GPGPU Memory Characterization: A Cross-Platform Quantitative Study

Adi Fuchs Noam Shalev Avi Mendelson

Technion Computer Engineering Center, Technion IIT, Haifa, Israel
{adi,noams,avi.mendelson}@technion.ac.il

Abstract

General-purpose GPUs (GPGPUs) hold the potential of a high computational throughput – supporting the execution of many concurrent tasks. The amount of computational intensity is enabled by the abundance of simple, low power execution units in a typical GPU microarchitecture. These systems trade performance features that consume much power, such as out-of-order execution, with new SW/HW interfaces. Thus, GPGPUs highly depend on efficient utilization of their microarchitecture that is enabled by software optimizations that should be carried out carefully. This is not a trivial task, since the optimizations that are done for one GPGPU architecture may not fit other systems. Unfortunately, many of the dominating characteristics of GPGPU microarchitectures are not publically available, as manufacturers tend to keep them under strict confidentiality. The purpose of this work is to suggest a set of micro-benchmarks that aims to reveal the characteristics of different graphics cards, in respect to features that may impact the optimization of GPGPU applications. As a first step in this direction, this paper focuses on memory architecture of different NVIDIA cards. The tools and results presented in this work can be used either as a baseline for comparison between different generations of GPU cards and/or as a guideline for GPU programmers optimize their future applications. In order to explore the new proposed tools on different platforms, the kernels were executed under four different GPU systems using the CUDA programming environment. This work highlights several insights that are often oblivious to programmers and can significantly affect GPU performance: in all systems tested, the overhead of fine grained synchronization for memory bound workloads resulted in a slowdown of over 260%; the inability to coalesce concurrent memory reads in massively parallel workloads caused a slowdown of over 264% and massive register spilling resulted a slowdown of more than 1140%. Another interesting insight which was noticed is that for some of the tests performed, newer GPU systems did not necessarily perform better than their predecessors.

1. Introduction

During the past decades computing industry has dealt with many challenges posed by the ever growing demand for computation. Power constraints have hindered the progress in single task performance [1] forcing a shift towards parallel hardware architectures, such as multi-core CPU and GPU architectures, that present a high computation potential without an increase in processor frequency rates. In order to exploit the benefits of these architectures, a shift in the commonly used software models was needed as well, towards parallel programming paradigms - as they enable the programmers to express computationally independent segments within a program as parallel tasks, which can be concurrently executed on different computation machines. The parallel computation era induces a growing dependency between hardware and software, as programmers and programming library developers should be more aware of the underlying hardware behavior, in order to efficiently utilize it. When comparing multicore CPU architectures to GPU architectures, the latter consist of a larger number of simpler processing elements, highly parallel memory architecture and mainly were used for graphic computations. As general purpose tasks become increasingly parallel, some of the recent programming environments, such as CUDA [2] and OpenCL [3] enable the execution of general purpose kernels on GPUs (often referred to as: "GPGPU"). Since GPGPU programs target an efficient utilization of GPU architecture, they usually consist of many concurrently executing threads, which strain the various architectural elements and the memory in specific, making these programs highly sensitive to the underlying micro-architecture and the amount of collaboration between hardware and software. In this work a set of micro-benchmarks was developed to expose some of the various behaviors in the GPGPU memory hierarchy – workloads explore register file spilling, the GPUs sensitivity to synchronization granularity and the effects of spatial memory locality.

The main contributions of this work are:

1. A set of structurally independent kernels which unravel some of the GPU's micro-architecture attributes.
2. A quantitative study of four different commercial NVIDIA GPU platforms, from 2 major development generations - using the developed kernels as baseline for behavioral comparison.

The testing process unveiled several issues, such as caching issues in the new GPU generations for global memory and unexpected overhead caused by in-thread-block synchronization primitives. In some cases, overhead related to enforcing tight synchronization for memory bound workloads is significant and result in a slowdown of over 260%. This raises the question on the use of automated synchronization mechanisms and their costs – such costs impede potential performance of GPU programs, therefore fit a programming model for which synchronization must be done explicitly by the programmer. This reveals some other important aspects of NVIDIA's GPU characterization such as how efficient is the implementation of register spilling; this paper will report that under some conditions, it may cause a performance degradation of up to an order of magnitude.

2. Related Work

Several studies have dealt with GPU micro-architecture behavior; however, this work is the first to combine a quantitative study of four commercial GPU systems with benchmarks that are structurally independent, making them micro-architecturally neutral, this approach allows the benchmarks created to run on other GPU systems and in the future to be easily implemented on other GPGPU programming languages (e.g OpenCL). The work of Goswami et al. [4] has examined the behavior of a GPU environment for complex workloads (K-means, PCA etc.) to test different aspects, for that they have used a simulation environment (GPGPU-sim) and measured the runtimes under different configurations. Lashgar and Baniasadi [5] tested the implications of various control flow mechanisms on GPU memory behavior under GPGPU-sim as well, they have used CUDA to run a set of known benchmarks (NN, Matrix Multiplication etc.). Unlike the above mentioned papers, this work targets the pinpointing of specific behavioral patterns based on the results of synthetic benchmarks that were created and executed on real GPU systems. Wong et al [6] created a set of micro-benchmarks targeting various aspects of the NVIDIA GT200 GPU microarchitecture e.g. cache structure, branch divergence, clocking domains etc. The kernels presented in their work targeted specific structures in the micro-architecture (for example cache set structure), while this work contains generic kernels that do not have any structural assumptions (e.g. cache mapping, TPC/SM structure) on the tested micro-architectures, the affecting parameters were part of the programming model (e.g. number of threads, number of synchronization instructions) thus enabling a more extensive study: the kernels' code was compiled and executed in the exact same manner on the 4 systems tested without any adaptations, enabling the most reliable methodology for comparing various GPU systems.

3. Evaluation

3.1 Platforms

All systems run Ubuntu 12.04 on x86_64 architecture, the following tables contain the hardware configurations extracted using the **cudaGetDeviceProperties()** runtime function.

Table 1. CUDA Capability 2.x machines

	C2070	**Quadro 2000**
Device Name	Tesla C2070	Quadro 2000
GPU Architecture	Tesla	Fermi
CUDA Driver / Runtime Version	5.0 /5.0	5.0 /5.0
CUDA Capability	2.0	2.1
Global memory size	6144 MBytes	1024 MBytes
Multiprocessors	14	4
CUDA Cores/MP	32	48
Total number of cores	448	192
GPU Clock rate	1.15 GHz	1.25 GHz
Memory Clock rate	1.5 GHz	1.3 GHz
Memory Bus Width	384-bit	128-bit
L2 Cache Size	786432 bytes	262144 bytes
Constant memory size	65536 bytes	65536 bytes
Shared memory per block	49152 bytes	49152 bytes
Max registers per block	32768	32768
Warp size	32	32
Max threads / MP	1536	1536
Threads per block	1024	1024
Linux kernel version	3.2.0-32 -generic	3.2.0-32 -generic

Table 2. CUDA Capability 3.x machines

	GTX680	**K20**
Device Name	GeForce GTX 680	Tesla K20m
GPU Architecture	Kepler	Tesla
CUDA Driver / Runtime Version	5.0 /5.0	5.0 /5.0
CUDA Capability	3.0	3.5
Global memory size	4096 MBytes	4800 MBytes
Multiprocessors	8	13
CUDA Cores/MP	192	192
Total number of cores	1536	2496
GPU Clock rate	1.06 GHz	0.71GHz
Memory Clock rate	3 GHz	2.6 GHz
Memory Bus Width	256-bit	320-bit
L2 Cache Size	524288 bytes	1310720 bytes
Constant memory size	65536 bytes	65536 bytes
Shared memory per block	49152 bytes	49152 bytes
Max registers per block	65536	65536
Warp size	32	32
Max threads / MP	2048	2048
Threads per block	1024	1024
Linux kernel version	3.2.0-38-generic	3.2.0-38-generic

3.2 Benchmarks

In order to get the best performance out of NVIDIA's cards, this work presents a new Micro-benchmark suite, using the CUDA programming environment – the target of these benchmarks was to measure several aspects in cross-platform micro-architecture: structural characteristics such as cache line or pre-fetch sizes for the various memory types, and behavioral characteristics such as the effects of memory coalescing, cache misses, registers file spilling (a scenario in which a kernel's variables cannot fit in the register file and are stored in the memory) and by that provide a rough estimation to the performance variance between a highly tuned GPU kernel and an highly unbalanced kernel. For example, many adjacent memory reads from different threads can be grouped by the GPU memory scheduler to the same transaction by memory coalescing, while concurrent reads to distinct memory areas cannot be grouped and are performed in a sequential manner – resulting in severe performance degradation of almost an order of magnitude merely due to bad spatial locality, which is sometimes oblivious to the GPU programmer.

3.3 Methodology

The notation for benchmark performance in this work is derived from the latency perceived by the threads running the kernels, using CUDA's clock() function. Meaning, all kernels are structured in the following manner:

```
<kernel definitions>
start=clock();
<kernel execution code>
end=clock();
return (start-end);
```

The results extracted from the kernels were scaled from number of clocks to the actual latency according to the GPU clock rates given in Table 1 and in Table 2. The performance notation in this work was derived from perceived latencies; if needed, the throughput can be derived as well by combining the perceived latencies with the number of running threads.
All kernels were executed under CUDA runtime version 5.0, compiled using "-O3" optimization flag ('-Olimit=118245' was sufficient for the kernels containing larger procedures).

3.4 Main Results

The tests conducted aim to both unveil the implications of programming patterns on the performance of the GPU memory architecture and compare memory architecture related features on different NVIDIA CUDA generations. In specific, these tests highlight 4 main aspects in the memory hierarchy:
1) The prefetch mechanism of the GPU, which translates into the ability of the GPU to exploit spatial memory locality, using its cache mechanisms.
2) The overhead of global memory synchronization granularity in memory intense workloads.
3) The contribution of memory coalescing to performance and the implications on performance for cases in which coalescing cannot be performed.
4) The implications of the register spilling phenomenon that occurs in cases for which local variables cannot fit in the register file.

3.4.1 Exploring locality different types of memory

The purpose of this kernel is to discover the sizes and latency implications of caching mechanisms, if present. This is done by examining the effect of 'cold-start' misses – meaning, cache misses resulted from accesses to memory regions never before read. For that, kernels allocate a large array and perform 1024 sequential and dependent reads, divided to 512 couples. Each couple consists of "small jump, large jump" memory accesses: "large jump" is a fixed large distance for which no reasonable caching mechanism is designed to perform a pre-fetch (we have used a fixed size of 4KB) the size of "small jumps" varies between 1 and 512 bytes at distinct kernel executions. Note that this access pattern also prevents from stride detection mechanisms, if exist, to perform a pre-fetch. Benchmarks execute a single kernel at a time since the purpose of this kernel is to discover variance in latency resulted from a change in locality.

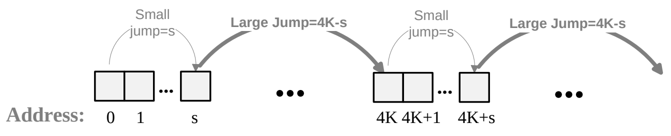

Figure 1. The memory access pattern for small jump size = S.

To more accurately formulate the expected kernel results, the following functions were defined:

$$L(s,n)=\begin{cases}1 & level\ 'n'\ cache\ line\ size\ is\ larger\ than\ s\\ 0 & otherwise\end{cases} \tag{1}$$

$$H(s,n)=L(s,n)\cdot\prod_{j=1}^{n-1}\left(1-L(s,j)\right)$$

$L(s,n)$ returns 1 if the line size of level n of the cache is larger than s. Therefore, $H(s,n)$ shall result in 1 if the line size of level n of the cache is larger than s, but all of the cache levels which are closer to the processor, down to L1, have a smaller cache line size than s. Given N cache levels, and the access time for each cache level k, $T(level\ k)$, the expected kernel latency for "small jump" stride of size s should be the following:

$$T(s)=512\cdot\left[\underbrace{T(MemoryAccess)}_{long\ jumps}+\underbrace{\sum_{k=1}^{N}H(s,k)\cdot T(level\ k)}_{short\ jumps}\right] \tag{2}$$

Note that $H(s,k)$ returns non-zero result only for one value of k. Also mark that after each long jump, a memory access will be needed. Thus, we get that the expected time for the kernel to execute consists of 512 memory accesses (for the 512 long jumps) and 512 accesses to the first level of the cache (or memory) which contains the address of our last access plus our stride s. In order to refrain from further complicating formula (2) the assumption for the latency in level k, $T(level\ k)$, is that it includes the latencies of the seek in lower levels.

The kernel was executed for 4 different memory types supported by the CUDA runtime environment: the global memory, the constant memory, the shared memory and the texture memory.

Shared memory:
In current GPU systems the shared memory is an on-chip memory, making it potentially faster than other memory types. It is allocated per thread block, so all threads in the block have access to the same shared memory. Since the shared memory is relatively small this specific kernel contained a relatively small array of ~48KB.

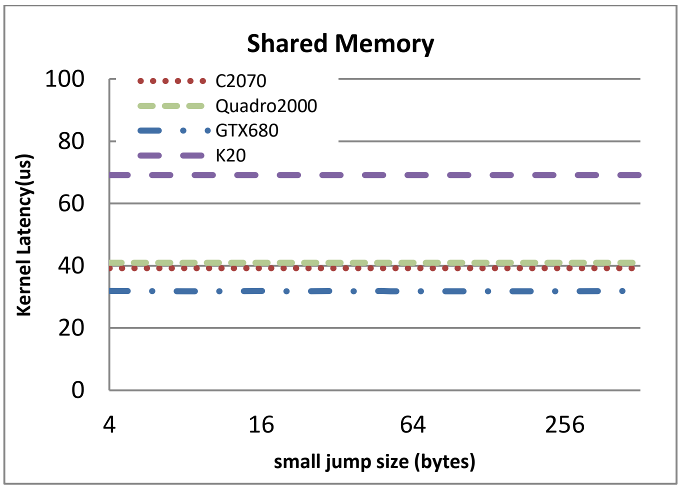

Figure 2. Execution results for shared memory with varied stride sizes

As one can infer from figure 2 – for all systems tested, the shared memory has a fixed latency – this implies that shared memory is <u>not cached</u> (though it is often used as a user managed cache on software level). Though original kernels' results were in number of clocks - when combining the perceived latency with the clock rates in Table 1 and Table 2, it appears that shared memory is around 70% <u>slower</u> for the new TeslaK20 CUDA system (generation 3.5).

Texture memory:
Unlike other memory types, Texture memory space is an abstraction provided by the GPGPU programming environment, rather than a actual memory mapped to a physical device, as described in the CUDA programming guide [8] it is optimized for multidimensional accesses and can contain up to 4 coordinates.

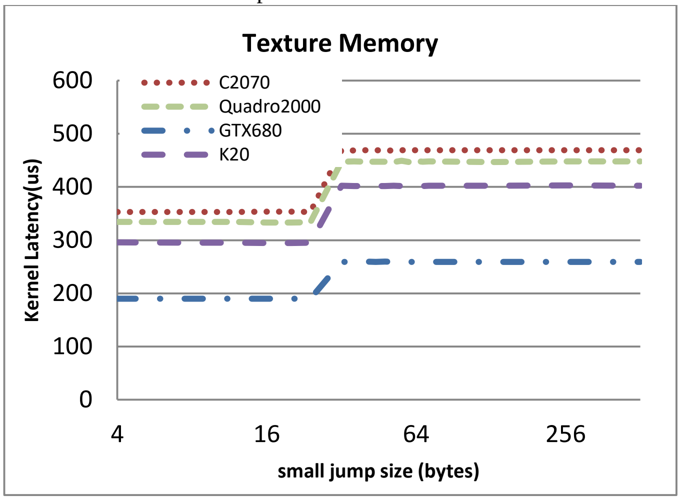

Figure 3. Execution results for texture memory with varied stride sizes

As seen in figure 3 - for all systems tested, texture memory latency increased for a step size of 32 bytes, indicating that texture memory caches a line size of 32 bytes in the first level; this is likely since textures in CUDA consist of 4 dimensional coordinates which can be either long or double precisions (8 bytes each).

Constant memory:
The constant memory is an on-card memory, containing read-only data (i.e. variables and arrays annotated by the reserved 'constant' key word in CUDA) – the constant memory is accessible to all threads and blocks within a grid.

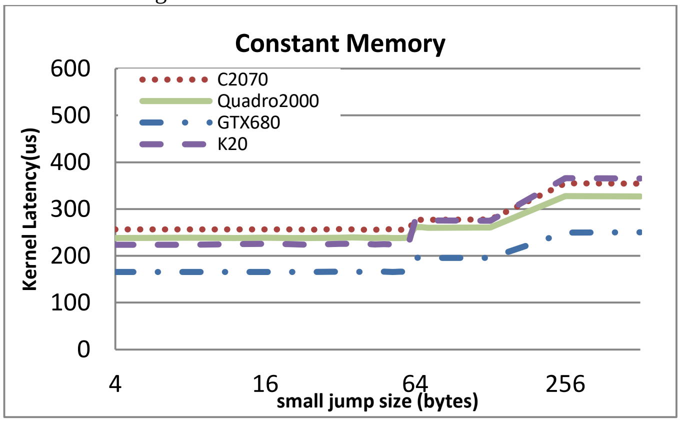

Figure 4. Execution results for constant memory with varied stride sizes

For all systems tested, Latency change for constant memory in 2 distinctive points - in 64 bytes and 256 bytes, implying that constant memory has 2 levels of cache – the first of 64 byte line size, the second level is 256 bytes. Previous studies exploring the GT200 GPU system [7] reveal that GT200 has an L2 cache line of size 256 bytes – if this is the case here, it implies that upon access the GPU systems both loads the corresponding 64 bytes line into the L1 cache and initiate a pre-fetch transaction to request the corresponding 256 bytes line to the L2 cache.

Global memory:

Global memory stores global variables and variables and it can be used both by the GPU and the host (after proper mapping)

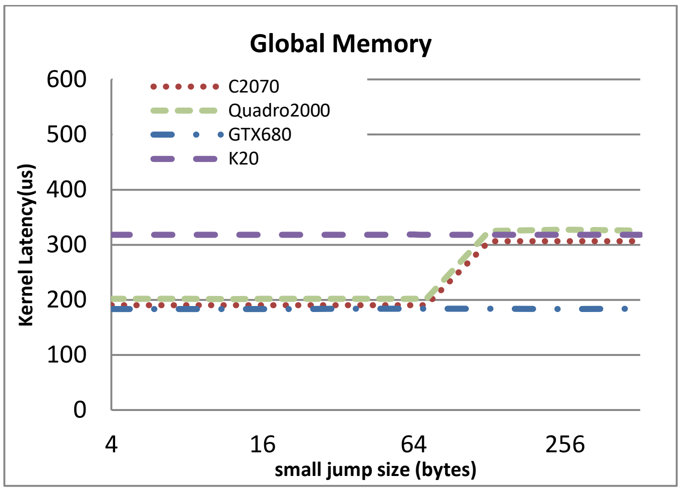

Figure 5. Execution results for global memory with varied stride sizes

As one can infer from figure 5, the global memory behaves differently for the previous generation CUDA systems (2.x). For the previous generation systems, the latency is increased in step sizes of 64 bytes which indicate a first level cache line size of 64 bytes. For newer generation systems no latency increase can be seen in the graphs, indicating the absence of caching mechanism for global memory. A possible reason for the lack of global memory caching in 3.x devices is due to high memory clock rates comparing to their predecessors, as mentioned in tables 1+2 – this suggests that the GPU manufacturers for CUDA devices with generations favored higher frequency memories over caching mechanisms.

3.4.2 The effects of local thread synchronization

This kernel explores the overhead caused by synchronizing threads for a memory bound workload, consisting solely of memory writes. The kernel performs 1024 memory accesses and using CUDA's __syncthreads() to synchronize threads belonging to the same blocks. The parameter being changed here is N = number of __syncthreads() instructions, as N increased – so is the frequency of synchronization instructions, starting from N=1 (__syncthreads() is called only at the end of all 1024 accesses) and reaching N=1024 (__syncthreads() is called after every memory access) - given a general N the kernel execution code is structured in the following manner:

"1024 Memory writes, N syncs" kernel execution code

```
$arr[0] \ = \ p++;$
$arr[1] \ = \ p++;$
...
$arr[(1024/N)-1] \ = \ p++;$
$\_syncthreads();$
$arr[1024/N] \ = \ p++;$
$arr[(1024/N)+1] \ = \ p++;$
...
$arr[(2*1024/N)-1] \ = \ p++;$
$\_syncthreads();$
```

...

As N increases, the overhead of memory synchronizations increase accordingly, in a sense one can perceive N=1024 as fine grained synchronization (synchronization performed after every memory access) performed for benchmarks with high memory traffic. The benchmarks were executed several times - both for varied number of synchronizations and for varied number of threads. All threads execute the abovementioned kernel code on the same array, mapped to the global memory; by enforcing such workload with varied synchronization frequency this benchmark targets a quantification of the overhead caused by global memory synchronization in a memory bound workload

CUDA 2.x systems:

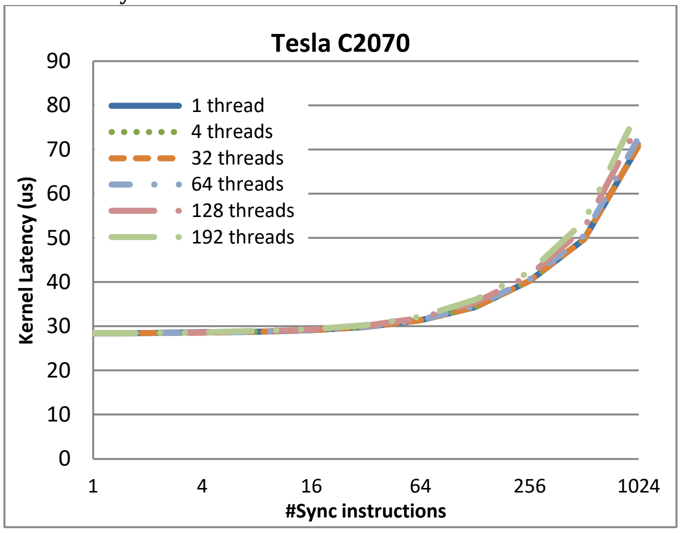

Figure 6. Latency of 1024 global memory writes w.r.t the frequency of synchronization instructions for Tesla C2070.

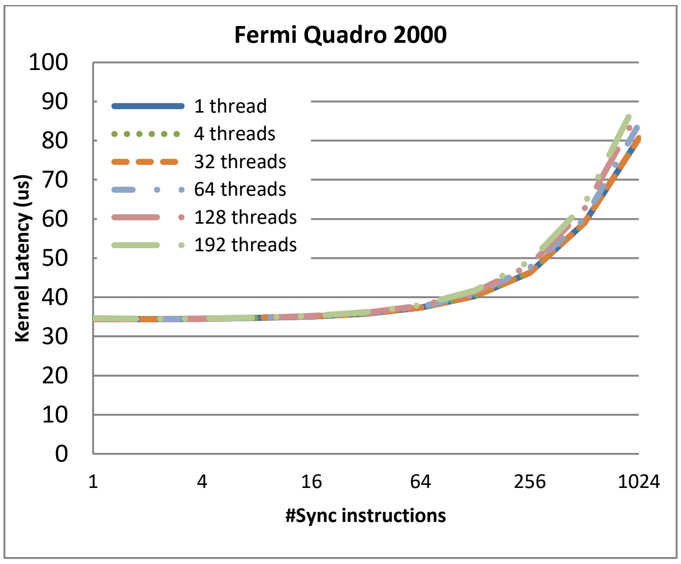

Figure 7. Latency of 1024 global memory writes w.r.t the frequency of synchronization instructions for Fermi Quadro 2000.

CUDA 3.x systems:

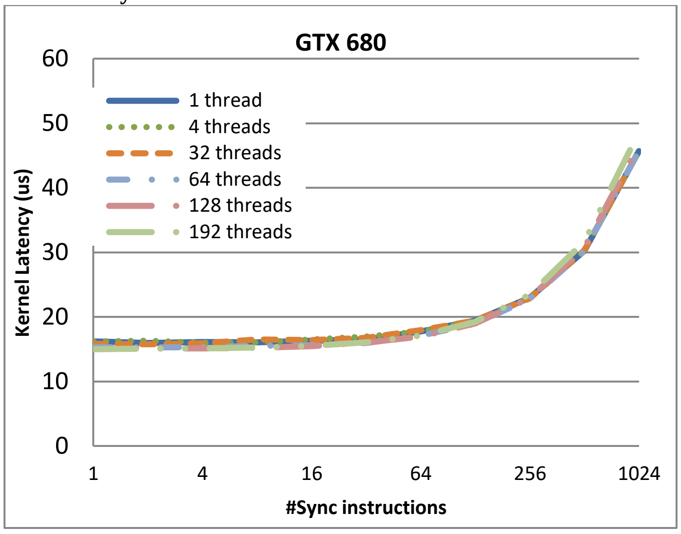

Figure 8. Latency of 1024 global memory writes w.r.t the frequency of synchronization instructions for Kepler GTX680.

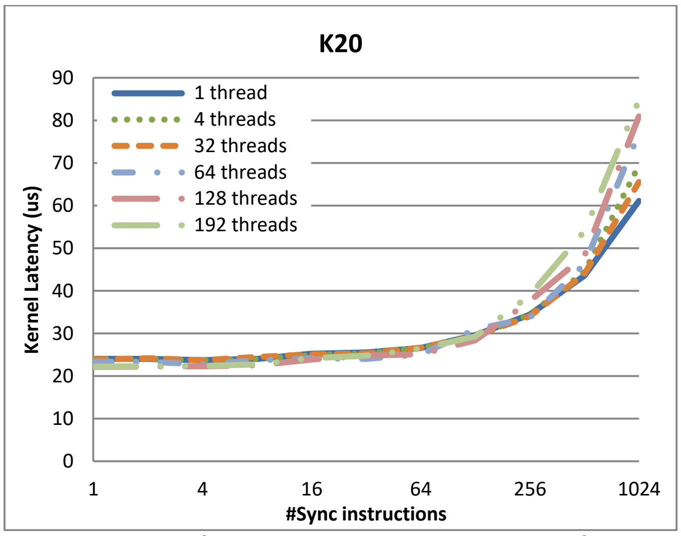

Figure 9. Latency of 1024 global memory writes w.r.t the frequency of synchronization instructions for Tesla K20.

When looking at figures 6-9 one can deduct that a change in the number of concurrent threads increases latency by a maximum of 13% for Fermi Quadro 2000, 11% for Tesla C2060, 38% for Tesla K20 and 6% for Kepler GTX 680, while the increase in the frequency of synchronization instructions (up to one for each memory write) increases latency by 163% for Fermi Quadro 2000, 178% for Tesla C2060, 223% for Kepler GTX 680 and 281% for Tesla K20. This clearly demonstrates the magnitude of overhead resulted from fine grained synchronization in high memory bound benchmarks, harming both latency and throughput thus impeding the gain from a high number of threads. It can also be inferred that Kepler GTX680 demonstrated the best performance, while the Tesla K20 GPU, which is a more advanced device, suffered from some fluctuations and performed worse.

3.4.3 The effects of memory coalescing

In order to examine the behavior under various concurrent global memory access patterns, threads executing this kernel invoke a sequence of 1024 read instructions from adjacent addresses, each thread starts from a different offset – by changing the number of threads and the size of the offset, this micro-benchmarks simulates varying stress on the memory scheduler, as well as examining the size of the coalescing window.

"1024 reads + varied offset" kernel execution code:
*start = thread_id*offset;*
for (i =0; i < 1024; i++) read(array[start+i])

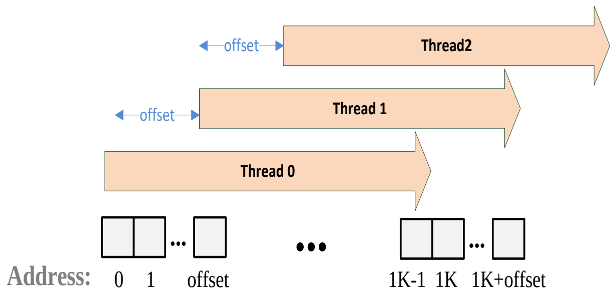

Figure 10. Memory coalescing benchmark flow for 3 threads

The cross-thread offset was changed from 4 bytes (high adjacency between threads, the memory scheduler can coalesce several threads reads) to 1KB (all threads read from distinct memory areas)

Tesla C2070+ Fermi Quadro 2000:

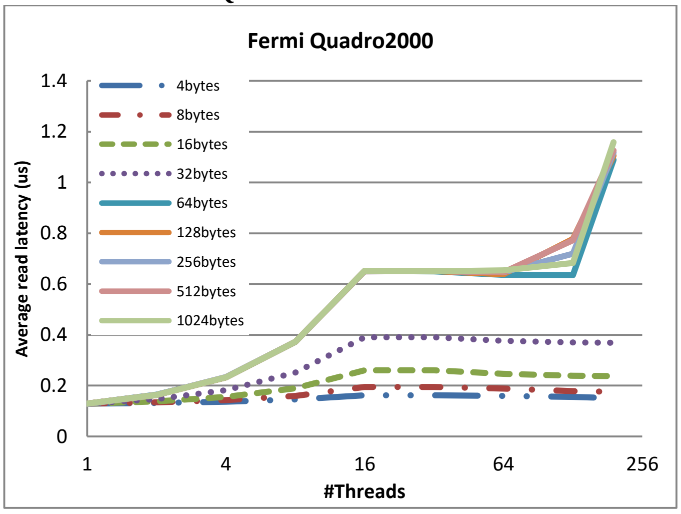

Figure 11. 1024 consecutive memory with varied threads staring points w.r.t number of threads for Fermi Quadro 2000

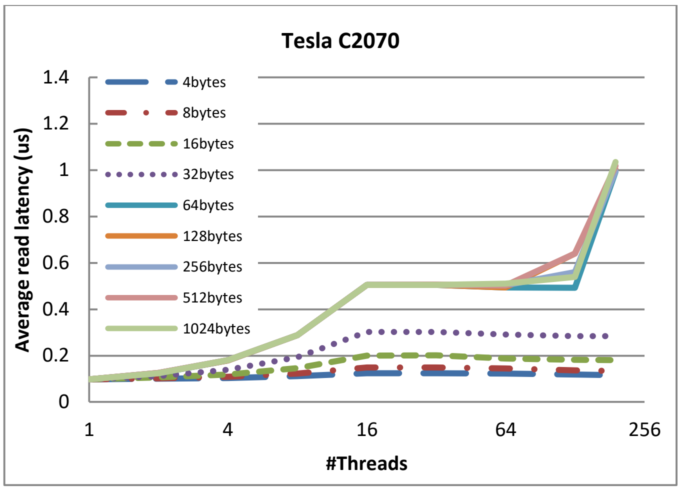

Figure 12. 1024 consecutive memory with varied threads staring points w.r.t number of threads for Tesla C2070

CUDA 3.x systems:

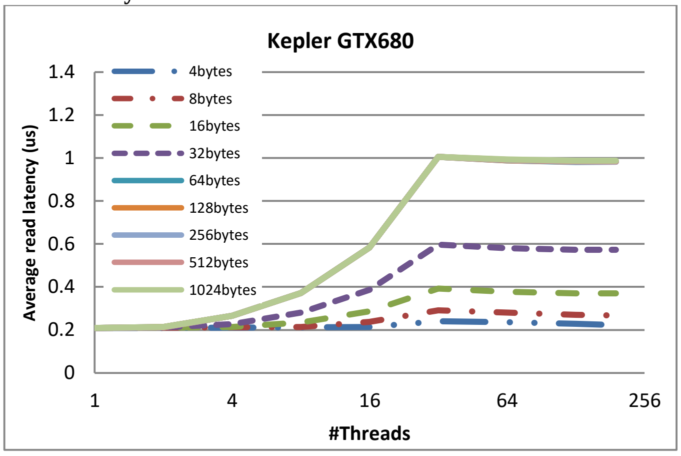

Figure 13. 1024 consecutive memory with varied threads staring points w.r.t number of threads for Kepler GTX 680

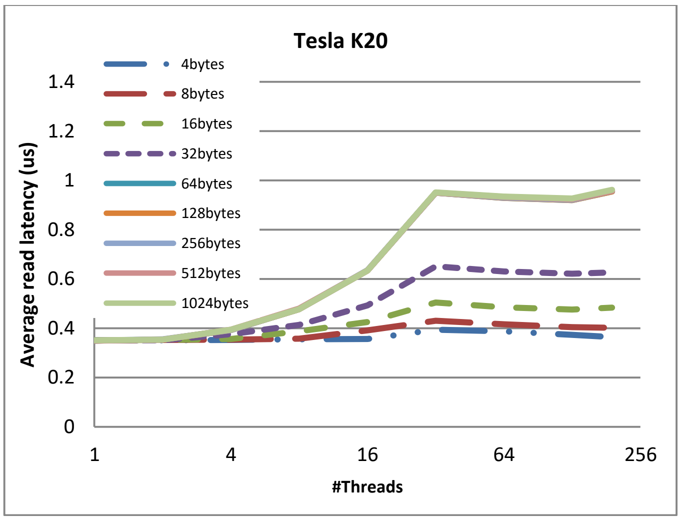

Figure 14. 1024 consecutive memory with varied threads staring points w.r.t number of threads for Tesla K20

The issues that can be inferred from the above figures are as follows:
1. Latency increases as a function of the number of threads – since not all memory transactions that are invoked concurrently can be executed concurrently via coalescing.

2. The latency increases as the offset increase – a larger offset reduces the spatial locality between threads, therefore the ability to coalesce their memory transactions. The overhead of high offset combined with many concurrently executing threads was 898% for the C2070 system, 770% for the Fermi Quadro 2000 system, 443% for the Kepler GTX680 system and 263% for the Tesla K20 system.

3. The increase in latency plateaus for 16 threads in both 2.x systems. The probable reason is caching – since all threads perform serial reads, as number of threads increase so is the number of lines concurrently loaded to the global memory cache (therefore threads can use data pre-fetches earlier by other threads). An additional increase in latency is spotted starting 128 threads for both systems, for offsets larger than 32 bytes. The reason for that is a global cache fill-up, which is caused by the increasing of both the offset size and the number of threads, which together enlarge the effective working set, crossing the maximal cache set capacity. The loss of locality causes performance degradation of up to 650% in both systems (comparing to the executions with 4 bytes strides).

4. The increase in latency plateaus for 32 threads in 3.x systems and in addition, overall latency is higher than the latency of 2.x systems – the reason for that probably also lies in the absence of cache mechanism for 3.x systems: without the ability to coalesce reads, and without any cache structure, 3.x systems perform worse than their 2.x counterparts, under memory bound workloads and no ability to coalesce the accesses to the memory.

3.4.4 The effects of register spilling

When a program has more live variables than the machine has registers, some variables are "spilled" from the register file into the memory. The purpose of this benchmark is to quantify the effects of register spilling, by changing the number of long variables defined in the kernel. Since too many variables cannot fit in the register file at a given time, they shall be stored in the memory that will be used as an extension to the register file. In order to prevent the compiler from optimizing the code and group the variables, the variables are loaded and stored in a random order, thus creating a dependency tree which is too complex to be resolved for a large enough number of variables. Below is the pseudo code for the benchmark kernel, for which X represents the number of variables.

Kernel definition code:

```
<define X long variables initialized to random values >
Long v;
```

kernel execution code:

```
$DO$ 4096:
$v \ \ += x_i; \ \ /* \ (x_i \ is \ a \ randomly \ chosen \ variable) \ */$
$x_k += v; \ \ /* \ (x_k \ is \ a \ different \ randomly \ chosen \ variable) \ */$
```

CUDA 2.x systems:

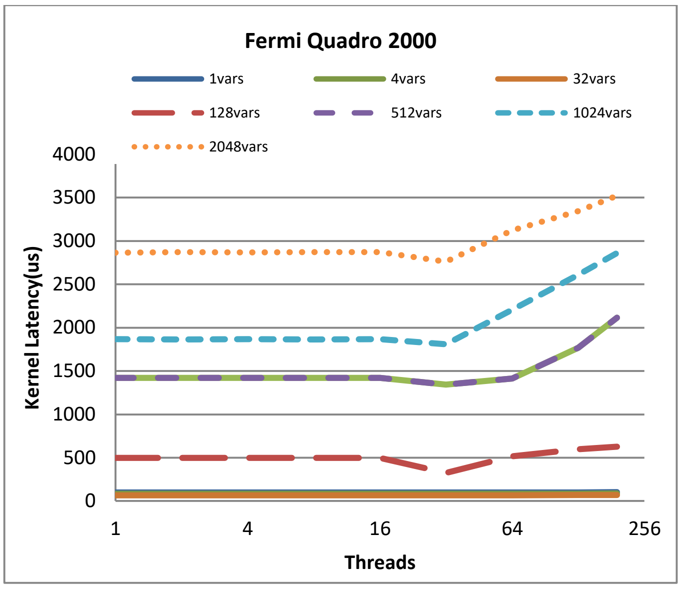

Figure 15. 4096 reads and writes with varied number of long variables w.r.t number of threads for Fermi Quadro 2000

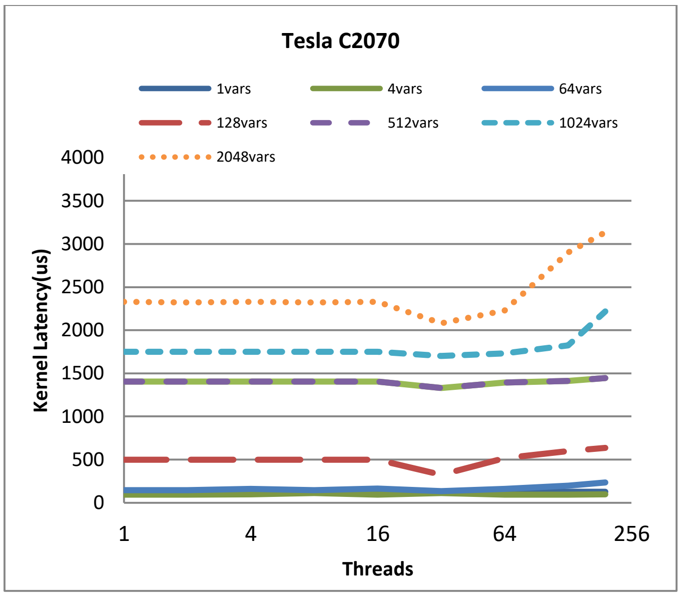

Figure 16. 4096 reads and writes with varied number of long variables w.r.t number of threads for Tesla C2070

CUDA 3.x systems:

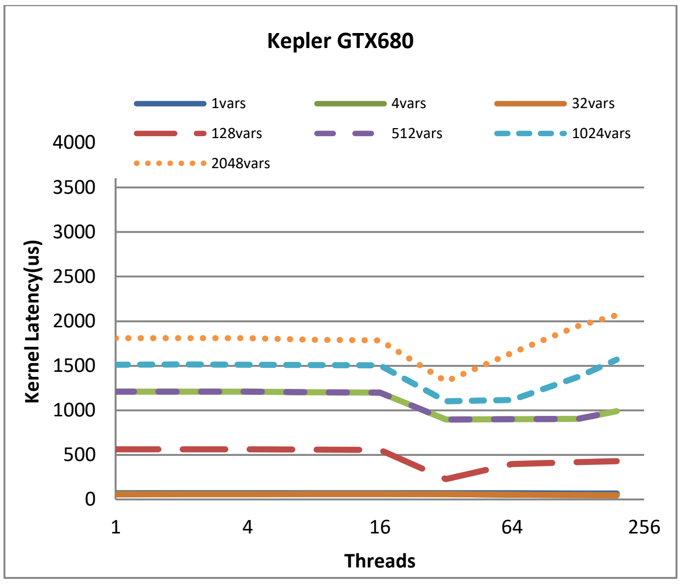

Figure 17. 4096 reads and writes with varied number of long variables w.r.t number of threads for Kepler GTX 680

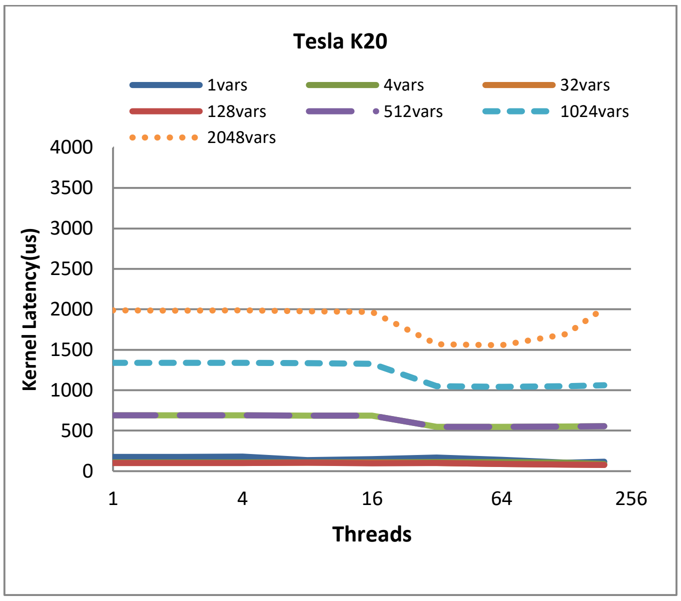

Figure 18. 4096 reads and writes with varied number of long variables w.r.t number of threads for Tesla K20

When examining the above results, some trends can be extracted from the figures above:

1. For all systems, the latency has been increased for benchmarks with large number of variables – the phenomenon was even more significant in cases for which number of threads was more than 32. When re-examining tables 1+2, there are 32 threads per warp. This indicates that the combination of multiple warps with large number of variables stresses the register file.

2. For all systems except the K20, the significant increase in latency started when the number of variables crossed 64, regardless of the number of threads - implying that most systems support a register file that is able to store 64 user variables (512B of storage) per thread. For the TeslaK20 system, the significant increase started for 512 bytes – this suggests that the K20 allocates for larger storage for variables than its predecessors (although table 2 states both GTX 680 and K20 has the same register file size).

3. For 32 threads and less, the increase in the number of concurrently running threads does not significantly affect performance for a small enough number of variables. Thus, the dominating factor for performance is the number of variables (and registers) used - implying that all systems allocate per-thread registers to store variables. For more than 32 threads, the dominating factor was still the number of variables but an additional increase in latency is seen as number of threads increase, indicating an overhead of a possible arbitration mechanism between executing warps.

4. When examining single thread performance, one can notice that for 2048 long variables (total size of 16KB), there is a significant slowdown of about 1140%. As shown in Table 1 and Table 2, this amount of data can easily fit in standard cache memories. However, as it appears in the figures, the considerable slowdown caused by the register spilling implies that all of the GPU systems that were tested do not cache the spilled variables, but just save them in another memory (slower than the cache) instead.

4. Conclusions

In this work several GPGPU benchmarks were written in the CUDA programming environment. The target was to provide a way for pinpoint the strengths and pitfalls in current commercial GPU systems with an emphasis on the memory hierarchy – these benchmarks are structurally independent, as there were no assumptions on any internal hardware structures or cache mapping schemes. This made it natural for the benchmarks to be executed on multiple GPU systems and motivated the cross-platform comparison made in this work, given 4 different NVIDIA GPU systems. The structural independence approach also made the benchmarks code highly robust, as it can be easily re-written and executed under other programming languages and to be tested on different GPU systems.

In spite of structural independence, combining with the information available in GPU specs these benchmarks unveiled some strengths and pitfalls that are the results of internal GPU structure.

Some of the results that were found - although fit one's intuition, were not yet quantified by previous studies:

1. The overhead of fine grained memory synchronization for threads belonging to the same block has a significant effect on kernel latency (therefore can affect overall utilization and throughput) - the slowdown measured was over 260% while an increase in number of threads results in a slowdown of at most 138% - one can deduct from these findings that the cost of fine grained synchronization can impede future progress in GPU performance, therefore synchronization should not be enforce automatically by hardware or software, but rather handled explicitly by GPU programmers.

2. Not all memory types supported by the GPGPU programming environment perform the same, as they can be stored on different memory elements and implement different caching mechanisms. Moreover, caching mechanisms for the same memory type might not be implemented in some systems while they are implemented in other systems. (Such as the case of global memory caching, which is implemented in 2.x systems, but not in 3.x).

3. The cost of registers spilling is not dominated by the number of threads – suggesting a per-thread register allocation policy.

Other results found in this works were somewhat counter intuitive:

1. The inability to coalesce concurrently executing global memory transactions has a significant effect as well. In the benchmarks, each thread accessed data in a locally sequential manner, reading a series of consecutive addresses from a single array. In cases where concurrently executing inter-thread accesses were too distant they interfered with execution, resulting in a slowdown of over 264%.This has implications on the way GPU programs are written: although each thread has a good spatial locality when executed independently, poor inter-thread locality (i.e. concurrently executing threads access distant memory regions) can result an unexpected performance bottleneck due to inefficient memory scheduling.

2. The newest GPU system tested, the Tesla K20, did not achieve the best results in most benchmarks: (i) the lack of caching mechanism for the global memory in CUDA 3.x systems caused the inability of memory coalescing to appear for a smaller number of threads than 2.x systems (ii) The K20 also suffered from the most significant performance loss due to fine grained global memory synchronization (iii) the average access time to both shared and global memories was high. These findings emphasize the importance of regression in future GPU systems, as some micro-architectural elements of modern GPU systems tend to perform worse than their predecessors.

3. The cost of register spilling is surprisingly high as it can reach an order of magnitude in performance loss – even though the overall number of storage needed by the benchmarks was relatively small (at most 16KB per thread). This suggests that the register spilling caused by the benchmarks was poorly handled, perhaps due to inefficient memory mapping of variables that exceed the initial per-thread register allocation.

5. Future Work

This work introduces an extensive study conducted as well as its insights. There are several aspects that can be taken from this study and to be further elaborated in future GPU studies: All kernels created in this work were written in the CUDA GPGPU programming environment which is supported NVIDIA's GPUs, a further study can be performed by rewriting the benchmarks' kernels in other programming languages, such as the OpenCL programming language, that can run on other devices – allowing a quantitative study of other manufacturers such as Intel and ATI.

In all kernels, inner block behavior was tested – possible future directions for this work should include cross block behavior and its effect on performance such as register spilling for inter-block threads, and the effects of grid level memory synchronization. Another possible extension to this work can be done is a thorough analysis of dynamic execution - to assure that benchmarks behaved as expected, a static analysis of PTX dumps was done for most of the kernels created, future studies elaborating this work should might gather additional insight from a dynamic analysis using various mechanisms such as performance counters and runtime profiling tools. Once these benchmarks are more comprehensively studied, they can be extended to a more comprehensive package that can be used as an architecturally neutral benchmarking suite which serves as point of reference for cross-platform GPU comparison.

6. References

[1] A. Grove, “Changing vectors of Moore’s law”, IEEE International Electron Devices, June 2002.

[2] CUDA programming language home, NVIDIA corporation: http://www.NVIDIA.com/object/cuda_home_new.html

[3] "OpenCL - The open standard for parallel programming of heterogeneous systems", Khronos group http://www.khronos.org/opencl/

[4] N. Goswami, R. Shankar, M. Joshi, and T. Li, “Exploring GPGPU workloads: characterization methodology, analysis and microarchitecture evaluation implication”, in Proceedings of IEEE International Symposium on Workload Characterization (IISWC), December 2010.

[5] A. Lashgar and A. Baniasadi. "Performance in GPU Architectures: Potentials and Distances". 9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD), June 2011.

[6] H. Wong, M.-m. Papadopoulou, M. Sadooghi-alvandi, and A. Moshovos, “Demystifying GPU Microarchitecture through Microbenchmarking,” 2010 IEEE International Symposium on Performance Analysis of Systems and Software ISPASS ’10, March 2010.

[7] M.M. Papadopoulou, M.S. Alvandi, H. Wong, "Micro-benchmarking the GT200 GPU",

http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/microbenchmark_report.pdf

[8] CUDA programming guide, NVIDIA corporation: http://docs.NVIDIA.com/cuda/cuda-c-programming-guide/

[9] "GP-GPU: General Purpose Programming on the Graphics Processing Unit", Computer vision and Geometry group, ETH Zurich, spring 2011.

http://www.cvg.ethz.ch/teaching/2011spring/gpgpu/