between global memory and host memory.

```
sh int a[N]; // shared array
pr int i = 0; // private loop var
...
do {
   a[myID + i] = myID;
   qquad i++;
} while(i < 10);
```

```
do {
   Devicefunc1<<<1,N>>>(...);
   Sync(...);
   Devicefunc2<<<1,N>>>(...);
   Sync(...);
   Updategroup(...);
} while(!lastexit(...));
RestoreGroup(...);
```

Figure 5: Fork loop transformation example.

There are more issues to be considered to obtain a complete set of transformation rules. For example, the asynchronous mode might be handled like the synchronous mode with the exception that writes to shared variables do not split a function, and the sequential mode, in which only one processor is active, can be implemented on the host. Currently, we assume that I/O operations will only occur in the sequential mode, and thus need no further treatment.

4. Experimental Results

So far, our implementation of the transformation tool only consists of a runtime library of functions that provide code for initialization, data structures for group management and so on. Hence, we have tested the transformation with three applications that have been hand-transformed according to the transformation rules, without further optimization: sieve of Eratosthenes, quicksort, and matrix multiplication. We compare the performance of our transformed code both with compiled Fork code executed in the sequential PRAM simulator pramsim on the CPU, and with sequential versions of the applications running on the CPU. As platform, we use an Intel Core2 CPU (E6600, 2.4 GHz) and an Nvidia GeForce GT 430 graphic card, with a GPU equipped with 2 SMs and a Fermi architecture.

The runtimes for Eratosthenes and quicksort are depicted in Fig. 6. In the left part we can see that the transformed sieve program on the GPU is faster than both the sequential program and the simulator on the CPU. The speedup with a growing number of threads is notable, although the runtime only shrinks to about 70% when doubling the threads. The speedup compared to the sequential simulator is almost 100, so that runtimes of hours are now reduced to below a minute, which greatly increases usability in classes.

In the right part we see that the transformed parallel quicksort on the GPU is slower than a sequential quicksort on the CPU. This is not surprising as the amount of work in sorting 1 million numbers is small compared to the overhead of repeated group split. We consider this still an advantage as the compiled Fork code on the pramsim was not able to handle such a data size. With a growing number of threads, a runtime reduction is also notable, indicating that the transformed code is scalable.

To analyze where the overhead in the CUDA program for quicksort comes from, we measured the time spent in kernel synchronizations, and found that they comprise about 31% of the complete runtime for 4608 threads and 1 million elements to be sorted. This poses an extreme example where the groups do not perform much work beyond splitting. An improvement of this situation is to expected by faster synchronization mechanims, see the next section.

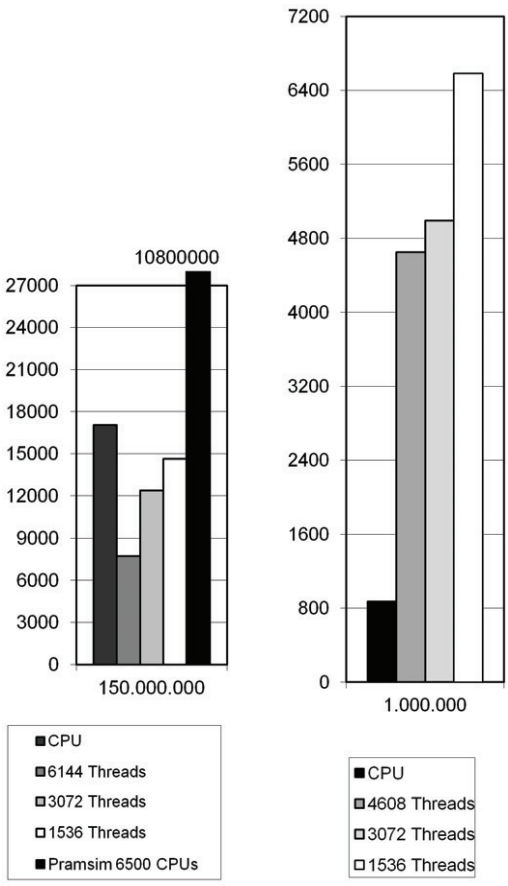

Figure 6: Runtime of Example Applications. Left: Sieve of Eratosthenes. Right: Quicksort.

For comparison, we also implemented an application from the field of linear algebra: the multiplication of two $N \times N$ matrices. In Table 1, we see the runtimes for the sequential simulator running the compiled Fork code on a CPU and a GPU executing the converted code. Not surprisingly, the GPU is much faster than the simulator on the CPU, as this code is so regular that no kernel split occurs, and thus serves as a kind of best case. The speedup is close to P if the Fork code uses a group of P PRAM processors.

N	P	simulator	GPU
2048	64	4.5	0.1
2048	128	5.5	0.06
4096	128	20.0	0.2

Table 1: Runtimes of $N \times N$ matrix multiplication with P PRAM processors on sequential simulator and GPU, measured in seconds.

5. Conclusion and Outlook

We have presented the transformations for a source-to-source compiler from Fork programs for PRAMs to CUDA C for GPUs, to enable the use of Fork for practical PRAM programming beyond toy examples, with reasonable runtimes. While many improvements are still possible, our preliminary experimental results indicate that for the

simple, non-recursive sieve application, we achieve speedup compared to both the PRAM simulator on a CPU, and a sequential version of the program, while for the highly recursive quicksort, we can cope with data set sizes previously out of reach for the simulator, and "only" suffer a slowdown of 6 compared to a highly optimized sequential version of quicksort.

Future work will comprise reducing the overhead e.g. created in recursive calls, by using stacks introduced with the Fermi architecture, and in implementing a prototype of the source-to-source compiler, to be able to deploy Fork in lab courses. To implement our source-to-source compiler, the Cetus framework might be used as in [10]. Also, a theorem prover such as Coq or Isabelle might be used to check the correctness of the transformation rules. We will also try to extend the compiler to OpenCL, to extend its use from Nvidia GPUs to other manufacturers. As OpenCL resembles CUDA in many facets, the amount of work to invest seems limited. A reduction of the amount of kernel restarts at group splitting and merging points e.g. in parallel divide-and-conquer programs could be achieved by static optimizations such as unfolding of recursive group-splitting constructs, eliminating redundant synchronization points or generating predicated code in the same kernel where appropriate, e.g. in loops with diverging control flow.

As chip density still grows exponentially, following Moore's Law, the near future will bring architectures that accommodate both CPU and GPU on the same chip, such as AMD Fusion (fusion.amd.com). This will provide a much higher bandwidth between CPU and GPU compared to today's PCIe interconnect, a tighter integration and sharing along the CPU memory hierarchy with the GPU, and a much finer task granularity than what is efficiently doable with today's GPUs. We expect that this will improve the performance achievable on such systems for Fork programs translated into CUDA or OpenCL.

Another possible source of optimization is the design of the original Fork programs, which often have been written with a certain space cost model in mind. For example, for the SBPRAM with its strict embedding of private address subspaces into the size-limited global shared SBPRAM memory, it was more space-economic to pass read-only parameters to functions as shared parameters (on the shared stack) rather than as private ones, because this reduces overall memory usage while there is no difference in execution time. This however leads to worse performance with the transformation approach, where private parameters could be passed on private stacks stored or cached in GPU shared memory instead, while shared ones have to stay in GPU global memory.

Acknowledgements

We thank the anonymous reviewers for their helpful comments.

References

[1] S. Fortune, J. Wyllie, Parallelism in random access machines, in: Proc. 10th Symp. Theory of Computing, 1978, pp. 114–118.
[2] C. W. Kessler, A practical access to the theory of parallel algorithms, in: Proc. ACM SIGCSE'04 Symposium on Computer Science Education, 2004.
[3] J. Keller, C. W. Keßler, J. L. Träff, Practical PRAM Programming, Wiley & Sons, 2001.
[4] B. Wesarg, H. Blaar, J. Keller, C. Kessler, Emulating a PRAM on a parallel computer, in: Proc. 21st Workshop on Parallel Algorithms and Architectures (PARS 2007), 2007, pp. 77–89.
[5] J. Keller, C. Kessler, B. Wesarg, Efficient simulation of Fork programs on multicore machines, in: Proc. 23rd Workshop on Parallel Algorithms and Architectures (PARS 2009), 2009, pp. 84–90.
[6] Nvidia, Nvidia CUDA architecture (2009).
[7] Nvidia, Nvidia CUDA C programming guide (v4.0) (2011).
[8] F. Dehne, K. Yogaratnam, Exploring the limits of GPUs with parallel graph algorithms, CoRR abs/1002.4482 (Feb. 2010).
[9] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proc. 23rd IEEE International Parallel and Distributed Processing Symposium, 2009.
[10] S. Lee, S.-J. Min, R. Eigenmann, OpenMP to GPGPU: A compiler framework for automatic translation and optimization, in: Proc. PPoPP, 2009, pp. 101–110.
[11] W. J. Paul, P. Bach, M. Bosch, J. Fischer, C. Lichtenau, J. Röhrig, Real PRAM programming, in: Proc. Euro-Par'02, 2002.
[12] X. Wen, U. Vishkin, FPGA-based prototype of a PRAM-on-chip processor, in: Proc. ACM Computing Frontiers, Ischia, Italy, 2008.
[13] M. Forsell, TOTAL ECLIPSE – An Efficient Architectural Realization of The Parallel Random Access Machine, Parallel and Distributed Comput., Ed. A. Ros, IN-TECH, Wien (2010) 39–64.
[14] B. Milde, N. Buescher, M. Goesele, Implementing the Global Cellular Automata on CUDA, in: Proc. 24th Workshop on Parallel Algorithms and Architectures (PARS 2011), 2011, pp. 28–37.