“Computing on GPUs”

Prof. Dr. Uli Göhner

DYNAmore GmbH

Stuttgart, Germany

Summary:

The increasing power of GPUs has led to the intent to transfer computing load from CPUs to GPUs. A first example has been the porting of computing intensive algorithms like e.g. ray-tracing algorithms from CPU to GPU. Through the Compute Unified Device Architecture (CUDA [4]) GPUs can also be used to increase computing speed for High Performance Computing applications. In this paper different parallelization strategies for different processor architectures are presented. They are compared and first experiences using GPUs for a collection of numerical applications are given.

Keywords:

Crash Simulation, GPU, CUDA.

1 GPU for compute applications

GPUs nowadays have evolved to the point, where not only graphical applications, but also computing intensive applications can be implemented on them. The performance of the GPU can in some cases be significantly higher than on the CPU. Future computing architectures for High-performance computing applications can therefore be hybrid systems with GPUs working together with multi-core CPUs. To become this vision reality, both an appropriate programming model and development tools have to be provided. In this paper different approaches have been evaluated and pros and cons of the different systems provided so far are discussed. As the typical rendering algorithms are more and more replaced by ray-tracing algorithms, also for graphical applications it becomes important to have general computing power inside the GPU. A number of ray-tracing algorithms have been developed to run on large compute clusters, similar to typical hardware platforms used in HPC-applications [9]. The big disadvantage for the usage of the cluster platform for ray-tracing applications was the bottleneck resulting from CPU main memory latency. Therefore ray-tracing algorithms have been developed for the GPU and have good success for real-time realistic rendering applications [6]

2 GPU architecture

For optimal performance of computing applications on the GPU, it is necessary to have a quick look on the hardware architecture of a modern GPU. We will shortly look on the G80 processor of NVIDIA, which is the basis of NVIDIA's Geforce 8800 unit. The G80 consists of 8 groups of 16 stream processors (marked as SP), which totals in 128 parallel processors. Each stream processor is a floating-point processor capable of operating on vertices, pixels or other data. The L1-Cache is shared between the 16 stream processors. Below the 8 groups of stream processors there is a crossbar switch connecting the stream processors to the frame buffer (FB) graphics memory.

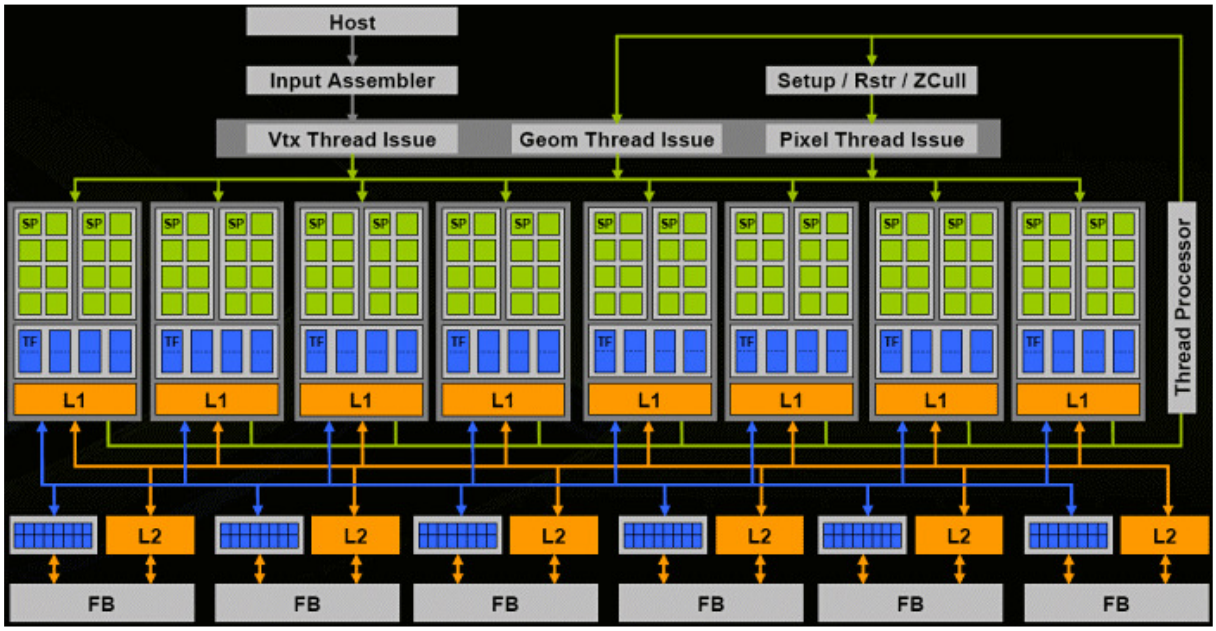

Figure 1: G80 architecture [4]

Similarily as for other GPUs, also the NVIDIA G80 processor has been optimized to get best performance for the typical "graphics pipeline" process as outlined in Figure 2 (Open GL Pipeline).
A large collection of triangles, vertices or pixels is processed in parallel similar to the way, vector processor machines have been working. The underlying concept is named SIMD (Single Instruction, Multiple Data), due to Flynn's- classification of multi processor architecture [2]. Some CPUs also offer a way of exploiting data level parallelism (ILP) using SIMD extensions. One example is Intel's SSE or IBM/Motorola's Altivec. Instruction parallelism means, that the same instructions are performed in parallel on different data items. For the chip designer, SIMD parallelism is a cheap way of increasing a processor's performance.

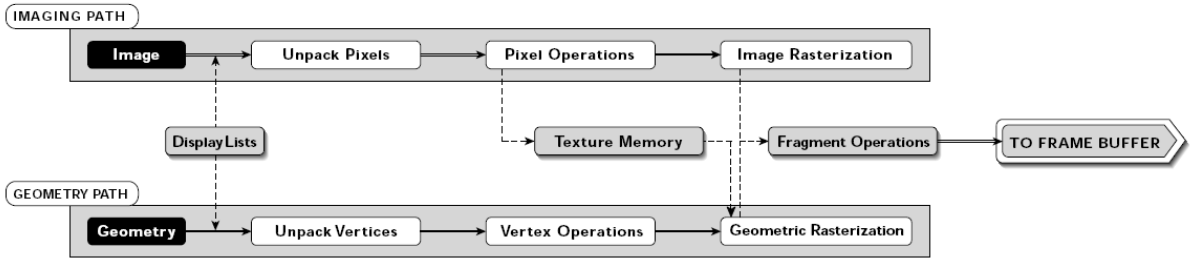

Figure 2: The OpenGL Visualization Programming Pipeline [7]

3 Programming pattern for GPUs

The hardware architecture outlined in the previous section allows both programming concepts for SIMD and MIMD parallel approach. In graphical applications the SIMD approach led to the concept of "stream programming", which means that the same program instructions are processed on large arrays of vertices, triangles, pixels in parallel. One instruction sequencer can work on an array of 128 processors with different in- and output data. At the same time a MIMD approach leads to multiple threads working in parallel and sharing the same memory.

4 Parallel computer architecture for High Performance Computing

4.1 Vector machines

Both SIMD and MIMD hardware architecture has been used for high performance computing in the past. Vector processors like the Cray XMP [1] were very popular platforms for high performance computing. Also the LS-DYNA performance has been optimized and is still being tuned for vector machines like e.g. NEC SX6 [8]. A large part of the code optimization for vector machines is being done by the compiler.

4.2 Symmetric Multi processing

SMP-parallel hardware architectures use a shared memory architecture as outlined in Figure 3:

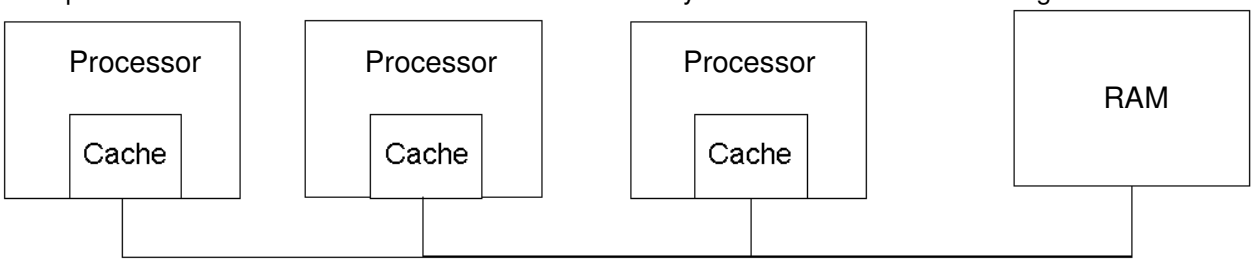

Figure 3 SMP hardware architecture

The following concepts for the usage of SMP-parallel hardware are available:

- OpenMP
- Pthreads

OpenMP works through compiler directives. Most compilers are starting and administrating different threads through the given compiler directives. For instance can different sections of a do-loop be performed in parallel. A "parallel for section" compiler directive will cause the compiler to start different threads on the multiple processors. All threads stay within the total memory space given to the parallel process. More general parallel programming principals can be realized by using pthreads. Pthreads can be started and synchronized by appropriate mechanisms provided by the pthread library. The programmer must keep an eye on consistent memory behavior and prevent the different threads from race conditions resulting in non deterministic behavior of the program. Mutual exclusive access to the shared variables must be realized by the programmer himself.

For LS-DYNA a parallel version based on OpenMP is available. As the Mutex controls prevent perfect parallelism, the performance of the OpenMP-parallel code is limited. Therefore in practice the SMP-version of LS-DYNA is only used up to 8 processors in maximum.

4.3 Distributed Memory parallel processing

A DMP parallel hardware architecture is outlined in the following Figure 4.

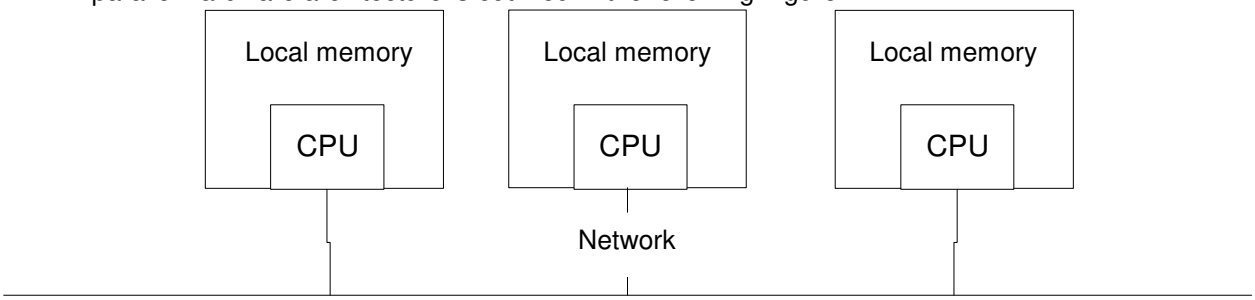

Figure 4: DMP architecture

Each processor only has local memory. If data are to be exchanged between the different CPUs, a concept of messages is being used. There are two main programming models for distributed memory parallel architecture:

- MPI (Message Passing Interface)
- PVM (Parallel Virtual Machine)

Both concepts realize different tools for the data exchange between processors. For instance the value of an input variable can be send to all processors, each of the processors can perform different computations based on this value and the different values can then be collected and processed to the resulting output value.

For LS-DYNA since the beginning of the early 1990s a MPI-version has been developed and is of widely use nowadays. Due to a domain decomposition technique (see Figure 5) the necessary communication is reduced to a minimum. This results in fairly good parallelization degrees and therefore good speedups can be achieved up to large processor numbers. Of course the speedup numbers, which could be achieved in reality, are much dependent on the bandwidth and latency of the underlying network connection.

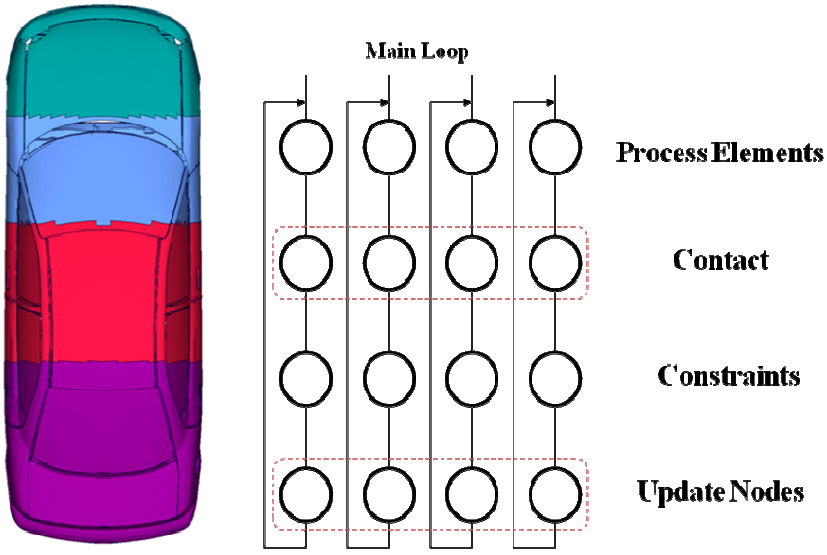

Figure 5: Domain Decomposition

5 Development tools for GPUs

A variety of tools for GPU programming is available for general computing applications.

5.1 OpenGL

By directly using the programming language OpenGL for graphical operations, developed by SGI [7], the GPU could already been used for matrix-vector operations. Arrays will be put into 2d-textures and the loop body will be activated through the rendering process. The result can be read from the frame buffer memory.

5.2 Stream programming

A stream is an array. The elements of this array can be operated on in parallel, similar to the SIMD approach outlined above. A fast communication from memory to the stream is essential. In the stream programming, model data are gathered from memory into the stream, operated on in the stream, and then scattered from the stream back into memory. As data are accessed in large chunks, data transport is optimized.

5.3 Peakstream

PeakStream's Platform consists of a combination of a virtual machine and a library. The virtual machine includes a scheduler, that schedules operations on both the CPU and the GPU. Also other special devices like the Cell processor can be addressed. Through this general concept an improved performance can generally be obtained on heterogeneous systems consisting of different special devices.

5.4 Brook Language

Brook is an extension of standard ANSI-C. It is based on the stream programming model, providing the following benefits:

- SIMD parallelism: Allows the programmer to specify, how to perform the same operations in parallel on different data.
- Arithmetic Intensity: Encourages programmers to specify operations on data which minimize global communication and maximize localized computation.

5.5 RapidMind

RapidMind was started at the University of Waterloo commercialized by the company RapidMind, which markets the RapidMind language and compilers. This language is embedded in C++ programs and allows to program GPUs directly without any shader language. The Rapid Mind compiler controls the communication between CPU and GPU

5.6 OpenCL

OpenCL (Open Computing Language) is an open standard for parallel programming of heterogeneous systems. With OpenCL a uniform programming environment is provided for software developers to write efficient, portable code for high-performance computing applications. The addressed platforms could be large-scale server systems, desktop computer systems and handheld devices using a mixture of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors.

5.7 Cuda

CUDA (=Compute Unified Device Architecture) was developed by NVIDIA in 2006 [4]. It is an extension to standard ANSI-C programming language similar to the BROOKS language described in chapter 5.4. CUDA supports the stream concept and corresponding operations. As shown in Figure 6 the NVIDIA graphics hardware is directly accessed by CUDA, so that a wide range of computing possibilities like e.g. branching, looping or pointer support can be provided. Additionally standard libraries for linear algebra (BLAS) and digital signal processing (FFT) are contained in the CUDA framework.

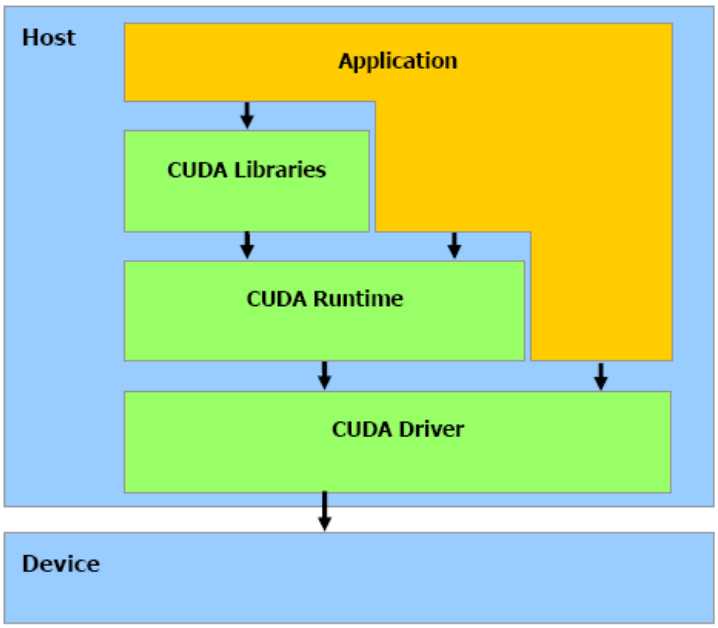

Figure 6 The CUDA development platform [4]

6 Benchmark results

A small project team has been formed at the University of Applied Sciences Kempten. Different hardware platforms are being investigated and the performance of a collection of benchmark examples is being tested using the CUDA development platform. The results of the benchmark studies will be given in the presentation at the time of the conference. In 2007 R. Lucas, G. Wagenbreth and D. Davis report initial results for an implicit calculation on the GPU based on an LS-DYNA benchmark example [3]. This work is being continued and the results will be published.

7 Literature

[1] Edirisooriya, S. Edirisooriya, G. "Enhancing vector access performance in CRAY X-MP memory system", Compcon, 1993.
[2] Flynn, M.: "Very High Speed Computing Systems", Proceedings of the IEEE, vol. 54, 1966.
[3] Lucas, R., Wagenbreth, G., Davis, D.: „Implementing a GPU-Enhanced Cluster for Large-Scale Simulations"
[4] "The NVIDIA Cuda Zone", www.nvidia.de, 2009.
[5] Pattani, P.G., Ray, H., Minga, A.K.: "Performance Evaluation of LS-DYNA3D on NEC SX-4 Series", 3rd International LS-DYNA conference, 1995.
[6] Purcell, J.T..: "Ray Tracing on a stream processor", phD-thesis, 2004.
[7] SGI OpenGL technical information – www.sgi.com, 2009
[8] Skinner, G., Lam, D., Masataka, K., Shimamoto, H.:"SPH performance enhancement in LS-DYNA", 8th International LS-DYNA Conference, 2006.
[9] Wald, I.: "Realtime Ray Tracing and Interactive Global Illumination", phD-thesis, 2004.