1. Introduction

Modern Graphics Processing Units (GPUs) offer massive parallelism, comprising many actual parallel 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, DNA sequencing [7] and medical imaging. For further examples and overview, see [3] and [10].

The trend to compute such workloads with GPUs will go on as the DirectX 11 (compute) or the OpenCL [11] 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 transient or computational faults and the program never finds the key? Transient faults can e.g. 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?

This work presents basic and fundamental research, answering the question how a system, equipped with multiple graphics cards can be harnessed to detect, predict, prevent and tolerate faults. We do not restrict ourselves to computations running on the GPUs alone. We also consider the outsourcing of application parts from the CPU to do redundant and thus dependable computations on GPUs. We are aware of the fact that the 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 review related work in Section 2 and present current GPU implementations and introduce an abstract GPU model. Section 3 shows how the massive parallelism of modern GPUs can be exploited for dependability. Section 4 summarizes and concludes the paper.

2. Related Work

Due to the novelty of this work, there are no existing hints to use graphics cards for dependability. We therefore briefly review recent approaches in many-core computing. Larrabee [5] 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 dramatically higher performance per watt and per unit of area than out-of-order CPUs on highly parallel workloads. Vision4ce [6] 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 robust functionality in the Gripworkx image processing framework. Vision4ce addresses rugged embedded computing challenges that might normally be served by more expensive and lengthy FPGA approaches.

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), running at 1.35 GHz. The 16 multiprocessors are clocked at 675 MHz. The architecture supports the single-program, multiple-data (SPMD) programming model. 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 broadcasting the current instruction to the eight SMs. Each SM 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.
Each SM has two special functional units (SFUs), which perform more complex FP operations such as the inverse square root and trigonometric functions. The arithmetic units and the SFUs are fully pipelined. Thus, each SM can perform 18 FLOPS per clock cycle (one multiply-add operation per SP and one complex operation per SFU). Each FP instruction is operating on up to 8 bytes of data. Integer multiplications are also processed by these two units. A lower precision version (24 bit instead of 32 bit) can be executed with the 8 SPs. An important factor that affects both performance and quality is the precision used for operations and registers. The GeForce Series support 32 bit and 16 bit floating point formats (called float and half, respectively) [9]. The float data type resembles IEEE754, (s23e8), half has an s10e5 format. The latest G200 supports double precision. A question here is if there are some factors which influence the precision and the absolute error of a computation.

The processors support gathering and scattering. Thus, they are capable of reading and writing anywhere in local memory on the graphics card or elsewhere (other parts of the system). The G80 has several on-chip memories that can exploit data locality and data sharing, e.g. 64 KB off-chip constant memory and 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’s value 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 memory that is useful for data that is either written and reused or shared among threads. Finally, for read-only data.
that is shared by many threads but not necessarily accessed simultaneously by all threads, the off-chip texture memory and the on-chip texture caches exploit 2D data locality. Threads executing on the G80 are organized into a three-level hierarchy. At the highest level, each kernel creates a single grid, which consists of many thread blocks. The maximum number of threads per block is 512. A kernel is executed on a grid of thread blocks, whereas a thread block consists of a limited number of threads which can cooperate. 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 constants and the global and texture memory. Each thread block is assigned to a single SM for the duration of its execution. 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. Finally, threads within a block are organized into warps of 32 threads. Each warp executes in SIMD fashion, with the SM’s instruction unit broadcasting the same instruction to the eight cores on four consecutive clock cycles. Fig. 2 shows a Thread Processing Cluster (TPC). 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 thread block or a ready warp in some other thread block assigned to the SM.

Fig. 2: A Thread Processing Cluster (TPC) within a system
2.2. An Abstract GPU Model

Graphics processing units use massive parallelism by using multiple cores having different capabilities and threads. Within a GPU there can be different cache-levels. A graphics card can be connected to other graphics cards in a system (e.g. Crossfire or SLI (Scalable Link Interface)) and can have different caches, e.g. a texture cache, instruction and data caches. We first note that every set in this work is finite and nonempty, unless otherwise stated. Let $\forall i \, G_i \neq H_i = (M_i, N), \, i \geq 0$ be $\text{Nei}$ units on level $i$, containing a number of memory cells $M_i$ (size in Bytes). Note, that a unit is not specified through its capabilities. Let $F_i \subseteq G_i \times H_i$ be the interconnect function. Thus $F_i$ is an undirected and acyclic graph. Note, that we have no start or end point.

2.3. 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) and kernel (GPU) code. NVIDIA’s compiler, nvcc, separates the host and kernel codes, which are compiled by the host compiler and nvcc, respectively. The host code transfers data to and from the GPU’s global memory via API calls and initiates the kernel. Tuning the performance of a CUDA kernel often involves a fundamental trade-off between the efficiency of individual threads and the thread-level parallelism (TLP) among all threads. For example, as each thread’s register usage increases, the total number of threads that can simultaneously occupy the SM decreases. Because threads are assigned to an SM not individually, but in large thread blocks, a small increase in register usage can cause a correspondingly much larger decrease in SM occupancy.

2.4. Experimental Setup and Clock Variation

In this Section we present the results from an experimental evaluation, since we wanted to determine basic capabilities and basic performance figures of the used graphics cards. We are aware of the fact that these figures can only be a starting point, but the data can serve as orientation. We first describe our experimental setup to determine the average fault rate, followed by a description of the workloads. Our setup consists of a 6 GB main memory Core i7 system, configured with two NVidia GTX260 cards. 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. 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 (800, 1650, 2700) MHz sometimes resulted in execution faults (non-SLI)
and complete system failures in SLI mode. Therefore, we applied less aggressive settings and varied the clock frequency between (500, 1150, 1900) and (700, 1400, 2500) MHz. The workload consisted of a computation of the blacksholes formula for 512 iterations. The workload was computed again on the CPU. We measured the bandwidth, the number of operations per second and the average absolute error, compared with the CPU implementation.

For 32 bit floats, we measured the error to be constantly at 0.0000152587890625 in comparison to a CPU implementation. This is because many GPUs do not work according to the IEEE754 standard. Of course, the pushing of clock frequencies to their limit resulted in a much higher absolute error (up to 60). Fig. 3 shows the influence of a variation of the clock frequencies of the engine, shader and memory on performance (SLI).

![Fig. 3: System performance while varying the clock frequency of engine, shader and memory](image)

### 2.5. Bandwidth Experiments

The question in this Section is to determine the bandwidth in Mbytes per second on different transfer sizes and different configurations of a SLI and non-SLI system. A SLI system can be constructed on a hardware level and is 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 number of host threads is independent from the number of threads running on the GPU. Blocks at a certain size were either transferred from the host to the device, from the device to the host and from device to device. Fig. 4 shows the results for all configurations and block sizes. To get a clearer picture, Fig. 5 shows the average transfer rate for each configuration.

![Graph showing bandwidth in MBytes/s for different block transfer sizes and configurations](image)

**Fig. 4: Bandwidth in MBytes/s for different block transfer sizes and configurations**

![Graph showing average bandwidth for all configurations](image)

**Fig. 5: Average bandwidth for all configurations**
Table 1 shows the arithmetic average execution time, bandwidth and operations per seconds from the experiments in Fig. 5.

<table>
<thead>
<tr>
<th></th>
<th>Time (s)</th>
<th>GB/s</th>
<th>GOps/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-SLI</td>
<td>11,3436288</td>
<td>70,5244262</td>
<td>7,05244263</td>
</tr>
<tr>
<td>SLI</td>
<td>11,1039175</td>
<td>72,0466879</td>
<td>7,20466884</td>
</tr>
</tbody>
</table>

Table 1: Comparison of non-SLI and SLI systems

3. Opportunities for Dependability

In this Section, we will discuss the opportunities for dependability offered by modern graphics cards. Note, that our terminology is based on [8]. 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. We will first have a look at the section means from the dependability tree (from [8]) in Fig. 6. Then we will discuss the means fault prevention, fault-tolerance, fault removal and fault forecasting in the following subsections.

![Fig. 6: A section from the dependability tree](image)

We distinguish different levels on which different dependability measures can be applied. Therefore, we depict the notational conventions in Table 2 and note the level, where zero (0) means the top level.
Table 2: Notational Conventions

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

In the next subsections, we will discuss the opportunities for dependability according to Fig. 6.

3.1. Fault Prevention

We shortly 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, 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 forecasting of faults can also be seen as essential part of fault prevention.

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 the integer capability of a graphics card. Functional redundancy can be easily achieved by either computing a calculation on the CPU and the GPU (this also automatically involves diversity in software) or by programming a set of functions again specially for the GPU. When voting between the results, we can use the inherent voting capability supported by CUDA.

\(^{1}\) A thread on a graphics card is a lightweight construct. A GPU usually needs thousands of threads to work effectively.
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. There is also the possibility to combine integrated mainboard GPUs and graphics cards to enhance performance. One should be aware of the fact that multiple (PCIe) graphics cards of different vendors could be installed simultaneously. 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, 2 GHz memory. 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.

3.2.2. Temporal Redundancy

Temporal redundancy is immanent for a multithreaded system, thus also for graphics cards comprising hundreds or thousands of threads. We can imagine a temporal redundant computation of a kernel, a grid or a thread block, since these units are independent. The only point where structural redundant 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. The synchronization between host and device can also be problematic.

3.2.3. Watchdogs

A GPU can be periodically triggered by an external timer to monitor activities. The timer routine is able to directly access the memory (mem) of the graphics card. The external timer is needed, because GPUs do not possess such a capability. The activities are e.g. CPU or fixed disk functionalities. Any activity and the current time is written in the mem. On a write of the current time, the last time will be copied to a different location within the mem. 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 the mem) a wakeup signal can be issued. Fig. 7 shows the algorithm.
3.3. Fault Removal

Fault removal is a hard thing to implement by using GPUs, because the faulty unit must be located and a prior or 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 can imagine something like a RAIG5 configuration (Redundant Array of Independent Graphics cards, according to a RAID5).

Fault removal within a CPU from a GPU is possible but far more difficult. Here 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 usually cannot map the main memory of the host to the device memory, which is smaller than that of the hosts’ memory, we must either do every memory write simultaneously on the card, significantly decreasing performance or a fault removal for a single application running on the CPU such as a daemon. For CPU states, this is no problem, because the amount of data to transfer is small.

3.4. Fault Forecasting

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 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. 8 briefly sketches the method without going into details.

```
Startup: History location h=0;

CPU:
  On_MCE: Write MCE-Flag to GPU memory, location h
  h=h+1 % N
  Call prediction on GPU

GPU:
  On_Call: Do prediction using h
```

Fig. 8: Basic (abstract) prediction of faults

Note, that we assume the outsourcing of CPU application parts to the GPU and not vice-versa.

4. Summary and Outlook

This work presented a first step and innovative approach to use GPUs for dependability. It has been shown how the existing parallelism of GPUs can be exploited for dependability. Future work will include a short-term reliability evaluation and the implementation of different means for structural and temporal fault-tolerance.

References


