Technische Universität Berlin
Fakultät IV - Elektrotechnik & Informatik
Seminar Recent Advantages in Multicore Systems

Recent advantages in increasing GPU efficiency

submitted by:
Michael Winkelmann (338024)
(miwinkel@mailbox.tu-berlin.de)

Sommersemester 2012

Abstract

Performing general purpose computations on graphics hardware becomes more and more popular and available on modern hardware architectures. In this seminar paper, three papers of recent proceedings of *HPCA 2012* and *Innovative Parallel Computing* are reviewed and compared. Each paper addresses a different issue in terms of computation efficiency and aims for a better usage of available resources. This leads to more performance, lower power consumption and a better user experience.

1 Introduction

Modern GPUs are not static fixed pipelined graphic-specific processors, in fact they are flexible programmable general purpose stream processors. GPUs are best suited for parallelizable tasks with a high arithmetic density which operate on large sets of homogenic data. Commonly, those tasks have a large portion of arithmetic tasks and a small portion of reading and writing operations. For example, those tasks often occur in scientific computing as well as in multimedia and gaming applications. Besides typical graphic applications, current GPUs are able to perform database operations [BS10], cryptographic algorithms [SG08] as well as scientific simulations [Gil09], mostly with a superior performance over CPUs and lower power consumption [MDZD09]. Performing general purpose computations on GPUs is abbreviated as GPGPU (**G**eneral **P**urpose Computation on **G**raphics **P**rocessing **U**nit).
Internally, a typical GPU consists of up to thousands of small arithmetic logical units (ALUs). In contrast, CPUs consist of a few large units which usually share a common cache. They are more flexible for tasks where branching occurs and treated data is heterogeneous, such as traversing hierarchal data structures. Moreover, the CPU has higher clockrates and better exploits instruction-level parallelism [YXMZ12].
The cores of a GPU are organized in a hierarchy [1]. Usually, a GPU has a number of *streaming multiprocessors* (SMs) and each SM contains multiple *streaming processors* (SPs). The SP groups threads into a *warp* which usually consists of 32 threads and is executed in single-instruction multiple-data (SIMD) manner. Multiple warps form a thread block and communicate data over the *shared memory*. Each SM can hold multiple thread blocks. The described hierarchy is depicted in figure 1.
In a typical GPGPU environment, the actual computation is called *kernel* and is

[1] In this seminar paper, I use the CUDA terminology. The OpenCL is quite similar, a table can be found in [Hin10].

executed by the CPU (host) and is performed by the GPU (called *device*). Since the device is dependent from the host, it cannot perform computations autonomously. GPU and CPU have both usually caches for each core as well as a larger cache whose data shared between the cores. However, the cache size of a CPU is much larger.
In state-of-the-art consumer devices, CPU and GPU are often integrated onto the same die. And even GPUs of mobile devices have the ability of performing general purpose computations. On such chips, there often is a third cache or a global memory which CPU and GPU are sharing. This eliminates costly CPU-GPU data transfers. By exploiting GPUs computational power, new use cases become possible which where before inapplicable on portable devices due to power and performance constraints. Another challenge of such fused architectures is load balancing of tasks which are more suited for CPUs and those more suited for GPUs and to use the available resources collaboratively.
The disadvantage of GPUs is their rudimentary support for multitasking, which is an essential property of modern CPUs. Concurrent execution of multiple tasks on a single GPU often results in a performance penalty. However, multitasking and load-balancing is necessary to achieve a certain quality of service and to fully exploit the available power, especially for irregular work loads. This limitation becomes more apparent as more and more programs use GPU functionality. Another issue is the high memory latency of GPUs. This seminar papers describe three approaches to improve performance of GPGPU:

- Simulate and test new multitasking approaches to optimize GPGPU workloads in paper *"The case for GPGPU spatial multitasking"* [ACKS12].
- Developing a new programming model to minimize communication overhead between CPU and GPU in paper *"A Study of Persistent Threads Style GPU Programming for GPGPU Workloads"* [GSO12].
- CPU assisted data prefetching for the GPU in paper *"CPU-assisted GPGPU on fused CPU-GPU architectures"* [YXMZ12].

The approaches and results of each paper will be described and reviewed. Afterwards, the papers will be compared and it will be examined if it is possible to combine the shown techniques to further increase GPU performance.

2 The case for GPGPU spatial multitasking

This paper describes approaches how to exploit the full GPU power when multiple tasks are need to be performed, especially for irregular workloads. The authors observed that many GPGPU applications are optimized for a certain GPU generation and show unbalanced resource management if they run on newer hardware. Three types of multitasking have been described:

- **Cooperative multitasking**. A single GPU application uses the device entirely. If a kernel function has finished, other applications can use the kernel. However, it is possible that malicious or malfunctioning applications may never finish. For this reason, some operation system have time limits on GPU computations.
- **Preemptive multitasking**. Each application has a time slot in which it can perform its calculations. Preemptive multitasking was not tested in this paper

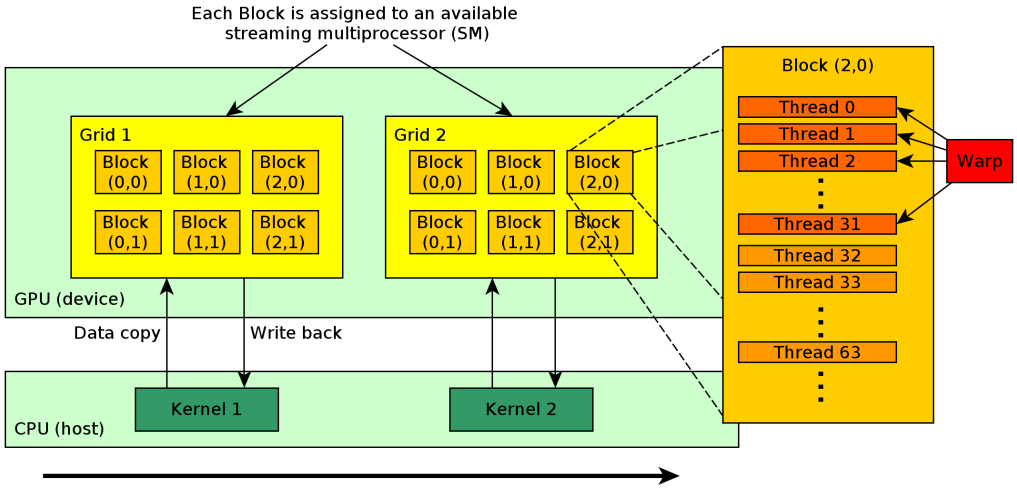

Figure 1: The CUDA programming model.

because the authors claim that it will always perform worse than cooperative multitasking due to context switch overhead.

- **Spatial multitasking**. This is the proposed approach of the paper as the GPU divides its resources rather than its computation time. It allows multiple GPGPU kernels to be executed simultaneously.

All current GPUs support cooperative multitasking and only some support preemptive multitasking [IOOH10]. Only the *CELL* processor and the Intel Larrabee (also known as *Knightsferry*) will support spatial multitasking. However, the latter one is still a prototype [OD12].
To evaluate these different types of multitasking, ten different example applications and benchmarks based on existing CUDA source codes were used, i.e. ray tracing, RSA and AES encryption, JPEG encoding and decoding, image denoising and others. These applications were tested with the GPGPU-Sim [ABF09] which is a cycle-accurate simulator. The default settings for simulating an NVIDIA Quadro FX 5800 GPU were used. Kernel initialization and clean-up was not simulated. Since the GPGPU-Sim has no built-in support for spatial and cooperative multitasking, the author wrote an extension for those use cases.
Because GPU simulation is more complex than CPU simulation, the test applications did not run entirely. Instead, the number of GPU cycles was fixed to 5 million cycles for each application. During a test run, the number of instructions, memory consumption as well as interconnect contention were measured.
During simulation of spatial multitasking, SMs are assigned dynamically to different applications. Many applications do not scale linearly if the number of SMs is increased. This behaviour was expected by the authors as applications are often not fully parallelizable.
Up to four applications ran simultaneously. Spatial multitasking always had better results than cooperative multitasking, with average speedups of 1.14, 1.22 and 1.30 for two, three and four simultaneously running applications, respectively. Memory utilization as well as interconnect latency were examined for spatial multitasking but not for cooperative multitasking.
Furthermore, different strategies on partitioning of SMs were tested. *Oracle best*

and *Oracle worst* form the theoretical maximum and minimum performance, respectively. As they are only theoretical maximum, these strategies cannot be applied to real world applications. Oracle best considers all possible partitions of SMs and select the one with the greatest speedup over cooperative multitasking. Three strategies turned out to be suitable for real world applications:

- **Smart even:** Assigns each application a certain portion of SMs, based on their number thread blocks.
- **Round:** This heuristic minimizes the number of rounds of execution among all applications and no application can take more SMs than it would have with the Smart even approach.
- **Profile:** Each application is profiled in an additional compiling step. With the profiling data, the optimal number of SMs for an application can be determined.

The Smart even heuristic is easy to implement, but can lead to inferior performance compared to the other two approaches. It had average speedups of 1.16, 1.24, and 1.32 over cooperative multitasking for two, three, and four applications, respectively. However, Rounds is difficult to implement but does not need an extra profiling step at compile or install time.
The authors propose that for spatial multitasking, the OS should be responsible for scheduling SMs workloads over the applications whereas the GPU should be responsible scheduling individual threads to each SM.

3 A Study of Persistent Threads Style GPU Programming for GPGPU Workloads

In the previous paper, it was shown that most of the tested applications do not fully exploit the available GPU power. However, it is still possible to increase performance for single or multiple tasks by adopting a new programming style called *Persistent Threads*, which is presented in paper [GSO12]. According to the authors, the PT-style is superior as it overcomes certain inefficiencies in current hardware and programming systems.
The traditional GPU programming style (further refered to as *non-PT*), the programmer has to abstract the program flow into virtual threads which are managed entirely by a hardware scheduler and cannot be influenced by the programmer. The nonPT-style is rather simple and can handle a broad range of applications. However, performance penalties can occur if there are irregular workloads.
In contrast, the proposed PT-style is based on software scheduling and the programmer has full control over the scheduler. From a programmer's view, threads are active for the entire duration of a kernel. The software scheduler is realized through work queues which can either be static (known at compile time) or dynamic (generated at runtime). This queue controls the order, location and timing of the execution of each block. Unfortunately, this programming style does not have any native hardware nor programming API support.
To evaluate the performance of PT and nonPT, several tests in OpenCL as well as in CUDA were written by the authors. The tests consisted of a varying number of fused multiply-add (FMA) operations on single-precision floating-point values.
Four different aspects of PT and nonPT have been further investigated.

- **CPU-GPU synchronization.** Kernel A produces a certain amount that is consumed by Kernel B. nonPT has a roundtrip communication overhead and the number of blocks of A and B needs to be equal whereas in PT, such issues do not occur. The authors conclude that using PT to minimize synchronization overhead works best with kernels of low arithmetic intensity with small workloads and few memory accesses. Moreover, the software scheduling approach is better for kernels with fewer number of thread groups and where the synchronization overhead represents a significant portion of the overall computation time.
  It was observed that the speed decreases with the number of scheduled blocks because the synchronization and scheduling overhead increases. However, with 16 blocks, the PT is approach is twice as fast but for a larger number of blocks, the costs convergence to the level of nonPT programming style. Generally, PT outperforms the nonPT tests with a small number of initial inputs but for larger workloads PT and nonPT show an equal performance. Moreover, CPU-GPU synchronization speed up behaves differently on different hardware platforms. For chips where GPU and CPU share the same die, PT has lesser benefit than for discrete systems where CPU and GPU are connected via PCI-Express.

- **Load balancing/irregular parallelism:** Load balancing issues and irregular parallelism most occur when traversing hierarchical or irregular data structures such as binary trees and hash tables. In such use cases, the work queue of a PT implementation allows to load balance the output of a kernel onto threads for further processing.
  The PT and nonPT load balancing behaviour was tested with an approach called *forest expansion*. Initially, this algorithm starts with a number of initial nodes (roots of a tree) and each thread performs a pre-defined amount of instructions and generates zero, one or two new nodes which subsequently added to the existing tree. In the PT implementation, new nodes to be processed are stored in a shared, global work queue, whereas in the nonPT approach, new nodes computed on the fly after a certain amount of nodes has been generated. This leads to a higher synchronization overhead but the more nodes per level the poorer is the performance gain for PT. For deep-recursive workloads and for a low number of initial nodes, PT clearly outperforms nonPT. As a solution, the authors propose that load balancing should be implemented in hardware and not in software.

- **Producer-consumer locality:** Producer-consumer locality minimizes the cost for data exchange of two kernels. The PT implementation outperforms an optimized nonPT for nearly every test, as depicted in figure 2. If the amount of work increases, the speedup of PT convergences to a factor of 1.1. With lower costs for kernel synchronization, the execution time could be reduced by a factor between 1.50 and 1.85. For the authors, the better control over scheduling and the exploitation of data locality is the biggest benefit of using PT style programming.

- **Global synchronization:** Global synchronization comes into play as the number of blocks is increasing and global synchronization becomes larger than kernel launch overhead. For PT implementations, global synchronization is much easier as it ensures that all blocks are resident. The behaviour of global synchronization is shown in figure 3. The GPU does not have built-in hardware support for synchronizing between active blocks on the same SM nor the entire

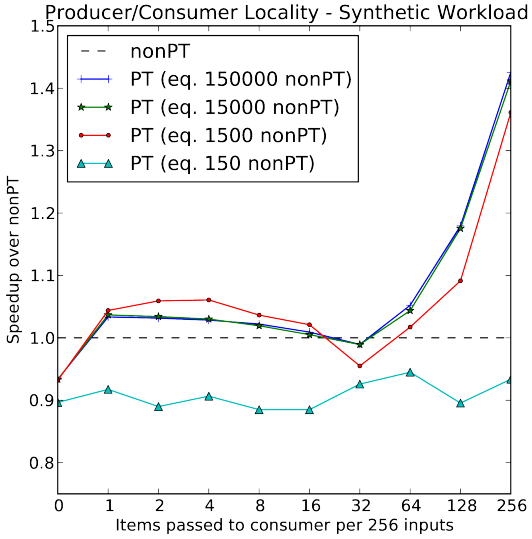

Figure 2: Results of producer locality which represents the data exchange between different kernels.

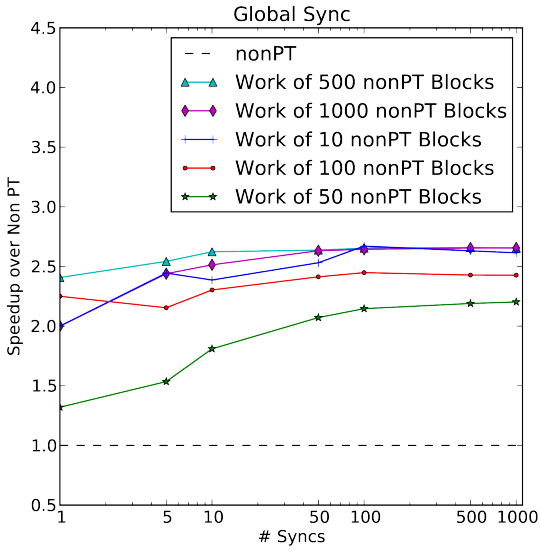

Figure 3: Global synchronization comes into play as the number of blocks is increasing and global synchronization becomes larger than kernel launch overhead.

GPU, but for synchronizing threads in single a block. The authors propose better research in language and hardware features to provide finer, faster and higher-level synchronization techniques to address those issues.

All in all, the authors conclude that PT is a promising programming technique, especially with better hardware scheduling. Issues concerning irregular workloads need to be further investigated.

4 CPU-assisted GPGPU on CPU-GPU architectures

The paper [YXMZ12] focuses on fused CPU-GPU architectures and tries to exploit additional computation power by generating special code for preprocessing on the CPU. On such fused CPU-GPU architectures, CPU and GPU share a common cache, the *L3 cache*. The authors developed a novel compiler algorithm to generate a CPU pre-execution program which prefetches the off-chip memory into the L3 cache for GPU threads.
This pre-execution program is started when the CPU launches a GPU kernel and only contains memory operations and address calculations but no floating point computations. This pre-execution is rather small, with an instruction overhead between 0.69% and 0.74%. The whole approach was not implemented and tested on real GPU hardware but simulated with marssx86 and GPGPU-Sim simulators which modelled an AMD-APU E2-3200. Benchmarks from the CUBLAS library [nVi12] where taken to evaluate performance in *instructions per cycle*.
The authors distinguish between two types of threads in GPU kernels for which the compiler algorithm behaves differently:

- **Lightweight workload**: In step 1, the compiler algorithm extracts the memory operations and the associated address computations and converts them into CPU functions. In step 2, a nested loop structure for prefetching data is generated. An outer loop traverses through all thread-blocks and an inner loop traverses through all concurrent threads of a thread-blocks.
- **Heavyweight workload**: In contrast to lightweight workload kernels, heavyweight kernels contain one or more loops in which global memory is accessed. Step 1 of the compiler algorithm is similar to lightweight workload threads. In step 2, there are three loops instead of two: An outer loop for the thread blocks, an additional inner loop which traverses each kernel loop and the most inner loop for traversing each concurrent thread.

Because a CPU has a higher clock rate than a GPU, an important issue is the synchronization of CPU and GPU which causes CPU code to run ahead of GPU code. As a result, the prefetching process needs to be done at the proper point in time. If the prefetching is done too early, data might be replaced before utilized whereas if it is done too late, memory latencies might occur.
To handle this issue, certain thread ids must be skipped in the inner loop of the pre-execution. The number of thread ids to be skipped is determined by a parameter called `skip_factor`. The parameter is chosen in a fixed (based on profiling) and an adaptive (determined during run-time) way. In the adaptive approach, the pre-execution program measures the number of cache hits and misses. The skip factor increases if the CPU program runs to far ahead and there were too many cache hits. By using these proposed techniques, for adaptive iterator update method, performance improvements up to 113% and 21.4% on average and for fixed iterator update

method, 126% at most and 23.1% on average could be measured. The more memory intensive the tasks the more they benefit from the pre-execution code. Furthermore, the pre-execution algorithm shows a high accuracy: 98.6% of the data blocks loaded from the CPU are actually accessed by the GPU.
Another test was to let the CPU perform certain threads actually intended to be performed by the GPU. In the simulators, the authors implemented a special CPU instruction which tells the CPU which GPU threads are occupied by the GPU so there is no workload overlap. Although this approach sounds quite promising, however, the overall performance gain was with an average of 5% is rather disappointing.

5 Comparison

The three presented papers consider different aspects of improving performance of GPUs.

The paper [ACKS12] focuses on how to run several different tasks in parallel efficient as possible. These used benchmarks comply very well to real world applications, however they were only performed in a simulator and thus are fully synthetic. However, in a simulation, all simulations were started simultaneously, which is not comparable to real world conditions. Moreover, kernel initialization and clean-up were not simulated. The third paper shows that this can have significant performance impact, especially for memory intensive tasks. Since multitasking is not a mandatory feature for portable devices, the use case of this approach rather aims for workstations, cloud computing and high performance computing.

The second paper [GSO12] focuses on how to exploit as much computational resources as possible on mostly heterogeneous tasks by adopting a novel programming style. The authors conclude that implementing native support of the PT programming would not be an "expensive exercise".
As in PT style, the programmer has to define scheduling behaviour on his own. More freedom can often lead to better performance exploitation. However, I expect that this programming style might be awkward, especially for unexperienced programmers. Due to manual handling of the scheduler, I assume that PT will not be more convenient for the programmer and development of GPGPU applications could take longer and be more expensive.
Furthermore, significant performance improvements could only be achieved in some special cases. It is an open question if PT is really a good win for a broad range of applications if one considers the rather moderate speedups. This approach aims for performing more irregular, heterogenous, former CPU-centric tasks on graphic cards. This affects the whole range of applications of GPUs, from portable devices to supercomputing.

The third paper's [YXMZ12] approach tries to achieve synergetic effects by optimizing data flow between CPU and GPU. Interestingly, the performance gain of 5% when distributing the workload on both CPU and GPU is rather small. This is probably caused by different data flow and clock frequency.
Although all results were only obtained by a simulator, the overall performance gain sound already promising. However, only the benchmarks from CUBLAS benchmark suite have been used. Thus, it would be nice to know how this approach works on other benchmarks such as those utilized in paper [ACKS12].
Furthermore, it remains an open question how the approach behaves on real hardware and whether it will be integrated into existing compilers. In this paper, only the behaviour of fused architectures was considered which are rather common in

portable devices such as notebooks, smartphones and tablets. It is up to further research how it behaves on distinct CPU-GPU environment where both device do not share a common cache. This would extend the field application towards high performance computing and cloud computing.
By comparing these papers, three general trends can be spotted:

- **GPUs became an established counterpart for CPUs.** Nearly every computing consumer device has a built in GPU whose available computational need to be used. In contrast to CPUs, GPUs consume generally less watts per FLOP which extends battery life and is environmentally friendly. Especially in the emerging market of portable devices, this can lead to better user experiences. Unfortunately, in none of the papers, the power consumption was measured.
- **GPUs adopt CPU's functionality.** Especially the first two papers try to make GPUs work better with multitasking as well as with heterogenous and irregular tasks. Generally, those approaches want to test how far GPUs can be optimized to perform CPU-like tasks. However, CPU-like GPU technologies such as Larrabee (also known as Knightsferry) and CELL were not considered at all. It would be interesting, if it is possible to implement CPU features on GPUs like branch prediction and data prefetching.
- **GPUs require novel programming and compiler techniques.** Although with CUDA and OpenCL there are de-facto standards in parallel computing, there are still some language features to be implemented. Desirable language features which are state-of-the-art in CPUs but not in GPUs are i.e. dynamic memory allocation, recursion and preemption. Additionally, better compiler and profiler techniques to generate more efficient machine code are eligible. As the second and third paper show, approaches particularly aimed for optimizing global synchronization, CPU-GPU communication and prefetching can lead to performance benefits.

A general point of critique is that in every paper, the presented measurement results were only obtained by GPU simulators. Each paper focused only on two or three particular hardware platforms, even though it was shown in the first paper, that the performance gains of an approach can vary largely on different hardware platforms.
Although the used simulators GPGPU-Sim and marssx86 claim to be cycle accurate, only implementations on real hardware can tell whether the described approaches actually work. As benchmarks and measurements are fully synthetic, all measurements should be considered without warranty.

6 Conclusion

To use the full potential of GPUs, there is still a lot of further research to do, primarily in terms of multiple, heterogeneous tasks as GPGPU applications are likely to exhibit unbalanced resource usage in future GPUs. Furthermore, a good trade-off between convenient programmability and performance needs to be found. This requires better compilers and the further development and utilization of well-defined, standardized programming languages such as OpenCL and CUDA.
Especially in the emerging market of cloud computing there are a lot of open questions, for instance how to load balance multiple GPUs interconnected over a network. Energy efficiency is another issue to addressed. Better tools need to be developed

and deployed to minimize energy consumption and future research papers should consider this problem more often.
As the presented papers show, a broad range of applications can benefit from GPUs processing power. Scientific computing for climate and physics simulations, more possibilities of portable devices and all of this performed with as less watts per FLOP as possible. Therefore, GPGPU is a future-proof research area.

References

[ABF09] Tor M. Aamodt, Ali Bakhoda, and Wilson W.L. Fung. GPGPU-Sim: A Performance Simulator for Massively Multithreaded Processor Research. Technical report, University of British Columbia, 2009.

[ACKS12] Jacob Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The Case for GPGPU Spatial Multitasking. In *HPCA*, pages 79–90, 2012.

[BS10] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. Technical report, Department of Computer Science, University of Virginia, 2010.

[Gil09] Mike Giles. GPUs for Scientific Computing. Technical report, Oxford-Man Institute of Quantitative Finance, 2009.

[GSO12] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Innovative Parallel Computing*, page 14, May 2012.

[Hin10] Vincent Hindriksen. Difference between CUDA and OpenCL 2010. 2010. `http://www.streamcomputing.eu/blog/2010-04-22/difference-between-cuda-and-opencl/`.

[IOOH10] Fumihiko Ino, Akihiro Ogita, Kentaro Oita, and Kenichi Hagihara. Cooperative Multitasking for GPU-Accelerated Grid Systems. Technical report, Osaka University, 2010.

[MDZD09] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical Power Consumption Analysis and Modeling for GPU-based Computing. Technical report, University of Houston, 2009.

[nVi12] nVidia. *CUDA Toolkit 4.1 CUBLAS Library*, January 2012.

[OD12] Hamid Oloso and Kaushik Datta. Experience with a Prototype Intel MIC System. Technical report, NASA, 2012.

[SG08] Robert Szerwinski and Tim Guneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. Technical report, Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany, 2008.

[YXMZ12] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. CPU-assisted GPGPU on fused CPU-GPU architectures. In *HPCA*, pages 103–114, 2012.