We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

4

Facts, Issues and Questions - GPUs for Dependability

Bernhard Fechner
FernUniversität in Hagen
Parallel Computing and VLSI Group

1. Introduction

Graphics Processing Units (GPUs) offer massive parallelism, comprising many actual paradigms like manycore, multithreading and SIMD. Today, nearly every computer is equipped with at least one graphics card, containing one or more GPUs bringing massive parallelism to the desktop. GPUs are usually used in their main function, that is, to compute visibility, lightning, perspective, etc. in games. As this technology is widely used, it is low-cost. In the majority of the cases, graphic cards do not spend their entire lives by executing game code. Thus, such a massive parallel system is underchallenged most of the time. Shortly after the availability of comfortable programming environments, based on CUDA (Compute Unified Device Architecture) or HLSL (high-level shader language), researchers have become interested in using this power for general-purpose computing (GPGPU, General-Purpose computing on the GPU). Thus, different applications originated, e.g. physics, cryptography 0, DNA sequencing 0 and medical imaging. For further examples and overview, see 0 and 0.
The trend to compute such workloads with GPUs will go on as the DirectX 11 (compute) or the OpenCL 0 standards show. The fault-tolerant execution of (sensible) workloads on GPUs was – to the knowledge of the author – never proposed. Sensible computations should be carried out in a reliable way. What is the sense of a computation to find a private key if the program is correct but the hardware is subjected to faults and the program never finds the key? E.g. transient faults can be caused from fluctuations in the main current, radiation or RAMs not running within their specification etc. What if an encryption is faulty due to temporal faults or how can we detect a faulty medical diagnosis? The need to do computations precisely has led to the development of more sophisticated and sometimes expensive graphics processing units 0, needed by CAD applications. Larrabee 0 is a many-core visual computing architecture. It uses multiple in-order x86 CPU cores that are augmented by a wide vector processor unit, as well as some fixed function logic blocks. This provides much higher performance per watt and per unit of area than out-of-order CPUs on highly parallel workloads. Vision4ce 0 launched a new line of General-purpose Rugged Image Processing (GRIP) products at the recent SPIE Defense and Security Symposium. The GRIP-Beta showed GPGPU-based image processing demonstrations, analog and Gigabit

Ethernet video streams and the functionality in the Gripworkx image processing framework. Vision4ce addresses rugged embedded computing challenges that might normally be served by more expensive FPGA approaches.

This work presents fundamental research, answering the question of how a system, equipped with multiple graphics cards can be harnessed to detect, predict, prevent and tolerate faults. Naturally, we do not restrict ourselves to computations running on the GPUs alone and also consider the outsourcing of application parts from the CPU to the GPU. We are aware of the fact that this evaluation can only be exemplary – but it can serve as a starting point and a priming of future work. All mechanisms are fully implementable in software and do not require special or modified hardware.

This work is structured as follows: we first present examples of current GPU implementations in Section 2. Section 0 shows how the massive parallelism of modern GPUs can be exploited for dependability. Section 0 summarizes and concludes the chapter.

2. Case Study and Programming Model

2.1 Case Study: The NVidia GeForce 8800 GTX

In this Section, we describe the basic architecture of the G80 GPU family from NVidia as this will help to understand the possibilities for dependability. The GeForce 8800 GTX is divided into 16 *streaming multiprocessors* (SMs), each containing eight *streaming processors* (SPs), making a total of 128 SPs. Each SM has 8,192 registers that are shared among all threads assigned to the SM. The threads on a SM core execute in SIMD (single-instruction, multiple-data) fashion, with the instruction unit (IU) broadcasting the current instruction to the eight SPs. Each SP has one arithmetic unit that performs single-precision floating point arithmetic and 32-bit integer operations. Fig. 1 shows an overview of the GeForce 8800 GTX.

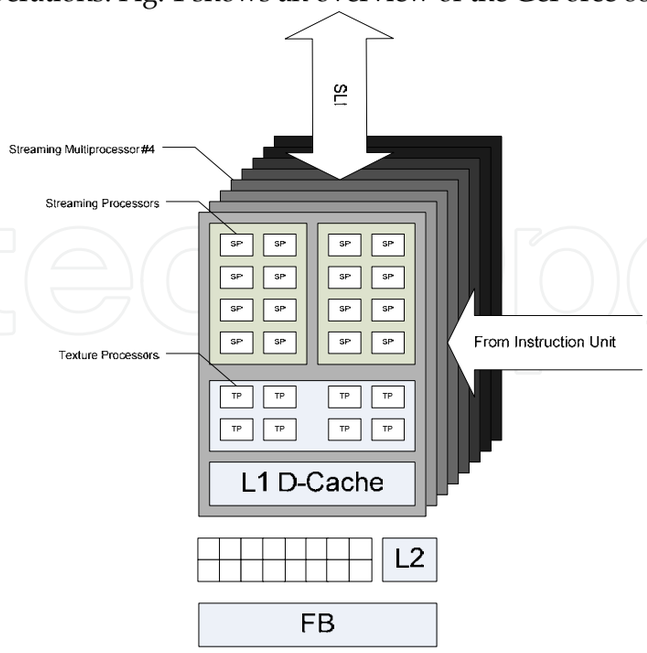

Fig. 1. The NVidia GeForce 8800 GTX

Each SM has two *special functional units* (SFUs), which perform more complex FP operations such as transcendental functions. The arithmetic units and the SFUs are fully pipelined. Each FP instruction is operating on up to 8 bytes of data. An important factor that affects both performance and quality is the precision for operations and registers. The GeForce Series support 32 bit and 16 bit floating point formats (called float and half, respectively) 0. The float data type resembles IEEE754 (s23e8), half has an s10e5 format. Some models, e.g. the G200 also support double precision in IEEE754R-format (one double-precision unit per SM). The processors support gathering and scattering. Thus, they are capable of reading and writing anywhere in local memory on the graphics card or in other parts of the system. The G80 has several on-chip memories that can exploit data locality and data sharing, e.g. a 64 KB off-chip *constant memory* and an 8 KB single-ported constant memory cache in each SM. If multiple threads access the same address during the same cycle, the cache broadcasts the address to those threads with the same latency as a register access. In addition to the constant memory cache, each SM has a 16 KB *shared (data) memory* that is either written and reused or shared among threads. Finally, for read-only data that is shared by threads but not necessarily to be accessed simultaneously, the off-chip texture memory and the on-chip texture caches exploit 2D data locality.

2.2 The CUDA Programming Model

The CUDA programming model consists of ANSI C supported by several keywords and constructs. CUDA treats the GPU as a coprocessor that executes data-parallel kernel functions. The developer supplies a single source program encompassing both host (CPU, c) and kernel (GPU, cu) code. The host code transfers data and code to and from the GPU's global memory via API calls and initiates the kernel. At the highest level, each kernel creates a single *grid*, which consists of many *thread blocks*. Each thread block is assigned to a single SM for the duration of its execution. A thread block consists of a limited number of threads which can cooperate. The maximum number of threads per block is 512. Threads from different blocks cannot cooperate. Each thread can read/write from/to thread registers, thread-local memory, shared memory in a block, the global memory and read from *constant memory* or the texture memory in a grid. The host has read/write access on the constant, global and texture memory. Threads in the same block can share data through the shared memory and can perform barrier synchronization. Threads are otherwise independent, and synchronization across thread blocks is safely accomplished only by terminating the kernel.
The IU manages things in groups of parallel threads, called *warps*. SMs can perform zero-overhead scheduling to interleave warps on an instruction-by-instruction basis to hide the latency of global memory accesses and long-latency arithmetic operations. When one warp stalls, the SM can switch to a ready warp in the same or different thread block assigned to the SM. Each warp executes in SIMD fashion, with the IU broadcasting the same instruction to the eight cores on a SM on four consecutive clock cycles. Since one pixel equals one thread, and since the SPs are scalar, the compiler schedules pixel elements for execution sequentially: red, then green, then blue, and then alpha.
Fig. 2 shows a Thread Processing Cluster (TPC) used on the G200 series with 10 TPCs in total. As depicted, a TPC comprises multiple IUs, SPs and local memory.

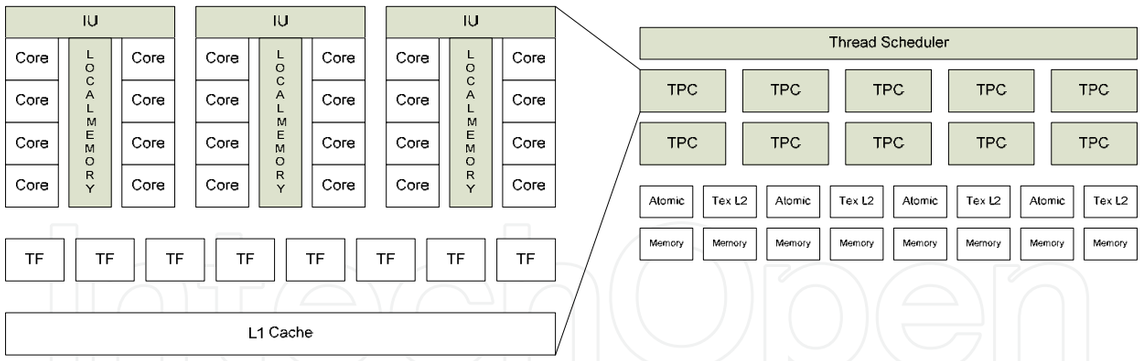

Fig. 2. A Thread Processing Cluster (TPC)

2.3 Experimental Setup and Clock Variation

In this Section we present the results from a first experimental evaluation by clock variation, since we wanted to artificially increase the fault rate, observe the system behavior concerning reliability and depict basic performance figures.

Our experimental setup consists of a 6 GB main memory Core i7 system, configured with two NVidia GTX260 cards (PCIe 2.0 x16). The two hard disks (500 GB) are in RAID 0 mode. In the first experiment with SLI, we adjusted the engine, shader and memory clock frequency. A SLI-system is constructed on hardware level and must be configured on software level. Either the GPUs work independently in non-SLI mode to support multi-view displays or all GPUs in a SLI configuration appear as a single unit, mainly used to speed up 3D applications and computations. For the CUDA programming environment, a non-SLI system appears as a set of graphics cards, a SLI system as one graphics card. Multiple GPUs appear as multiple host threads. The clock rate adjustment in SLI mode is done for both cards simultaneously, in non-SLI mode, both cards have to be configured separately. The maximum clock rate of (engine=800, shader=1650, memory=2700) MHz sometimes resulted in execution faults of a kernel in non-SLI mode and *complete system failures* in SLI mode. Therefore, we applied less aggressive settings and varied the clock frequency between (engine=500, shader=1150, memory=1900) and (700, 1400, 2500) MHz. The workload consisted of a computation of the blackscholes formula for 512 iterations. The same workload was also computed on the CPU. Besides precision issues (see Section 0) no deviation except for the highest clock settings occurred. Fig. 3 shows the influence of the variation of the clock frequencies of the engine, shader and memory on performance (SLI). Note that the bandwidth is the internal card bandwidth and not the bandwidth of the external interface (PCIe). From the experiments two simple but important conclusions can be derived:

1) a system in SLI mode is less reliable than one in non-SLI mode. Reliable calculations should be carried out on a non-SLI system. A SLI system has more advantage in computing-intensive applications. For bandwidth-intensive applications a non-SLI system should be preferred.
2) Within the overclocking experiments, the GPU rather tended to completely reject the execution of a kernel instead of doing faulty computations (overclocking applied at the beginning of the execution).

We are aware of the fact that these figures are only exemplary, but the results can serve as an orientation.

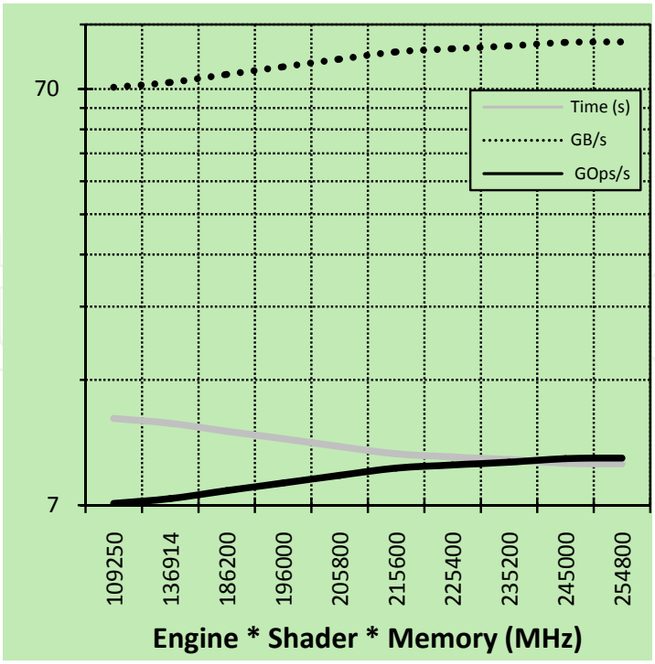

Fig. 3. System performance while varying clock frequencies

2.4 Bandwidth Experiments

The question in this Section is to determine the bandwidth in Mbytes per second for different transfer sizes and different configurations of a SLI and non-SLI system. The bandwidth is important e.g. when the results of a redundant computation must be transferred back to the CPU for a comparison. The basic bandwidths of PCIe 2.0 interfaces are depicted in Table 1.

PCIe-Slot	Lanes/ Direction	Bandwidth	Clock
x1	1	0.5 GByte/s	2.5 GHz
x4	4	2 GByte/s	2.5 GHz
x8	8	4 GByte/s	2.5 GHz
x16	16	8 GByte/s	2.5 GHz
x32	32	16 GByte/s	2.5 GHz

Table 1. Basic bandwidths of PCIe 2.0

Blocks with a certain size were either transferred from the host to the device, from the device to the host and from device to device. The maximum bandwidth for each device within the experiments is 8 GBytes/s. Fig. 4 shows the bandwidths for pageable and pinned memory. Pinned memory allows the compute kernels to access and share the host's memory. We applied the lowest clock settings (engine=500, shader=1150, memory=1900) to

determine a lower bandwith bound. From the results, we see that the host to device transfer (pinned memory) is the slowest form to transfer data, followed by the device to host (pageable) communication. Starting from block sizes greater than 65536 bytes, the device to device communication is the fastest way to transfer data.

Fig. 4. Bandwidth for different block transfer sizes

We note that the experimental bandwidth for the device to device communication is well above the limit of the PCIe 2.0 x16 specification. The reason for this is that transfers are done on the graphics card and do not pass the external PCIe bus.

2.5Precision Experiments

The realm of (COTS) GPUs is not precision, it is speed. Thus, applications running on GPUs must be questioned in general. Most GPUs use IEEE754R as floating-point format. In comparison to IEEE754 rounding occurs, leading to imprecision. But there are several work-arounds, including mixed-precision 0.

In this Section, we do not focus on rounding errors. We prefer an empirical analysis, since we do not know the implementation of the floating-point algorithms within the GPU. Especially the implementation of transcendental functions implies approximation algorithms, which we cannot know if we do not have a disclosure of the GPU implementation, which is not available to the public due to commercial reasons. To the knowledge of the author, this approach to examine the precision of GPUs is a novelty.

We present benchmarks to compute the deviation of GPU operations in comparison to a CPU implementation and regard three different data types: integer, float and double. Half-floats are supported by shaders and thus are not directly accessible by CUDA. As the half-float is inspired by IEEE754, infinity exists if all bits of the exponent are one and the

mantissa is zero. A half-float is a NaN if all exponent bits are one and the mantissa is not zero. The set of precision benchmarks can be downloaded from 0.
The benchmarks implement vector operations in dim(2^{24}) with different data types and operations, listed in Table 2. The vector data is randomized in each run. Each cell in Table 2 contains the maximum unsigned deviation from the CPU implementation. For computations which could cause overflows, such as the exponential function, the size of the numbers within the randomized vectors was limited.

Type	Single	Double	INT32
Add	0	0	0
Sub	0	0	0
Mul	0	0	0
Div	0.125	0	0
Sqrt	0.0000152588	0	0
Sin	0.000000119209	$1*10^{-16}$	0
Cos	0.000000119209	$1*10^{-16}$	0
Log	0.000000953674	$9*10^{-16}$	0
Exp	0.00195313	$4*10^{-16}$	0

Table 2. Maximum absolute deviation from CPU implementation

Astonishingly, basic arithmetic operations such as add and sub or mul and all integer operations do not lead to imprecision. From this, we can conclude that a scaling of small floats to integers can improve the precision in such a way that the CPU and the GPU results will not differ.

2.6Timing and mid-term Experiments

In this Section, we present the results of mid-term experiments to determine the timing variance and reliability/ stability of results. By a mid-term evaluation, we mean an observation interval of one week. A longer observation interval, e.g. over more than one month would be appreciated, but was not feasible due to the timely restrictions of this work. The precision benchmarks from subsection 0 were calculated 18^5 times. Additionally, we calculated the workload on the CPU with one core and a parallelized version on the 8 available cores. We measured the time for each calculation, GPU and CPU and calculated the average arithmetic mean. The graphics cards were configured in non-SLI mode. The results are depicted in Fig. 5. For some cases (INT operations), the OpenMP implementation was even faster than the GPU.

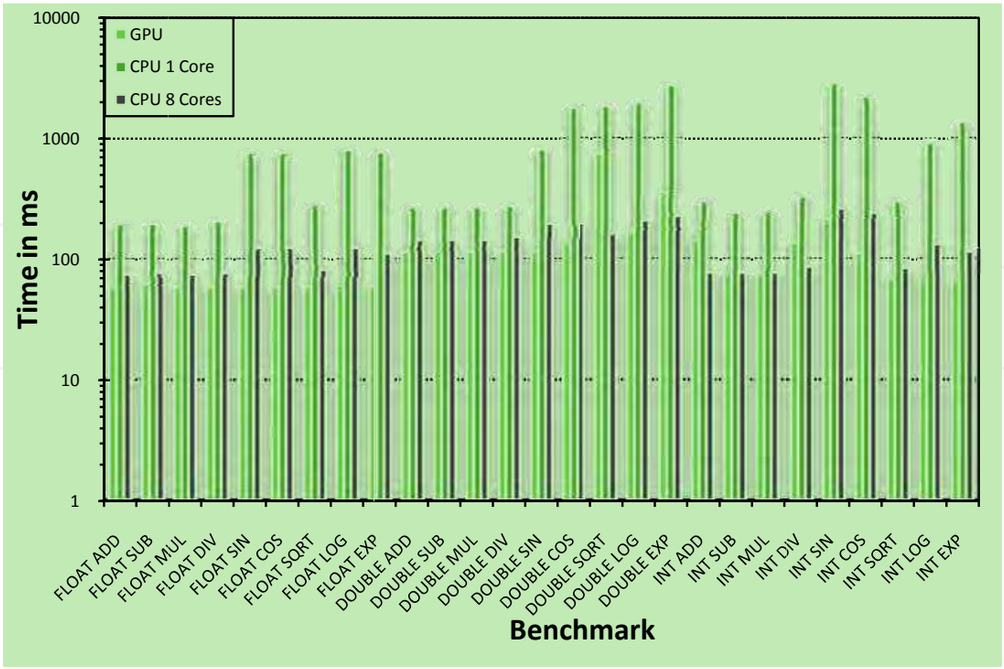

Fig. 5. Timing Results

From the results, we first see that complex operations and transcendental functions need more time – which is not surprising, since there are only two SFUs on one SM. Integers approximately need twice the same time than floats, doubles approximately twice the time than integers (in average). We noticed that the timings varied for both implementations, GPU and CPU. Thus, we calculated the maximum, minimum and average timing values for both implementations, GPU and multicore (even columns), depicted in Fig. 6. (floats).

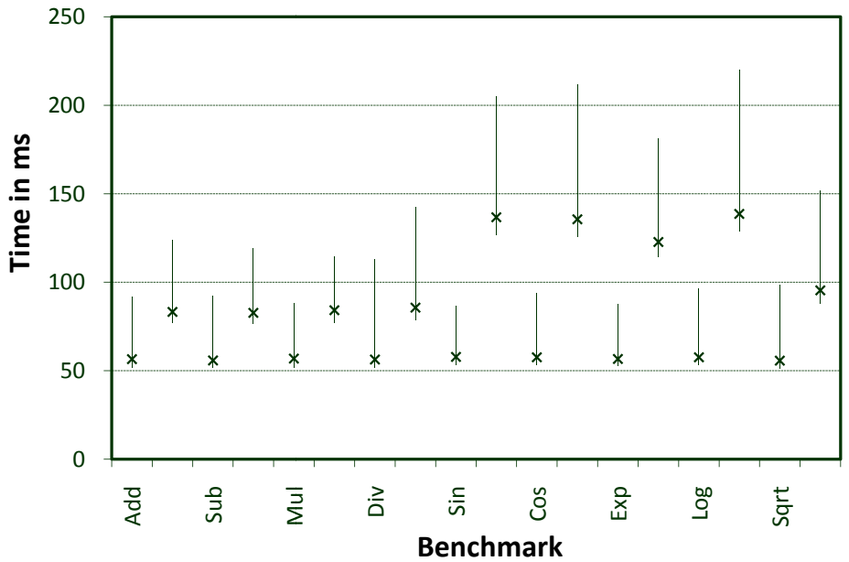

Fig. 6. Benchmark timing, variation, lowest, highest, average

The variations are e.g. caused by normal user interactions. We conclude that results cannot be expected at a certain time. Thus, computations on graphics cards may not be currently suitable for realtime applications. Interesting is that the timings from CPU and GPU have a connection, i.e. if the timing for the GPU was large, the timing of the corresponding CPU implementation was also higher. The deviation resulted in each run and the results seem to correlate. This is surprising, since we implemented an asynchronous version for the GPU which ran independently from the CPU. During the experiments, no unusual deviation (except precision) between CPU and GPU occurred. The results were stable during the whole observation period.

3. Opportunities for Dependability

In this Section, we will discuss the opportunities for dependability offered by graphics cards. Note, that our terminology is based on 0. We will first have a look at the section *means* from the dependability tree (from 0) in Fig. 7. Then we will discuss the means fault prevention, fault-tolerance, fault removal and fault forecasting in the following subsections. We do not specify the exact nature (e.g. bit-flip faults, transmission faults, permanent) of faults within a model, since we do not want to restrict our horizon by regarding at a special set of fault types but we are aware of the fact, that a fault model has to be developed later on.

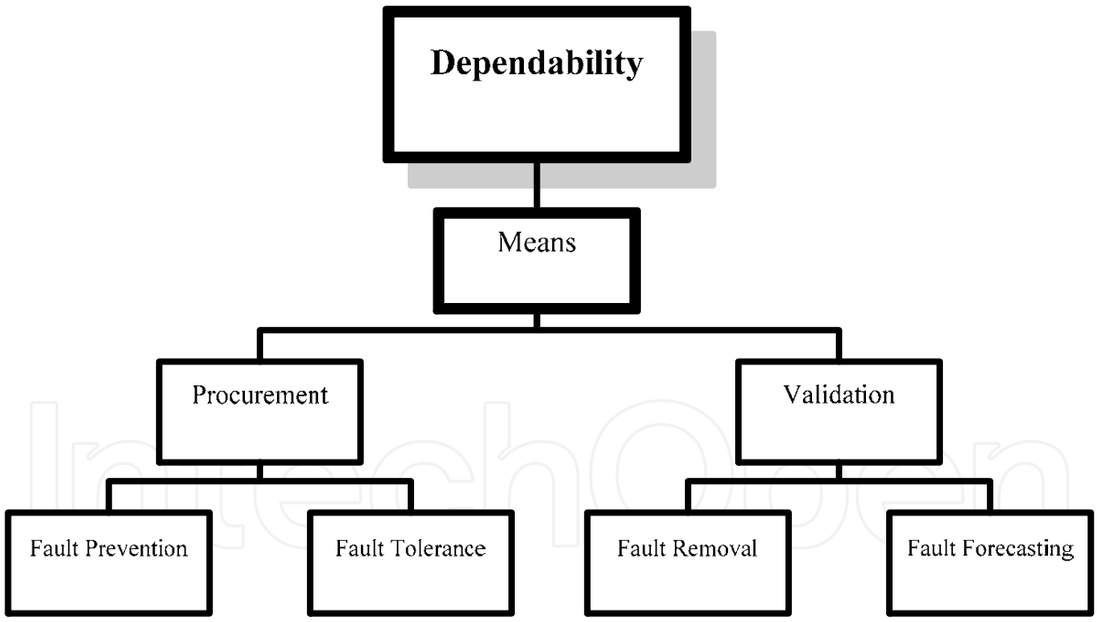

Fig. 7: A section from the dependability tree

We distinguish different levels on which different dependability means can be applied. Therefore, we depict the notational conventions in Table 3 and note the level, where zero (0) means the top level.

Level	Name	Meaning
0	Host	The host or the system; a computing system containing one or more CPUs and graphics hardware
Integrated Computing Hardware		
1	CPU	The central processing unit
2	Processing core	A core within a CPU
3	Thread	A hardware thread, consisting of registers etc.
Graphics Hardware		
1	Device	A single graphics card
2	GPU	A graphics processing unit
3	GP core	A core within a GPU
4	Grid	A set of thread blocks
5	Thread Block (TB)	A thread block consists of multiple threads

Table 3. Notational Conventions

3.1 Fault Prevention

We note that the development of an additional GPU kernel, doing the same task as the CPU at the same time, automatically involves diversity in hardware, software and design, since through different implementations and by using different compilers, we have diversity, considering the fact that we have only one system, but multiple versions of a program and multiple hardware realizations. Note, that the forecast of faults can also be seen as essential part of fault prevention (see Section 0 for details).

3.2 Fault-Tolerance

Basic means of fault-tolerance are structural, temporal, informational and functional redundancy. Naturally, all codes involving informational redundancy can be computed by graphics cards. An interesting idea is to speed up the calculation of Reed-Solomon-Codes by GPUs 0. Functional redundancy can be easily achieved by either computing a calculation on the CPU and the GPU, involving diversity in software or by programming a set of functions again for the GPU. When voting between the results, we can use the inherent voting capability supported by CUDA.

3.2.1 Structural Redundancy

Structural redundancy can be achieved by integrating multiple graphics cards into a single computing system. The result is massive redundancy, e.g. via dual, triple, quadruple configurations. Naturally, we are not able to tolerate permanent CPU faults, but permanent GPU faults. Note, that it is also possible to combine mainboard GPUs and external graphics cards. One should be aware of the fact that multiple (PCIe) graphics cards can be installed simultaneously, deriving diversity in hardware. The multiprocessing-paradigm has also arrived for GPUs. NVidia's GeForce 9800 GX2 contains a pair of 65 nm G92 graphics

processors running at 600 MHz. The ATI Radeon™ HD 4870 X2 has two 55 nm GPUs, a 512-bit GDDR3 memory interface and the option to construct a dual-mode CrossfireX configuration, resulting in a total of four GPUs. To lower physical dependencies, one should carry out redundant computations on different cards, then on different GPUs, then on different grids. The program/ operating system can additionally implement a scheduler, issuing different redundant computations to different parts of the graphics subsystem. The redundant computations can be called from the main program and run in parallel to the CPU calculation. A comparison can be done by the CPU or the GPU. However, the production of results must be synchronized. Fig. 8 shows the integration. Disadvantages besides synchronization are that the user must decide which code should be verified by the GPU and the source code of the application must be modified. Additionally, only system relevant routines could be modified.

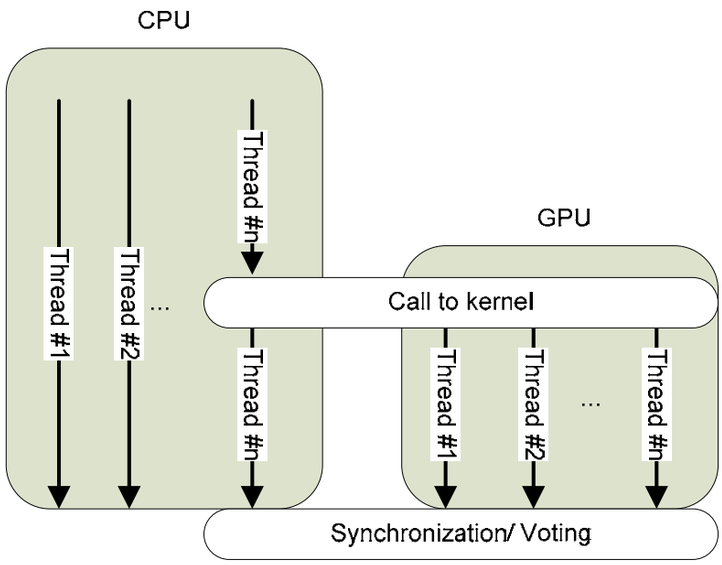

Fig. 8. CPU/ GPU redundant computations

From Fig. 8 we see that the combination of multiple host thread callers and GPU threads is possible. To do a synchronization without waiting times for the CPU and/ or the GPU, the results could be written into a buffer, where each calculation receives its very own identification. Thus, we do not have to wait for the results to arrive. A disadvantage is that in case of a rollback, already calculated results must be discarded. The synchronization of host and GPU threads offers a new perspective for research.

3.2.2 Temporal Redundancy

Temporal redundancy is an essential property of a multithreaded system, thus also for graphics cards comprising hundreds or thousands of threads. A temporal redundant computation can be done on every accessible element of the graphics card by redoing the calculation on the same or (better) on a different component. The only point where structural or temporal redundant threads are dependent is at the checking of results. The implementation in software is difficult, since CUDA does not differ between physical and

virtual threads. Here, data dependencies have to be regarded. Fig. 9 illustrates two possible forms of temporal redundancy.

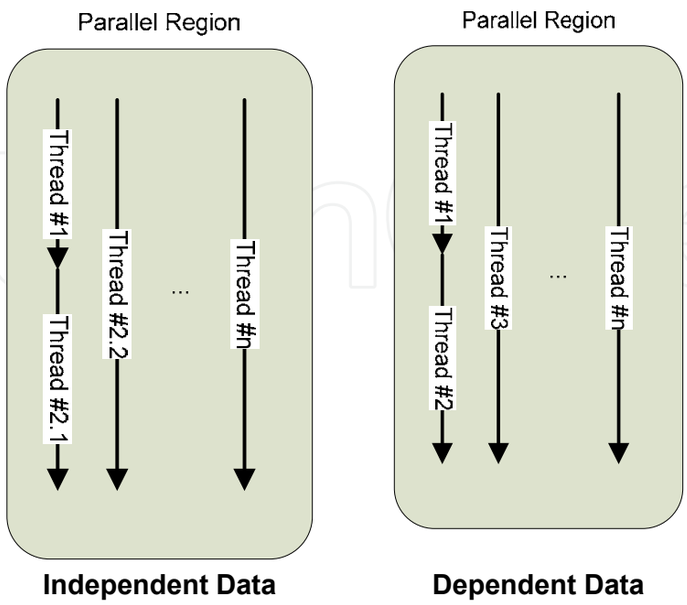

Fig. 9. Temporal Redundancy, Data Dependencies

Temporal redundancy on GPUs leads again to synchronization problems.

3.3 Fault Removal

Apart from the ECC fault removal within the GPU memory 0, fault removal is a hard thing to implement by using GPUs, because the faulty unit must be located and a prior and sane state must be restored. On a fault-free computation we must store a checkpoint. Here, we can fallback to classical schemes storing the checkpoint on hard disks or to store the checkpoint on the cards. The first thing is to use a triple card configuration to detect, locate and remove the fault within the graphics configuration. Here, the graphics cards ought to execute the same code, not strictly synchronously, but in a way that faults cannot propagate between cards. We do not discuss the removal of faults initiated from the CPU to the GPU ($CPU \Rightarrow GPU$) here, since our aim is to assist CPU calculations.

3.3.1 Watchdogs

A GPU can be periodically triggered by an external timer to monitor activities. The timer routine must be able to directly access the memory of the graphics card. The external timer is needed, because GPUs do not possess such a capability at the moment. The activities are e.g. CPU or fixed disk functionalities. Any activity and the current time are e.g. written to the texture memory. On a write of the current time, the last time will be copied to a different location within memory. If the new timer value does not differ from the last one, a fault is signaled. Furthermore, the GPU checks the activities. If no activities are recorded in the timer interval (no value has been written to memory) a wakeup signal can be issued. Fig. 10 shows the algorithm.

```
Startup:              E={}     // Empty event list E in mem
On timer:             // Compare new timestamp N with previous P in mem
         If N>P:      write P to previous timestamp in mem (P=>PP)
        Else Signal "Timer Fault"
             If E={}:Signal "Event List Empty - Wakeup"

On event:     Write event to E // Note that timer is also an event
```

Fig. 10. Watchdog Algorithm

The wakeup signal can be issued by writing to a dedicated memory location within the host's memory. If no OS restrictions apply, the GPU could write the recovery entry address to the CPU program counter.

3.3.2 Fault Removal GPU ⇒ GPU

We can imagine something like a RAIGx configuration (Redundant Array of Independent Graphics cards, according to a RAIDx – Redundant Array of Independent Disks). As we have not hardware controller to support RAIG, we only support Software-RAIG. As RAIG, we can consider the usual modes, listed in Table 4.

Mode	Meaning, Configuration
0	Two or more graphics cards doing independent calculations
1	Two of more graphics cards doing the same calculations in parallel
5	Two graphics cards doing the same calculations in parallel, securing the operands and the results in memory by a checksum, e.g. parity

Table 4. RAIG Modes

On the detection of a fault, we can vote among the results. If we include the CPU in the calculations, we have a TMR configuration and therefore can locate the faulty unit, if two results are equal. If the kernels are data independent, we can simply continue. If we have dependencies among the calculations, we have the option to either copy all memory contents and processing states of an assumed fault-free card to other all cards or copy the modified parts (see subsection 0).

3.3.3 Removal GPU ⇒ CPU

Fault removal within a CPU from a GPU is possible but far more difficult. CPU states must be written into the memory of the graphics card, also updated memory locations. We suggest checkpoint intervals between 10^6 (~4 MBytes written) and 10^7 (~40 MBytes written) memory writes. The checkpoint interval is restricted by the main memory of the graphics card, expected reliability and system performance. The CPU state is also stored in the main memory of the card. On a fault, the memory and CPU state must be transferred back. In Fig. 4 it is shown what bandwidth can be achieved. Since we cannot usually map the whole main memory of the host to the device memory, since it is smaller than that of the host's memory, we must either do every memory write of the CPU simultaneously on the card, significantly decreasing performance or do a fault removal for a single (system relevant) application running on the CPU such as a daemon. For CPU states, there is no problem, because the

amount of data to transfer is very small. Difficult is the injection of a previous state in the CPU. Here we can imagine a state memory for each CPU which can be written from the GPU and read by the CPU. Within a multicore system another (healthy) CPU can inject the state into the faulty CPU.

3.4 Fault Forecasting (with GPUs)

For the prediction of faults, a history of faults must be stored in the graphics card memory, because without knowledge of the past, we cannot predict future faults. The prediction can be done with various methods, e.g. causal Bayesian networks, Hidden Markov Models (HMMs) and the forward algorithm, etc. We propose to use the MCE (machine check exception) of modern processors to enter a special routine to compute the prediction. We assume the history to be organized as simple ring buffer of length N. The algorithm in Fig. 11 briefly sketches the method without going into details.

```
Startup:  History (h2) location h=0;

CPU:
         On_MCE:   Write MCE-Flag, time to GPU memory, location h2
           h=h+1 % N
           Call prediction on GPU
GPU:
  On_Call: Do prediction using h2
```

Fig. 11. Basic (abstract) prediction of faults

Note, that the forecast with HMMs implies very small numbers and hence precision problems. A small deviation can lead to faulty results. The scaling to big integers can limit these effects.

4. Summary and Outlook

This work presents a first step and innovative approach to use GPUs for dependability. We are aware of the fact that this work is rudimentary – but it can serve as a starting point and a priming of future work. It has been shown how the existing parallelism of GPUs can be exploited for dependability. Although we did not specify the exact nature of faults, since we did not want to restrict our horizon by regarding at a special set of fault types, the results and the physical context of the experimental setup strongly suggest to model transient faults. To lower physical dependencies, one should carry out redundant computations on different cards, then on different GPUs, then on different grids. From the experimental results some conclusions can be derived: a system in SLI mode is less reliable than one in non-SLI mode. Reliable calculations should be carried out on a non-SLI system. A system configured in SLI has more (proven) advantage in computing-intensive applications. For bandwidth-intensive applications a non-SLI system should be preferred. During the mid-term experiments, no unusual deviation (except precision) between CPU and GPU results occurred. The results were stable during the whole observation period.
Not everything is golden in this new world of opportunities. There are a few critical points which must be regarded by future research:

- The precision of results: fortunately all basic arithmetic operations such as add, sub and mul and all integer operations do not lead to imprecise results. A scaling of small floats to integers can improve the precision in such a way that the CPU and the GPU results will not differ.
- The synchronization of host and GPU threads offers a whole new perspective for research. The varying timings from CPU and GPU have a connection per computation, i.e. if the timing for the GPU was large, the timing of the corresponding CPU implementation was also higher. This is surprising, since we implemented an asynchronous version for the GPU which ought to run independently on the CPU. In their current implementation, graphics cards are not suitable for realtime applications.

Future work will include the implementation and analysis of the discussed dependability means and a long-term reliability evaluation.

5. References

[1]ACM Queue, GPUs Not Just for Graphics, Vol. 6, No. 2, March/ April 2008, ISSN: 1542-7730.

[2]J.-S. Huang et al. (NVidia corporation), United States Patent 7053901, System and method for accelerating a special purpose processor

[3]GPGPU. *General-Purpose Computation Using Graphics Hardware*, http://gpgpu.org, checked 05/15/2008.

[4]NVidia. *Technical Brief. NVidia GeForce 8800 GPU Architecture Overview*, Nov. 2006. http://www.NVidia.com/object/IO_37100.html, checked 05/15/2008.

[5]Larrabee: A Many-Core x86 Architecture for Visual Computing. Seiler, L., Carmean, D., Sprangle, D., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P. Proceedings of SIGGRAPH 2008.

[6]www.vision4ce.com, checked 06/16/2009.

[7]Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A. (2007). "*High-throughput sequence alignment using Graphics Processing Units*". BMC Bioinformatics ***8:474****: 474.* doi:10.1186/1471-2105-8-474

[8]J.C. Laprie, *Dependability: Basic Concepts and Terminology* Springer-Verlag, 1992. ISBN 0387822968

[9]I. Pharr, Matt. II. Fernando, Randima. *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*, edited by Matt Pharr; Randima Fernando, series editor. ISBN 0-321-33559-7.

[10]John D. Owens et al. *A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum*", 2007, http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x , pp. 80-113, vol. 26 , no. 1

[11]Khronos OpenCL Working Group. *The OpenCL Specification. Version: 1.0, Revision: 33,* Aaftab Munshi (ed.), http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf

[12]http://pv.fernuni-hagen.de/~fechner/GPU.html, checked 06/03/2009

[13]http://www.NVidia.de/page/tesla_computing_solutions.html, checked, 06/03/2009.

[14]Curry, M.L.; Skjellum, A.; Ward, H.L.; Brightwell, R. *Accelerating Reed-Solomon coding in RAID systems with GPUs.* In Proc. Of the IEEE International Symposium on Parallel and Distributed Processing, pp. 1 – 6, 2008.

[15]R. Strzodka, D. Göddeke. *Mixed precision methods for convergent iterative schemes*. In Proc. of the 2006 Workshop on Edge Computing Using New Commodity Architectures, pp. D–59–60, 2006.

[16]A. Moss, D. Page, N. Smart, *Toward Acceleration of RSA Using 3D Graphics Hardware*. Cryptography and Coding, pp. 369–388. December 2007.

[17]N. Maruyama, A. Nukada, S. Matsuoka, Software-Based ECC for GPUs, Symp. on Application Accelerators in High Performance Computing, 2009.

Parallel and Distributed Computing

Edited by Alberto Ros

ISBN 978-953-307-057-5

Hard cover, 290 pages

Publisher InTech

Published online 01, January, 2010

Published in print edition January, 2010

The 14 chapters presented in this book cover a wide variety of representative works ranging from hardware design to application development. Particularly, the topics that are addressed are programmable and reconfigurable devices and systems, dependability of GPUs (General Purpose Units), network topologies, cache coherence protocols, resource allocation, scheduling algorithms, peertopeer networks, largescale network simulation, and parallel routines and algorithms. In this way, the articles included in this book constitute an excellent reference for engineers and researchers who have particular interests in each of these topics in parallel and distributed computing.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen