Seminar Paper

General Purpose Computation on Graphics Processing Units (GPGPU) using CUDA

within the seminar

Parallel Programming and Parallel Algorithms

(Winter Term 2009/2010)

Alexander Zibula

Examiner:	Prof. Dr. Herbert Kuchen
Advisor:	Philipp Ciechanowicz
	Department of Information Systems
	Practical Computer Science
Date of submission:	20.12.2009

Table of Contents

1 Introduction 1

2 GPGPU Basics 2

2.1 Graphics Pipeline 2

2.2 Graphics APIs 3

2.3 Graphics Shaders 3

2.4 Graphics Hardware 5

3 GPGPU and GPU Computing 5

3.1 Stream Processing 5

3.2 Traditional GPGPU 6

3.3 GPU Computing with the Compute Unified Device Architecture 7

3.3.1 Introduction to CUDA 7

3.3.2 CUDA Programming Model 8

3.3.3 CUDA GPU Hardware Architectures 10

3.3.4 A Simple C for CUDA Program 10

3.3.5 Compiling CUDA Programs 11

3.4 Evaluating and Comparing CUDA 12

3.4.1 Evaluating CUDA 12

3.4.2 CUDA and CPU Parallelization 14

3.4.3 Nvidia CUDA versus ATI Stream 14

4 Algorithms and Applications for GPU Computing 15

4.1 Matrix-Vector Multiplication on CUDA 15

4.2 Matrix-Matrix Multiplication on CUDA 16

4.3 A Little Experiment 16

4.4 Application Scenarios 18

5 Summary and Outlook 19

A Specification and Result Tables 21

B Additional Figures 22

C Matrix Multiplication Listing 24

Bibliography 27

List of Abbreviations

ALU	Arithmetical Logical Unit
BLAS	Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
ECC	Error Correcting Code
FFT	Fast Fourier Transform
FP	Floating Point
GPGPU	General Purpose Computation on Graphics Processing Units
GPU	Graphics Processing Unit
HPC	High-Performance Computing
PCIe	Peripheral Components Interconnect Express
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SP	Streaming Processor
SSE	Streaming SIMD Extensions

1 Introduction

Graphics processing units (GPUs) are special processors which traditionally were used to accelerate computer graphics by offloading work from the CPU. Today, GPUs are highly parallel many-core processors which enable general-purpose computation on graphics processing units (GPGPU). GPGPU has already been an issue since 2002 but a huge interest did not evolve until Nvidia released the CUDA platform in 2007. Developers and researchers started to use CUDA for parallel programming. The current high visibility in science and practice, especially in parallel, scientific and high-performance computing (HPC), is one reason for this paper. Further motivation arises through interest in computer graphics and parallel computing.

Nvidia's architecture CUDA was chosen for this paper because it was the dominating platform for GPGPU at the time of writing. Nvidia was the first vendor that diffused a comprehensive architecture combining huge programmability, performance, and ease of use. However, CUDA is challenged by AMD's alternative ATI Stream [AMD09a] as well as two standardization approaches, OpenCL [Kh09b] and DirectX11 DirectCompute [MS09d]. Nvidia is also facing competition in other markets. Intel and AMD both use a platform strategy, combining x86 CPUs, graphics, and chipsets and trying to put Nvidia out of the chipset market [Wi09]. While Nvidia confirms that it has no intention of constructing x86 processors [Cr09], GPGPU, HPC, and parallel computing have become a major strategic pathway. Effects of this are huge research, marketing, and collaboration efforts, e.g. lectures, tutorials, student scholarships, and partnerships with professors, universities, and software development companies, which resulted in a large amount of scientific publications and parallel applications [NVI09m].

The purpose of this paper is to introduce the reader to the field of GPGPU and the practice of CUDA. Thus, the remainder of this paper is structured as follows: Chapter 2 briefly addresses fundamentals of computer graphics and graphics hardware, establishing a basis for understanding GPGPU. Chapter 3 will elaborate on traditional GPGPU and the newer GPU Computing approach CUDA. Chapter 4 compares implementations and performances of matrix multiplication algorithms and addresses application scenarios. Finally, Chapter 5 will give concluding remarks and an outlook to future work.

2 GPGPU Basics

The following chapter will introduce basics of computer graphics and graphics hardware which form the background of GPGPU. During this, some basic terms for computer graphics objects will be needed: *Geometric primitives* are simple atomic geometric objects like points, lines, triangles, or other polygons. The corner points of these objects are called *vertices*. Another basic object is a *fragment* which is the basis for a pixel. In addition to the color value, the fragment also contains other information that is needed before the pixel is drawn, e.g. the position, the depth, or the alpha value (for transparency) [Ha06, Ih09 pp. 9-10].

2.1 Graphics Pipeline

A graphics pipeline (also called rendering pipeline) is a model that describes different steps performed to render a scene. The pipeline concept can be compared to the CPU instruction pipeline: The individual steps are done in parallel, but are blocked until the latest step is finished. One simple model of a (fixed-function) graphics pipeline is depicted in Fig. 1.

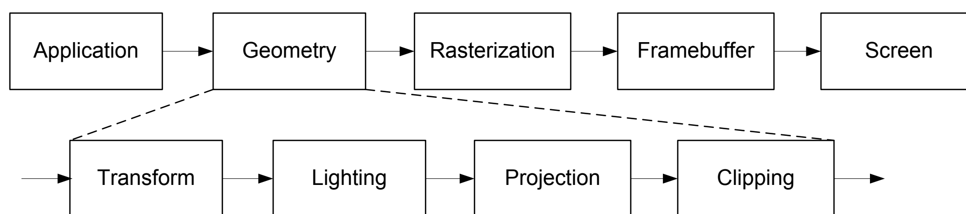

Fig. 1: Simple graphics pipeline [BB03 pp. 71-73, XP07 p. 200].

The *application* changes the scene e.g. reacting on user input. The components of the new scene are forwarded to (*model & camera) transformation*. In this step, local object coordinates are transformed into a global coordinate system. The camera is positioned in the scene and the coordinate system is adjusted to it. During *lighting*, color values are calculated for all vertices depending on the position of lights and the properties of the corresponding triangles. In the *projection* step, the 3D scene is mapped into a 2D image space. During *clipping,* unnecessary primitives are eliminated. In the *rasterization* step hidden surfaces are removed (using the z-buffer algorithm) and the scene is transformed into a bitmap by calculating color values for every pixel [BB03 ch. 2, XP07].

2.2 Graphics APIs

Graphics APIs provide programmers a high level of abstraction and simplify the software development process by hiding complexity and capabilities of graphics hardware and device drivers. The two most important graphics APIs will be briefly introduced in the following.

Direct3D is an API for drawing 3D graphics and the most prominent component of the comprehensive DirectX API collection for multimedia applications on Microsoft platforms (Windows and Xbox). An advantage for programmers using DirectX is the huge market penetration, which enables Microsoft to define minimum hardware specifications for graphics components in collaboration with the graphics vendors. Disadvantages like the fact that it is proprietary, low backward compatibility, and short release cycles can be criticized [BB03 p. 4]. However, the last two arguments also provide the basis for innovations: Until Direct3D 10, the most interesting development for GPGPU was the introduction of different shader models (cf. Section 2.3). The current version 11 has been released in October 2009 and features hardware support for tessellation, which increases the amount of polygons through subdivision of polygons at runtime within the pipeline of the GPU, increased multi-threading support (for multi-core CPUs), and DirectCompute, Microsoft's new approach to GPGPU [Be09a].

OpenGL [SWN06, Kh09b] is a cross-language, cross-platform API for 2D and 3D computer graphics, and the main alternative to Direct3D. OpenGL is coordinated by the Khronos Group [KH09a], an industry consortium of more than 100 members including AMD, Intel, and Nvidia. The latest version 3.2 was released in August 2009. OpenGL as well as DirectX nowadays heavily rely on graphics shaders which will be presented in the following Section 2.3.

2.3 Graphics Shaders

In computer graphics, *shaders* are small programs that run on the GPU or processors which execute them. The beginning of shaders marked the transition from a fixed function pipeline over a configurable to a programmable one [Ih09 p. 7-9]. Shaders made the rendering process flexible and enabled new graphical effects. The first shaders were introduced in OpenGL 1.5 and Direct3D 8. Later versions of the APIs revised the shader specifications and increased flexibility, e.g. by loosening constraints like the maximal amount of instruc-

tions per shader. The APIs currently specify three types of shaders used for graphics: vertex, geometry, and pixel shaders[1]. *Vertex shaders* can change coordinates or vertices while *geometry shaders* are able to generate or duplicate new geometric primitives from existing primitives in the pipeline (e.g. for instancing trees or characters). Both shaders can be mapped to the geometry step of the earlier presented fixed function pipeline (cf. Fig. 1). *Pixel shaders* (in OpenGL called *fragment shaders*) are performed after the rasterization step. They operate on single pixels/fragments and perform e.g. color or shadow changes [NVI06].

A major innovation in graphics shaders was the introduction of *unified shaders* which consist of two aspects: The *unified shader model* harmonizes the instruction sets between different shader types and was introduced in DirectX 10. This is not to be confused with the *unified shader architecture*, which means that all shaders have the same hardware layout and can be used dynamically as vertex, geometry or pixel shaders. Nvidia introduced the unified shader architecture with the GeForce 8 series. Fig. 2 depicts the difference between the architectures. On the right the same shader core is reused iteratively for different shader tasks. The benefit of this approach is the possibility to dynamically react on different pixel/geometry complexities of different scenes to increase performance and efficiency (cf. Fig. 8).

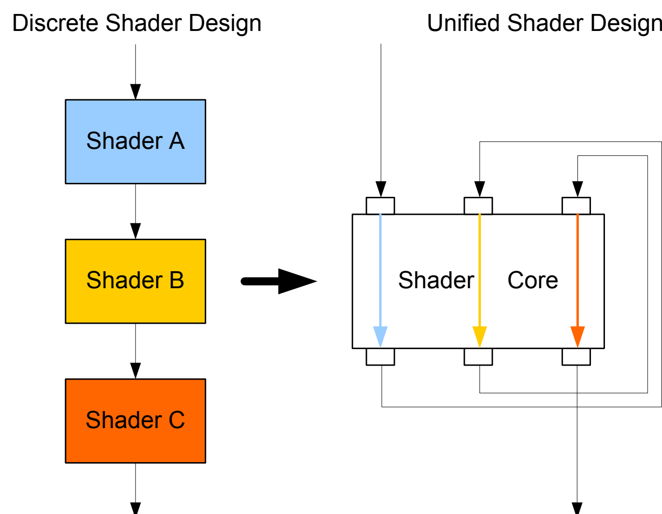

Fig. 2: Comparison of a discrete and a unified shader design [NVI06 p. 21].

Shaders are programmed in different languages: While low level programming is done with assembler, high-level approaches are the High Level Shading Language (HLSL)

[1] Not counting the DirectX11 Compute Shaders, which are not used for graphics purposes.

[MS09b] targeted to Direct3D programming and developed by Microsoft, C for Graphics (Cg) [FK06] developed by Nvidia in collaboration with Microsoft and targeted to both Direct3D and OpenGL programming, as well as the OpenGL Shading Language (GLSL) for OpenGL programming [Ih09 p. 23-29].

2.4 Graphics Hardware

In modern PCs, GPUs are either present on a dedicated graphics card or on the motherboard as integrated graphics solution. The latter usually have little or no own graphics memory, compete with the CPU in utilizing main memory, and reside at the lower price and performance spectrum. However, the computing power is generally sufficient for simple 2D and 3D graphics tasks. Problems arise e.g. with complex 3D video games in high resolutions, CAD software, or GPGPU. High-performance GPUs are typically only available as dedicated graphics cards. The cards are connected to the system via an expansion slot, currently PCI Express (PCIe) v2.0 which uses point-to-point serial links. The serial links are composed of one to 32 *lanes*, each lane carrying 500 MB/s. Most contemporary cards are connected via 16 lanes which allows for a data transfer speed of 8 GB/s (full duplex) [PCI09]. It will later become clear that this is a major bottleneck for GPGPU applications.

3 GPGPU and GPU Computing

3.1 Stream Processing

The basic programming model of traditional GPGPU is *stream processing*, which is closely related to SIMD[2]. A uniform set of data which can be operated in parallel is called a *stream*. The stream is processed by a series of instructions, called *kernel* [Ow06]. Stream processing is a very simple and restricted form of parallel processing (also called embarrassingly parallel) that abstains from explicit synchronization and communication management. It is especially designed for algorithms that observe high arithmetic intensity[3], data parallelism, and data locality. It is not reasonable in case of data dependencies, recursion or random memory accesses [Ow06, BFH04]. Computer graphics processing is very

[2] Single Instruction Multiple Data, in the Flynn’s taxonomy of computer architectures.

[3] The ratio between calculation and memory operations [Ha06 ch. 1.1].

suitable for this, where vertices, fragments and pixel can be processed independently of each other, with clearly defined directions and address spaces for memory accesses [Ow06]. The stream processing programming model allows for more throughput oriented processor architectures. For example, without data dependencies caches can be reduced in size and the transistors can be used for ALUs instead. Fig. 3 visualizes a simple model of a modern CPU and a GPU. The CPU uses a high proportion of its transistors for controls and caches while the GPU uses them for computation (ALUs).

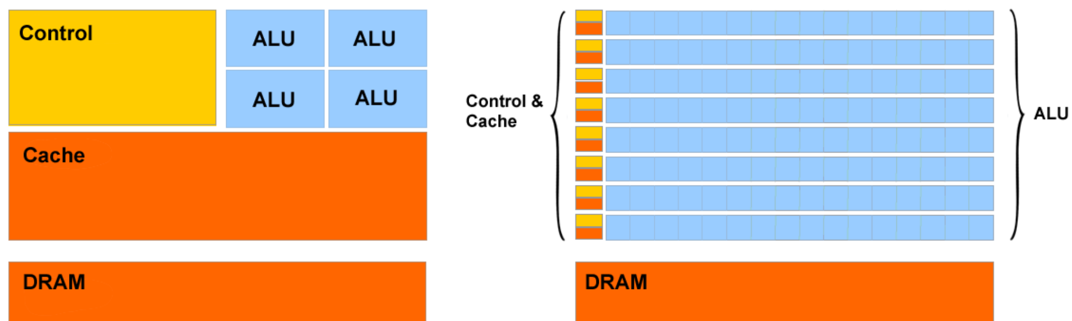

Fig. 3: Simple comparison of a CPU and a GPU [NVI09d p. 3].

3.2 Traditional GPGPU

Traditional GPGPU was already possible in 2002. The requirements for this were the increasing performance and programmability, the latter realized through graphics shaders and the introduction of more complex and precise data types. Early GPUs operated with eight bit integers (pixels with 256 colors). Floating point data types with different grades of precisions were added later [Ha06 ch. 2]. The first GPGPU programs directly used the graphics APIs and hence, were written in HLSL, GLSL, or Cg. The programs had to utilize the computational units on the graphics card in a restrictive and differentiated way. The texture unit was used as read only memory, the framebuffer as write only memory. The vertex and pixel shaders were used to execute the kernels. The rasterizer was used for address calculation [Ha06].

The traditional GPGPU approach had several drawbacks: The programming model was improvised and clumsy. The code became complex and the programmer needed to have a very deep understanding of the graphics APIs and the specific GPU architecture. Precision was limited (double precision was far away and rounding standards were vendor-specific) [NVI09f p. 3]. These drawbacks have been targeted by Nvidia CUDA, which will be discussed in detail in the following section.

3.3 GPU Computing with the Compute Unified Device Architecture

3.3.1 Introduction to CUDA

CUDA is a comprehensive software and hardware architecture for GPGPU that was developed and released by Nvidia in 2007. It is Nvidia's move into GPGPU and HPC, combining huge programmability, performance, and ease of use. A major design goal of CUDA is to support heterogeneous computations in a sense that applications are serial parts of an application are executed on the CPU and parallel parts on the GPU [NVI09c p. 1]. An overview of CUDA is given in Fig. 4. The core components reside at the bottom: the CUDA-enabled GPU (1), CUDA support in the OS kernel (2) (e.g. for hardware initialization and configuration) and the CUDA driver[4] (3) [NVI09a p.1].

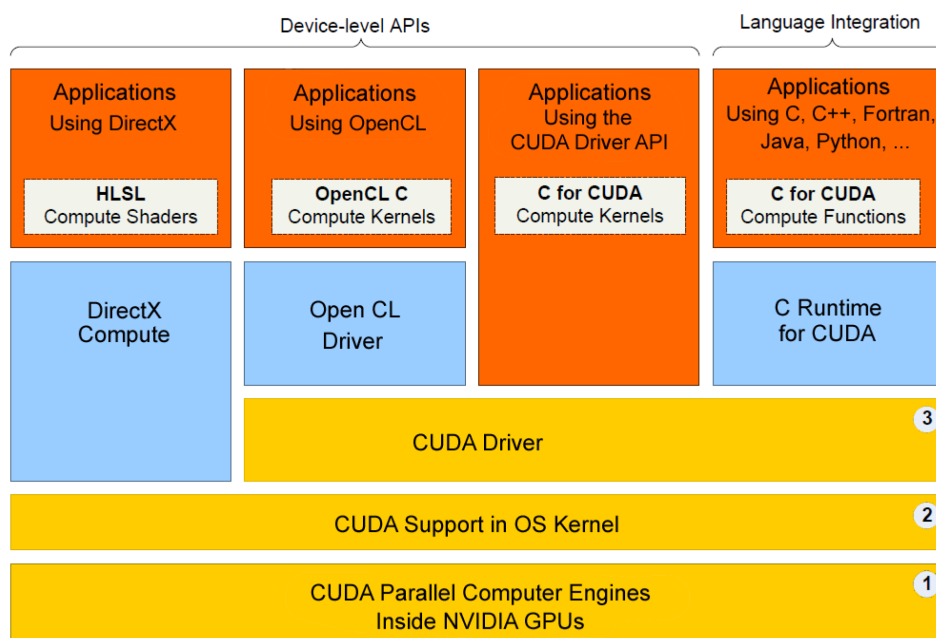

Fig. 4: CUDA overview [NVI09a p. 1].

CUDA currently supports two different kinds of programming interfaces. Via the device level APIs (cf. left part of Fig. 4) it is possible to use the new GPGPU standard DirectX Compute by writing compute shaders in HLSL. The second standard is OpenCL, governed by the Khronos Group (as is OpenGL). OpenCL kernels are written in OpenCL C. Both approaches are independent of the GPU hardware and therefore also applicable to AMD GPUs. The third device-level approach is low-level CUDA programming which directly uses the driver. It offers huge control but is rather complex since it handles binaries or as-

[4] Integrated into the general Nvidia GPU driver.

sembly code. An alternative to the device level API is the language integration programming interface (cf. right column of Fig. 4). Using the C runtime for CUDA is a high-level approach that requires less code and is easier in programming and debugging. Hereby, it is also possible to use bindings for other high-level languages e.g. Fortran, Java, Python, or .NET languages [NVI09a pp. 1-3].The remainder of this paper will focus on this last approach.

On top of the CUDA runtime, Nvidia offers functional libraries, like *CUBLAS*[5] and *CUFFT*[6] which provide is simple standard interface to often used routines. Further components of the software development environment are the compiler *nvcc*, the debugger *cudagdb*, the profiler *cudaprof*, documentation and SDK code samples [NVI09a pp. 2]. The compilation process will be covered in more detail in Section 3.3.5.

3.3.2 CUDA Programming Model

The CUDA programming model encourages dividing problems in two steps: At first into coarse independent sub-problems (→ grids) and afterwards into finer sub-tasks that can be performed cooperatively (→ thread blocks). The programmer writes a serial C for CUDA program which invokes parallel *kernels* (functions written in C). The kernel is usually executed in thousands of threads, which the programmer organizes in a hierarchy (cf. Fig. 5). Single threads are grouped into a *thread block* (*block* for short). Within a block, threads cooperate via barrier synchronization and access to a shared memory which is only visible to the block. Each thread in a block has a unique *thread ID* `threadIdx`. Blocks are grouped into a *grid*. Blocks within one grid are independent. Each block in a grid has a unique *block ID* `blockIdx`. Grids can be executed either independent or dependent. Assuming sufficient hardware capacity, independent grids can be executed in parallel. Dependent grids can only be executed sequentially. An implicit inter-kernel barrier ensures that all blocks of a previous grid have terminated before any block of the new grid is started [NBG08, NVI09d, Fa08].

[5] For BLAS (Basic linear algebra subprograms), a de facto API standard for linear algebra routines.

[6] For FFT (Fast Fourier transform), an efficient algorithm for computing the discrete Fourier transform.

The difference between invoking a kernel and calling a normal C function is the specification of the amount of threads per block and blocks per grid. This is done via three pairs of angle brackets:

```
kernel<<<dimGrid, dimBlock>>>(parameter_list);
```

Blocks and grids can also be addressed in higher dimensions. Therefore, `dimGrid` and `dimBlock` are three dimensional vectors of the type `dim3`.

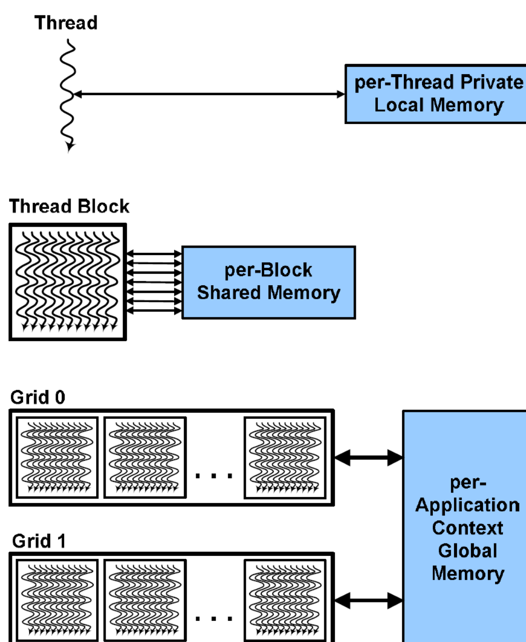

Fig. 5: CUDA thread and memory hierarchy [NVI09f p. 6].

Closely related to the thread hierarchy is the memory hierarchy (cf. Fig. 5). Each thread uses private *local* memory for registers, stacks frames, and register spilling. Each block uses a *shared* memory which is supposed to be very fast. A typical usage scenario would be that the kernel code initializes variables in the shared memory, the threads in the block calculate, using these variables, and the results are copied to the *global* memory which is visible to the whole application. The global memory enables communication between blocks of independent grids. On a typical Nvidia GPU architecture, the shared memory is implemented by a low-latency on-chip RAM close to the processors while the global memory is implemented by the DRAM (cf. Section 3.3.3) [NBG08].

Due to specifics of the hardware architecture and for performance benefits, the programming model has the following restrictions: Firstly, it is not possible to create threads or blocks from within a parallel kernel, which simplifies the thread scheduler. Secondly, blocks within a grid are independent and cannot communicate. This enhances scalability,

because blocks can be distributed on an arbitrary amount of processors[7]. Thirdly, recursion is not supported, as it violates the stream processing programming model [NBG08, p. 47-48].

3.3.3 CUDA GPU Hardware Architectures

Nvidia's first GPU supporting CUDA was the *G80* introduced in November 2006 on the GeForce 8800. The G80 has 16 streaming multiprocessors (SMs), each of them having 16 KB shared memory and eight streaming processors (SPs) resulting in a total of 128 SPs (cf. Tab. 1, p. 21) [NVI06]. The *GT200* architecture succeeded the G80 in June 2008. It increased the amount of SMs to 30, resulting in 240 SPs. Furthermore, double precision FP capability was added. Nvidia's future CUDA hardware architecture is called *Fermi*[8]. It features 16 SMs, each of them having 32 SPs (now also called CUDA cores) and 64 KB shared memory which is configurable as larger shared memory or larger L1 cache (48/16 KB or 16/48 KB). The total amount of SPs is 512 and the whole GPU shares a L2 cache of 768 KB. Further improvements of Fermi are eight times faster double precision performance, IEEE 754-2008 FP precision, and error correcting code (ECC) memory, especially required for reliability requirements of scientific computing [NVI09f].

3.3.4 A Simple C for CUDA Program

C for CUDA is the main language that is used for programming in CUDA. It differs from standard ANSI C in a few special language extension and API calls. The differences will be introduced by means of a simple example, the SAXPY operation, which combines a scalar multiplication with a vector addition in single precision and is part of the BLAS API. Listing 1 denotes a simple conventional implementation in C.

```
void saxpy_serial (int n, float alpha, float *x, float *y) {
      for (int i = 0; i < n; ++i) {
            y[i] = alpha * x[i] + y[i];
      }
}
// Call the function (Invoke serial SAXPY kernel)
saxpy_serial(n, 2.0, x, y);
```

Listing 1: SAXPY y = ax + y in a serial loop [NBG08 p. 45].

[7] However, thread blocks can coordinate via of atomic functions on the global memory (available since compute capability 1.1).

[8] Fermi graphics cards are supposed to be available in the first quarter of 2010.

Listing 2 shows the corresponding implementation in C for CUDA. The `__global__` qualifier indicates that the following function is a CUDA kernel that is compiled for the GPU and globally visible. Instead of a loop, each pair of vector elements is processed in a separate thread. The implicitly defined variables `blockIdx`, `blockDim` and `threadIdx` are used to assign the right vector elements to the corresponding thread. Note that the computation line `y[i] = alpha * x[i] + y[i]` is identical to Listing 1. The variable `nblocks` is used to calculate how many thread blocks are needed, depending on `n` and the number of threads per block (here 256). With this specification the kernel can be invocated. Further C for CUDA language constructs will be used in Chapter 4.

```
// SAXPY kernel definition
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      if (i < n) {
            y[i] = alpha * x[i] + y[i];
      }
}
// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel <<< nblocks, 256 >>> (n, 2.0, x, y);
```

Listing 2: SAXPY y = ax + y in a parallel CUDA kernel [NGB08 p. 45].

3.3.5 Compiling CUDA Programs

Compiling a CUDA program involves more steps than compiling a normal C program (cf. Fig. 6). This is due to Nvidia's hardware abstraction and the fact that CPU and GPU architectures are targeted [NVI09c]. As seen in the example before, the source code for the CPU and the GPU can be mixed into one file. The C/C++ preprocessor from Edison Design Group (EDG) is used for splitting into separate sets of .cpp files. The CPU code can be compiled with an arbitrary C compiler. The GPU code is compiled with Open64, an open-source C/C++ compiler. Nvidia's version of Open64 creates .ptx files (Parallel Thread eXecution). These GPU-independent files are later assembled into GPU-specific executables. [Ha08 p. 6, NVI09k].

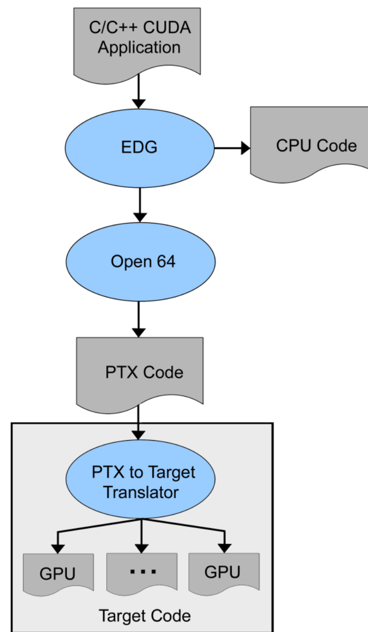

Fig. 6: CUDA compilation process [Ha08 p. 6].

3.4 Evaluating and Comparing CUDA

3.4.1 Evaluating CUDA

After getting to know CUDA it will now be evaluated using the criteria: diffusion, investment, market, innovation, development tools, programming model, and performance in applications:

- *Diffusion:* Another advantage is the diffusion of CUDA-enabled hardware: Nvidia sold already more than 100 million CUDA-enabled GPUs for desktops, notebooks, workstations and supercomputer clusters [NVI09a p. 1]. Knowledge of CUDA is spreading in the academic community, which is highly enthusiastic about many-core programming and GPU Computing.
- *Investment:* The CUDA software is free and the hardware is relatively cheap because of mass production and strong competition. However, CUDA causes highly specific costs: It requires special programming knowledge and written (C for CUDA) programs are only targeted to Nvidia GPUs.
- *Market:* The competition in the parallel programming market is strong. Alternatives to CUDA are ATI Stream, OpenCL, and DirectCompute. The latter two have the enormous advantage of hardware platform independence. However, OpenCL and DirectCompute also run on the CUDA platform. It is likely that Nvidia will support

CUDA for some years even if OpenCL and DirectCompute will be used more and more.

- *Innovation:* Nvidia has a lot of experience in parallel processing, because GPU's have been massively parallel long before CUDA. The GPU market is highly innovative and significant performance enhancement and feature additions through new architectures are expectable.
- *Development tools:* Nvidia has released tools to support CUDA software development, like the debugger and the profiler. The CUDA SDK v2.3 basically supports Visual Studio. While syntax highlighting is supported, a wizard for creating CUDA projects is missing, which make is necessary to copy configurations from existing projects. A comprehensive Microsoft Visual Studio integration (called *Nexus*) is scheduled for the upcoming CUDA version 3.
- *Programming model:* The programming model is highly scalable and relatively easy. The automatic thread management reduces the parallel programming complexity significantly. It allows for explicitly defining parallelism of selected parts of the code. CUDA shifts the focus from thinking about thread management to the task of decomposing data which is highly important in times of many-core processor architectures. The programming model can also be criticized: Parallelizing an existing sequential program may require significant code changes, like packaging the GPU code into the kernels, managing data-transfers as well as a lot of manual code optimizations. A possible improvement is *hi*CUDA [HA09], a directive based even higher-level data parallel language that is supposed to simplify CUDA programming. The main advantage is that it uses annotations to the original sequential C program, which eliminates the necessity of writing specific GPU functions (kernels). The developers used a source-to-source translator and compared the generated CUDA programs against five manually optimized benchmark programs and did not discover significant performance loss [HA09]. Another simplification approach is CUDA-lite which performs automatic memory optimizations. It also uses annotations and optimizes CUDA programs that only use global memory [ULB08].
- *Applications & Performance:* Very high speedups in comparison to CPU execution have been reported for a lot of different applications [CUDA ZONE, cf. Chapter 4]. However, the benefits are much more useful for scientific computing than for consumer applications. Applications need a lot of data parallelism and computational in-

tensity to benefit from the massively parallel architecture of GPUs. Otherwise, the memory transfer time between CPU and GPU post a significant overhead and offset the computational advantages. Furthermore, the impressive theoretical maximal performances are hardly achieved by any application (cf. Chapter 4).

3.4.2 CUDA and CPU Parallelization

CUDA can be compared to OpenMP [HL09] for CPU parallelization which is also targeted to shared memory architectures. It is as simple as CUDA, also using a compiler-based approach and adding directives to C code enabling explicit, selective parallelization [De09 pp. 36-37]. Like CUDA, details of thread management do not have to be handled by the programmer. This naturally reduces the degree of flexibility and manual optimization potential which would be possible e.g. with the Win32 Threading API [De09 pp. 38-40].

It has also been shown that the CUDA programming model can also be an efficient data-parallel programming model for multi-core CPUs: MCUDA [SSH08] is a framework that features a source-to-source translator and a runtime system. It translates kernels into explicit loops, tries to map the CUDA memory hierarchy to L1 and L2 caches of the CPU and to make use of the SIMD instruction set extensions like SSE [SSH08].

3.4.3 Nvidia CUDA versus ATI Stream

An alternative to CUDA is ATI Stream. The programming model of Stream is not too different. This is because both have the same roots: the Stanford University project Brook [BFH04]. However, ATI's solution was up to now less successful in terms of performance and adoption. While Nvidia has only presented Fermi on paper, AMD already released DirectX 11 compatible graphics cards and their recently released dual GPU flagship Radeon HD 5970 is currently the fastest available graphics card with a theoretical maximal performance of 4.64 TFlops (single precision). AMD is also adding support for OpenCL and DirectCompute [AMD09a].

4 Algorithms and Applications for GPU Computing

The following chapter will cover algorithms and applications that can be ported to the GPU and benefit from its parallelism. It will start with basic algorithms for matrix-vector and matrix-matrix multiplication, carry out a performance experiment and cover further application scenarios.

4.1 Matrix-Vector Multiplication on CUDA

Matrix-vector multiplication is common task that is also part of the BLAS Level 2 API (there called SGEMV). The number of necessary floating points operations in the case of a square $n \times n$ matrix A and a vector x of size n is $2\,n^2$, while the minimum number of memory accesses is $n^2 + 3\,n$ [MB08 p. 286]. This is why the algorithm is not very arithmetical intensive but more memory bound. Listing 3 shows a simple kernel implementation for this task. The algorithm assumes that a thread that executes the kernel performs all calculations for a single row of the matrix which is good for parallel write operations but not optimal in terms of memory accesses [MB08 p. 288].

```
__global__ void mv_kernel (float *A, float *x, float *y, int n) {
      int j;
      int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
      float t = 0.0f;
      for (j=0; j<n; j++) {
            t += A[i + j * n] * x[j];
      }
      y[i] = t;
}
```

Listing 3: Simple CUDA matrix-vector multiplication kernel [MB08 p. 288].

Better algorithms use the cache more efficiently. The key parameter for this is the optimal thread and grid layout. [Fu08] proposed improvements (in comparison to an older CUBLAS version) which group threads into two-dimensional blocks of size 16x16 and make use of the texture memory unit for accessing the matrix. Nvidia incorporates such improvements into their CUBLAS library. [MB08] also analyzed the case of banded and sparse matrices. Sparse matrices have also been discussed in [NBG08 p. 49-50]. Next, matrix-matrix multiplication will be analyzed, which is even more suitable for GPU processing (and experiments) because of the higher arithmetic intensity.

4.2 Matrix-Matrix Multiplication on CUDA

Matrix-matrix multiplication is also specified in the BLAS API (in this case Level 3, called SGEMM). In the case of two square $n \times n$ matrices A, B the number of necessary floating points operations is $2\,n^3$, while the minimum number of memory operations stays in $O(n^2)$. This means that the arithmetic intensity is within $O(n)$ and that the task is no longer memory bound.

The CUDA programming guide [NVI09d] gives two examples for matrix multiplication that can also be found in the SDK. The first one is straight forward solution where each thread reads a row of A and a column of B and computes the resulting element for matrix C (cf. Fig. 9). The solution does not make use of shared memory. Matrix A is read `B.width` times and B is read `A.heigth` times from global memory which makes the implementation rather slow [NVI09d p. 20-22]. The second solution is more elegant. Matrix C is subdivided into square matrices C_{Sub} assigned to thread blocks. C_{Sub} is calculated by multiplying two rectangles (A_{Sub} and B_{Sub}) (cf. Fig. 10). The rectangles are also subdivided into square matrices of length `BLOCK_SIZE` fitting into the shared memory. With this approach the number of global memory reads of matrix A is reduced to `B.width/BLOCK_SIZE` (resp. `A.heigth/BLOCKSIZE` for matrix B) [NVI09d p. 22-26]. The performance of the second algorithm has been tested in the following Section 4.3. The code for this can be found in Listing 4. The code defines a type `Matrix`, getters `GetElement()` and setters `SetElement()` for matrix elements and a routine for generating submatrices `GetSubMatrix()`. The host function `MatMul` allocates device memory for the matrices via `cudaMalloc()`, copies the data from the host to the device via `cudaMemcpy()` and invokes the kernel. The kernel `MatMulkernel` performs the multiplication as stated above. A_{Sub} and B_{Sub} are declerated as shared memory variables using the `__shared__` qualifier. In order to synchronize the threads before and after the inner loop `__syncthreads()` is called. After kernel execution the host function copies the result matrix C back to the host and clears the device memory via `cudaFree()`.

4.3 A Little Experiment

In the following section the performance of a CUDA will be measured by means of a dense square matrix-matrix multiplication in single precision. The experiments were performed

on a GeForce 8800 GTS in a Windows Vista 64-bit system equipped with an AMD Phenom 9650 Quadcore CPU (cf. Tab. 2, Tab. 3, and Tab. 4). In addition to the second algorithm presented above, the CUBLAS implementation of SGEMM [NVI09j] was also used. The CPU code was a simple iterative solution with three nested loops (performed on one core). The code has been slightly modified for measuring runtimes. The compilation was done with Visual Studio 2008. The measured runtimes have been averaged (from five iterations) and have been converted into GFlops assuming $2\,n^3$ floating points operations for two square $n \times n$ matrices A, B.

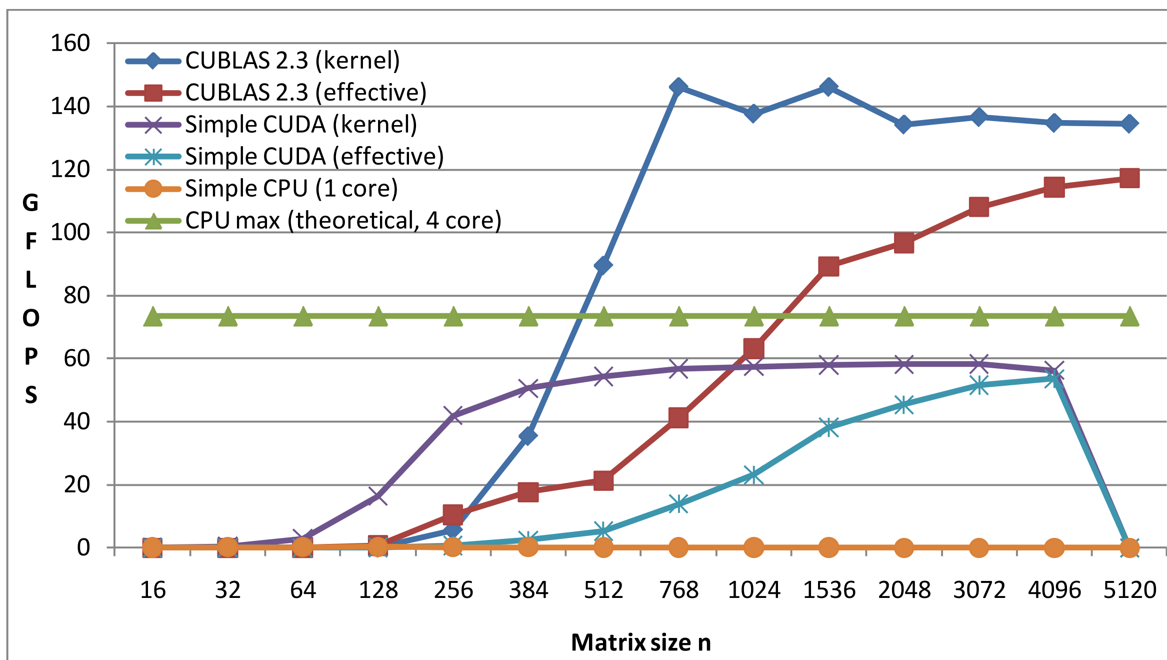

Fig. 7: Performance of the CPU and the GPU in square, dense, matrix-matrix multiplication (cf. Tab. 5 for values).

The results show that the simple CPU solution performs poorly. This is because the naïve serial algorithm was used which only ran on one core. Furthermore, a significant drop (from around 200 MFlops to about 60 MFlops) in the CPU performance was observed for $n \geq 512$ (cf. Tab. 5), which might be due to the fact that one matrix does not fit into the CPU L2 and L3 caches any more. A highly optimized BLAS library like ATLAS [AT09] or AMD AMCL [AMD09b] would achieve better results. For similar CPU (Phenom 9600) and matrix sizes between $n = 512$ and $n = 6272$ performances of between 14.9 and 15.9 GFlops (one core) using ACML SGEMM have been reported [Sa09]. However, the theoretical maximum performance of the used CPU is only 73.6 GFlops (18.4 per core). This means that for matrices $n \geq 1536$ the GPU would have outperformed even the best parallel algorithm for this CPU. Also note that the GPU was released two years before the CPU.

However the CPU solution can compete with the GPU for very small matrices ($n \leq 128$) (cf. Tab. 5). This is due the overhead of kernel invocation and memory transfer times, which compensate only in the case of large matrices. The highly optimized CUBLAS routine offers much more performance than the simpler CUDA kernel. Nevertheless, the GPU solutions achieve only about one third of their theoretical maximal performance (which is 346 GFlops), while the CPU achieves about 80 %. This becomes even clearer when including memory transfer times (effective timings), which are caused by the PCIe bottleneck. A further constraint is memory size. The matrix multiplication was not possible with the simple CUDA solution for $n = 5120$ because it ran out of memory. The main takeaway of this experiment is, that the GPU can significantly speed-up compute intensive operations like dense matrix multiplication, but that the overhead of memory transport to the GPU only makes only sense in case of large problems (in this case approx. for matrices $n \geq 1024$).

4.4 Application Scenarios

The main application scenario for GPU Computing is scientific computing which has an unstoppable request for computing power in various fields of natural sciences. Due to compute intensity and data parallelism the problems are well suitable for GPU processing. Furthermore, cost and energy efficiency are very good arguments for building supercomputers with GPUs. One example for a GPU super computer is the Tianhe-1, which is ranked #5 in the November 2009 release of the TOP500 list [TOP09]. The system uses 5120 AMD GPUs and achieves more than 500 TFlops (double precision). Applications include petroleum exploration and aircraft design. A second example is CSIRO's supercomputer cluster, which is currently constructed in Australia. It uses 64 Tesla graphic cards with a total of 256 GPUs and is supposed to achieve a performance of more than 256 TFlops (double precision). It will support computation in research areas like biology, chemistry, climate prediction, or astrophysics [CSI09]. Supercomputers usually use special HPC cards[9]. However it may be even more cost efficient to use consumer graphics cards. One example for this is the FASTRA II, developed und used by the Vision Lab of the University of Antwerp. It features seven graphic cards[10] built into a desktop tower, achieving a theoretical performance of 12 TFlops (single precision), with hardware costs of below

[9] Like Nvidia Tesla or ATI Firestream, which have a larger DRAM than consumer graphics cards.

[10] Six Nvidia GTX295 and one GT275 with a total of 13 GPUs.

6000 €. The system is used for medical research, especially creating three-dimensional computer tomography images with CUDA programs [UA09].

GPGPU is also a topic in distributed grid computing. One example is Folding@home, a project by the Stanford University that does protein folding for medical research [SU09]. The second example is the Berkeley Open Infrastructure for Network Computing (BOINC) [UC09] which hosts a variety of scientific applications. At the moment it offers six different projects for CUDA and one for ATI Stream. Both platforms offer a simple opportunity to donate unused computing power for the progress of science.

Besides scientific computing, there are only some interesting applications. One example are database systems that can be accelerated using CUDA, e.g. PostgreSQL [Ho09]. Furthermore, CUDA can accelerate digital image and video processing like H.264 encoding which is extremely compute intensive. Exemplary applications that promise to accelerate encoding between 30 % and a factor of ten are the Badaboom Media Converter (CUDA) or Cyberlink PowerDirector 7 (CUDA and Stream). However, these applications still offer less configuration possibilities and slightly lower visual quality than CPU encoders [BGT09 p. 92]. GPU computing is also used in security and forensics application, like calculating hashes for brute force password recovery, e.g. with Elcomsoft Distributed Password Recovery. It was shown that common password policies (like a password length of eight) may have to be revised due to GPU computing [AD09]. Physics simulation is not only an issue in scientific computing (e.g. in astronomy) but also in computer games, enabling more realistic physical effects like simulating gravity forces between thousands of objects. In the past this was done with the PhysX engine and dedicated physic accelerator cards e.g. from Ageia. After Nvidia took over Ageia in 2008 and integrated the engine into CUDA, it is now possible to use graphics cards as physic accelerators.

5 Summary and Outlook

The paper introduced the basics of GPGPU by having a look at computer graphics and graphics hardware. The stream processing programming model and the traditional GPGPU approach was presented. CUDA was introduced, including the programming model, hardware architecture, compilation and examples. The benefits and drawbacks have been evaluated. The experiment proved performance benefits for multiplication of large matrices and described further scientific application scenarios.

It should have become clear that GPGPU is an emerging technology, with the potential of improving the processing time of parallelizable algorithm significantly. This development could lead to a marginalization of the CPU for selected fields of application, especially in HPC where GPUs can substitute CPUs. Additionally, the differences between CPUs and GPUs are becoming smaller. While CPUs become more parallel, GPUs become more flexible programmable. The market is striving into a further integration of the GPU and the CPU. Intel has a project called Larrabee which is a cluster of x86 processors enhanced with vector instructions and also capable of graphics processing. During a presentation at the SCC09 in Portland an overclocked version achieved a SGEMM performance peak of 1006 GFlops (about 400 GFlops during normal operation). It was supposed to be released at the beginning of 2010 but has lately been canceled as a commercial product. It will instead only be used in research and development [Mü09]. An alternative many-core processor project is Intel's Singlechip Cloud Computer (SCC), which features 48 cores [INT09].

Further research on parallel programming and parallel computing will have to address standardization approaches for GPGPU programming (including platforms for heterogeneous landscapes) like OpenCL and DirectCompute. A second problem that has to be worked on is putting efficient programming of many-core processors into practice. At the moment the majority of application is not yet or poorly parallelized. With two or four cores this may be compensated by running several programs. But with hundreds and thousands this is no longer the case. The CUDA programming model can be a solution for this. It efficiently manages thousands of threads, which has been proved by several applications (cf. Chapter 4). Even if OpenCL and DirectX11 will be used more in the future of GPGPU, knowledge of CUDA will still be a valuable asset. This is first of all because knowledge of CUDA is the basis for OpenCL and DirectX11 DirectCompute and the main ideas do not differ much and secondly because CUDA GPUs and the development platform also support the two standards.

A Specification and Result Tables

	G80	GT200	Fermi
Release Year	2006	2008	2010
Fabrication Process	90 nm	55 nm	40 nm
Number of Transistors	681 million	1.4 billion	3.0 billion
Streaming Multiprocessors (SM)	16	30	16
Streaming Processors (per SM)	8	8	32
Streaming Processors (total)	128	240	512
Single Precision FP Capability	128 MAD ops/clock	240 MAD ops/clock	512 FMA ops/clock
Double Precision FP Capability	None	30 FMA ops/clock	256 FMA ops/clock
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB

Tab. 1: Comparison of the three major CUDA GPU architecture specifications [NVI09f p. 11] Note that the number of processors are maximum values.

Feature	Details
CPU	AMD Phenom X4 9650
Memory	6 GB DDR2-800
Operating System	Windows Vista SP2 64 Bit
GPU	PNY GeForce 8800 GTS
GPU Driver	ForceWare 190.62
GPU Software	CUDA Toolkit & SDK v2.3

Tab. 2: System specification.

Feature	Details
Name	PNY GeForce 8800 GTS
GPU	G80
#SPs (cores)	96
Release	2006
Interface	PCIe x16
Core Clock Rate	513 MHz
Shader Clock Rate	1188 MHz
Memory Clock Rate	792 MHz
Memory Size	640 MB GDDR3
Memory Bus Width	320 bit
Memory Bandwidth	63.4 GB/s
DirectX Support	10.0
OpenGL Support	3.2

Tab. 3: GPU specification.

Feature	Details
Name	AMD Phenom X4 9650
Release	2008
#Cores	4
Clock Rate	2300 MHz
L2-Cache	512 KB per core
Fabrication Process	65 nm

Tab. 4: CPU specification.

N	Simple CPU (1 core)	Simple CUDA (kernel)	Simple CUDA (effective)	CUBLAS 2.3 (kernel)	CUBLAS 2.3 (effective)
16	0.17	0.05	0.00	0.02	0.00
32	0.19	0.41	0.00	0.22	0.01
64	0.20	2.95	0.01	1.49	0.17
128	0.27	16.37	0.10	3.83	0.68
256	0.24	41.96	0.72	5.59	10.49
384	0.22	50.54	2.45	35.39	17.69
512	0.08	54.38	5.25	89.48	21.30
768	0.07	56.81	13.91	146.12	41.18
1024	0.05	57.47	23.18	137.66	63.16
1536	0.05	57.97	38.26	146.12	89.26
2048	0.04	58.18	45.37	134.22	96.73
3072	0.05	58.28	51.61	136.56	108.09
4096	0.04	56.26	53.60	134.72	114.44
5120	0.04	-	-	134.46	117.22

Tab. 5: Results of the Matrix-Matrix Multiplication Experiment for dense, square Matrices (GFlops).

B Additional Figures

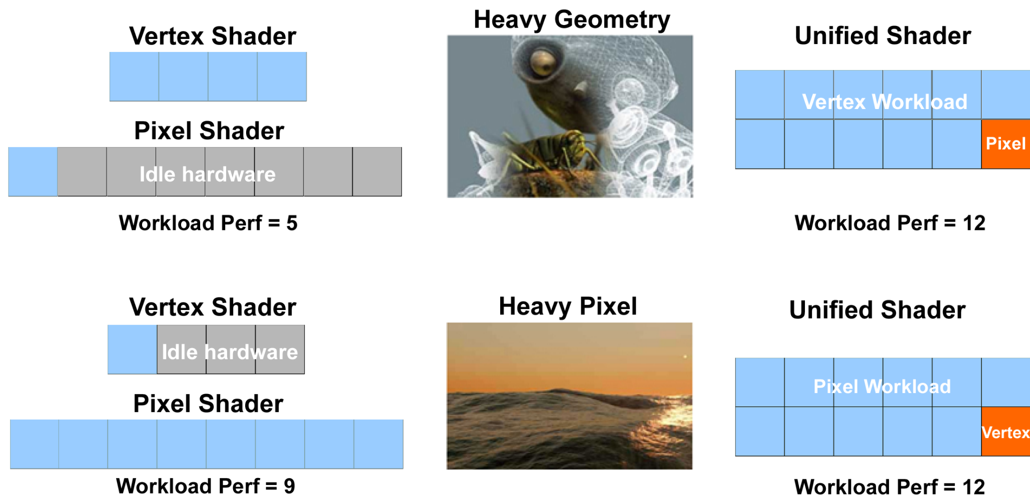

Fig. 8: Performance of a discrete vs. a unified shader architecture [NVI06 pp. 23-24].

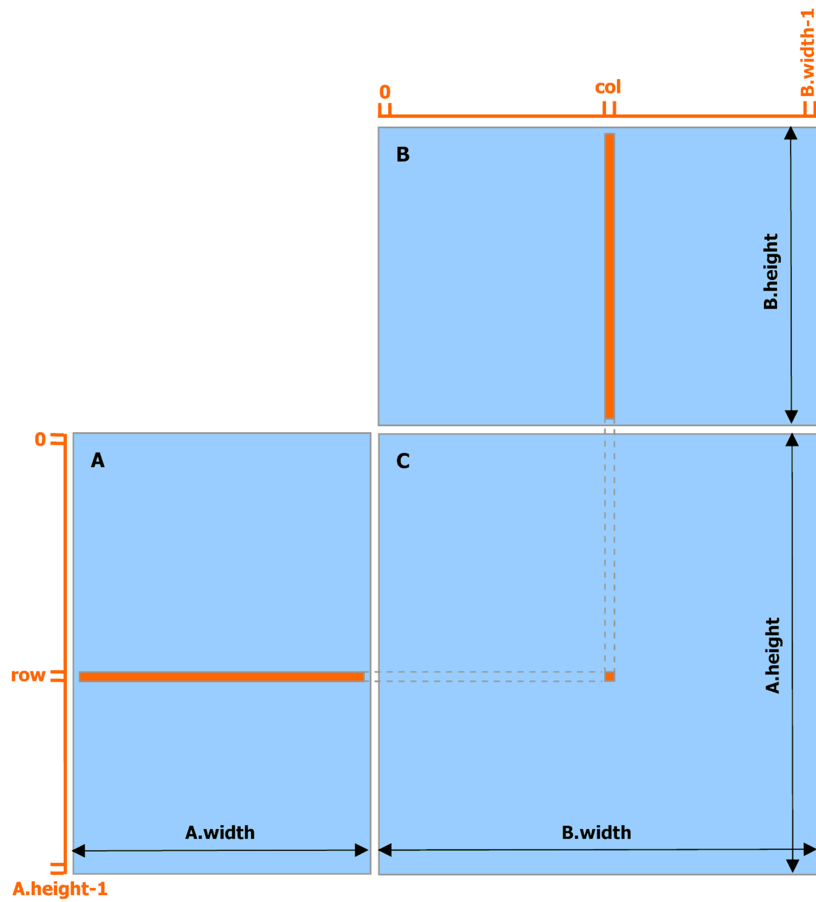

Fig. 9: Matrix-matrix multiplication without use of shared memory [NVI09d p. 22].

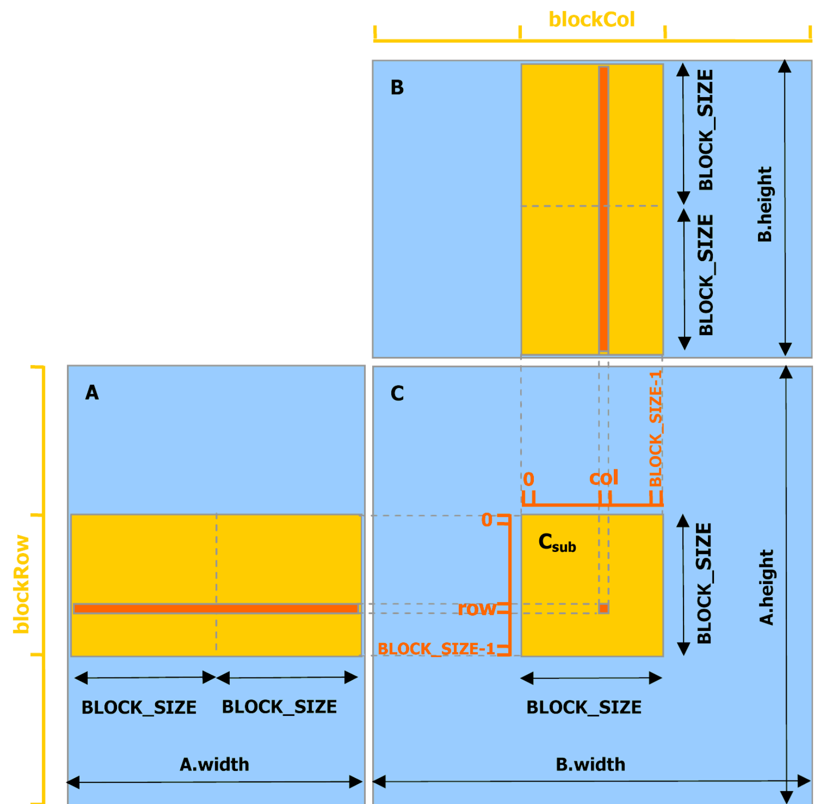

Fig. 10: Faster matrix-matrix multiplication with use of shared memory [NVI09d p. 26].

C Matrix Multiplication Listing

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
      int width;
      int height;
      int stride;
      float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col) {
      return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value) {
      A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
      Matrix Asub;
      Asub.width    = BLOCK_SIZE;
      Asub.height   = BLOCK_SIZE;
      Asub.stride   = A.stride;
      Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                    + BLOCK_SIZE * col];
      return Asub;
}
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C) {
      // Load A to device memory
      Matrix d_A;
      d_A.width = A.width; d_A.height = A.height;
      size_t size = A.width * A.height * sizeof(float);
      cudaMalloc((void**)&d_A.elements, size);
      cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

      // Load B to device memory
      Matrix d_B;
      d_B.width = B.width; d_B.height = B.height;
      size = B.width * B.height * sizeof(float);
      cudaMalloc((void**)&d_B.elements, size);
      cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

      // Allocate C in device memory
      Matrix d_C;
      d_C.width = C.width; d_C.height = C.height;
      size = C.width * C.height * sizeof(float);
```

```
    cudaMalloc((void**)&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size,
    cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
        // Synchronize to make sure that the preceding
```

```
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```

Listing 4: Matrix-Matrix Multiplication using shared memory [NVI09d pp. 22-25].

Bibliography

[AD09] Arbeiter, S.; Deeg, M.: Bunte Rechenknechte. Grafikkarten beschleunigen Passwort-Cracker. In C't: Magazin für Computer-Technik, pp. 204–206, 2009(6).

[AMD09a] AMD: ATI Stream Technology. http://www.amd.com/stream, 2009.

[AMD09b] AMD: AMD Core Math Library. http://developer.amd.com/Libraries/acml/pages/default.aspx, 2009.

[AT09] ATLAS: Automatically Tuned Linear Algebra Software. http://math-atlas.sourceforge.net, 2009.

[BB03] Bender, M.; Brill, M.: Computergrafik. Ein anwendungsorientiertes Lehrbuch. Hanser, München, 2003.

[Be09a] Bertuch, M.: 3D-Evolution. Microsofts 3D-Schnittstelle DirectX 11 im Detail. In C't: Magazin für Computer-Technik, pp. 174–177, 2009(8).

[Be09b] Bertuch, M.: Parallel-Werkzeuge. Datenverarbeitung auf Grafikkarten mit ATI Stream und Nvidia CUDA. In C't: Magazin für Computer-Technik, pp. 142–147, 2009(11).

[BFH04] Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers. ACM, New York, pp. 777-786, 2004.

[BGT09] Bertuch, M.; Gieselmann, H.; Trinkwalder, A.; Windeck, C.: Supercomputer zu Hause. Ungenutzte PC-Rechenleistung ausschöpfen. In C't: Magazin für Computer-Technik, pp. 88–95, 2009(7).

[Cr04] Crow, T. S.: Evolution of the Graphical Processing Unit. http://www.cse.iitb.ac.in/graphics/~anand/website/include/papers/gpu/gpu_evolution.pdf, 2004.

[Cr09] Crothers, B.: Nvidia CEO says 'no' to Intel-compatible chip. http://news.cnet.com/8301-13924_3-10393045-64.html, 2009.

[CSI09] CSIRO: CPU-GPU supercomputer cluster. Fact Sheet. http://www.csiro.au/resources/GPU-cluster.html, 2009.

[De09] Deilmann, M.: Das richtige Werkzeug. Eine Fallstudie zu parallelen Programmiermodellen. In Entwickler-Magazin: Software, Systems & Development, pp. 32–40, 2009(6).

[Fa08] Farber, R.: CUDA. Supercomputing for the Masses. http://www.ddj.com/cpp/207200659, 2008.

[Fe07] Fernando, R.: GPU Gems. Programming techniques, tips, and tricks for real-time graphics. Addison-Wesley, Boston, 2007.

[FK06] Fernando, R.; Kilgard, M. J.: The Cg Tutorial. The definitive guide to programmable real-time graphics. Addison-Wesley, Boston, 2006.

[Fu08] Fujimoto, N.: Faster matrix-vector multiplication on GeForce 8800GTX. In: International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, pp. 1–8, 2008.

[Ha06] Harris, M.: Mapping Computational Concepts to GPUs. In (Pharr, M.; Fernando, R. Ed.): GPU Gems 2. Programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley, Upper Saddle River, ch. 31, 2006.

[Ha08] Halfhill, T. R.: Parallel Processing with CUDA. In: Microprocessor Report, 2008(1).

[HA09] Han, T. D.; Abdelrahman, T. S.: hiCUDA: a high-level directive-based language for GPU programming: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. ACM, New York, pp. 52-61, 2009.

[HL09] Hoffmann, S.; Lienhart, R.: OpenMP. Eine Einführung in die parallele Programmierung mit C/C++. Springer, Berlin, 2009.

[Ho09] Hoff, T.: Scaling PostgreSQL using CUDA. http://highscalability.com/scaling-postgresql-using-cuda, 2009.

[Ih09] Ihde, H.: Shader mit GLSL. Eine Einführung in die OpenGL Shading Language. Diplomica, Hamburg, 2009.

[INT09] Intel: Single-chip Cloud Computer. http://techresearch.intel.com/articles/Tera-Scale/1826.html, 2009.

[KES09] Kindratenko, V. V.; Enos, J. J.; Shi, G.; Showerman, M. T.; Arnold, G. W.; Stone, J. E.; Phillips, J. C.; Hwu, W.-M. W.: GPU Clusters for High-Performance Computing: Workshop on Parallel Programming on Accelerator Clusters (PPAC), 2009.

[Kh09a] The Khronos Group. http://www.khronos.org/, 2009.

[Kh09b] The Khronos Group: OpenGL. http://www.opengl.org/, 2009.

[Ki03] Kilgard, M. J.: Cg in Two Pages. http://arxiv.org/pdf/cs/0302013, 2003.

[MB08] Maciol, P.; Banas, K.: Testing Tesla Architecture for Scientific Computing: the Performance of Matrix-Vector Product: International Multiconference on Computer Science and Information Technology, pp. 285–291, 2008.

[MRH09] Mensmann, J.; Ropinski, T.; Hinrichs, K.: An evaluation of the CUDA architecture for volume rendering, Münster, 2009.

[MS09a] Microsoft: Programming Guide for Direct3D 10. Pipeline Stages. http://msdn.microsoft.com/en-us/library/ee415715(VS.85).aspx, 2009.

[MS09b] Microsoft: Programming Guide for HLSL. http://msdn.microsoft.com/en-us/library/ee418343%28VS.85%29.aspx, 2009.

[MS09c] Microsoft: DirectX11. http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx, 2009.

[MS09d] Microsoft: Learn DirectX. http://msdn.microsoft.com/de-de/directx/bb896684%28en-us%29.aspx#articles, 2009.

[Mü09] Müssig, F.: Intel macht Rückzieher bei Larrabee. http://www.heise.de/newsticker/meldung/Intel-macht-Rueckzieher-bei-Larrabee-878138.html, 2009.

[NBG08] Nickolls, J.; Buck, I.; Garland, M.; Skadron, K.: Scalable Parallel Programming with CUDA. In ACM queue : tomorrow's computing today, pp. 40-53, 2008, 6(2).

[Ng08] Nguyen, H.: GPU Gems 3. Addison-Wesley, Upper Saddle River, 2008.

[NV09m] NVIDIA: CUDA ZONE. http://www.nvidia.com/object/cuda_home.html, 2009.

[NVI06] NVIDIA: GeForce 8800 GPU Architecture Overview. http://nvidia.com/object/IO_37100.html, 2006.

[NVI08b] NVIDIA: GPU Programming Guide GeForce 8 and 9 Series. http://www.developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf, 2008.

[NVI09a] NVIDIA: CUDA Architecture Overview v1.1. Introduction & Overview. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf, 2009.

[NVI09b] NVIDIA: CUDA C Programming Best Practices Guide. CUDA Toolkit v2.3. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf, 2009.

[NVI09c] NVIDIA: CUDA Development Tools v2.3. Getting Started. http://developer.download.nvidia.com/compute/cuda/2_3/docs/CUDA_Getting_Started_2.3_Windows.pdf, 2009.

[NVI09d] NVIDIA: CUDA Programming Guide v2.3.1. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf, 2009.

[NVI09e] NVIDIA: CUDA Reference Manual v2.3. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf, 2009.

[NVI09f] NVIDIA: Next Generation CUDA Compute Architecture: Fermi. Whitepaper v1.1, http://nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[NVI09g] NVIDIA: OpenCL Programming for the CUDA Architecture v2.3, http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingOverview.pdf, 2009.

[NVI09h] NVIDIA: OpenCL Programming Guide for the CUDA Architecture v2.3, http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf, 2009.

[NVI09i] NVIDIA: OpenCL Jumpstart Guide. Technical Brief v0.9, http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf, 2009.

[NVI09j] NVIDIA: CUDA CUBLAS Library v2.3, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/index.html, 2009.

[NVI09k] NVIDIA: CUDA Compiler Driver NVCC v2.3, 2009.

[NVI09l] NVIDIA: Compute PTX: Parallel Thread Execution ISA v1.4, 2009.

[Ow06] Owens, J.: Streaming Architectures and Technology Trends. In (Pharr, M.; Fernando, R. Ed.): GPU Gems 2. Programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley, Upper Saddle River, ch. 29, 2006.

[PCI09] PCI-SIG: PCI-e specifications. http://www.pcisig.com/specifications/pciexpress/, 2009.

[PF06] Pharr, M.; Fernando, R.: GPU Gems 2. Programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley, Upper Saddle River, 2006.

[Sa09] Saastad, O. W.: Memorandum on Graphic Processing Unit (GPU) performance. http://folk.uio.no/olews/Memorandum-on-GPU-performance.pdf, 2009.

[SSH08] Stratton, J. A.; Stone, S. S.; Hwu, W.-M. W.: MCUDA. An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In (Amaral, J. N. Ed.): Languages and compilers for parallel computing. 21st international workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008 ; revised selected papers. Springer, Berlin, pp. 16–30, 2008.

[St97] Stephens, R.: A survey of stream processing. 34(7). In Acta Informatica, pp. 491–541, 1997.

[SU09] Stanford University: Folding@home. http://folding.stanford.edu, 2009.

[SWN06] Shreiner, D.; Woo, M.; Neider, J.; Davis, T.: OpenGL Programming Guide. The Official Guide to Learning OpenGL Version 2.1, 2006.

[TB08] Tanenbaum, A. S.; Baumgarten, U.: Moderne Betriebssysteme. Pearson Studium, München, 2008.

[TOP09] TOP500: Supercomputing Sites. http://www.top500.org/, 2009.

[UA09] University of Antwerp: FASTRA II. the world's most powerful desktop supercomputer. http://fastra2.ua.ac.be/, 2009.

[UC09] University of California: Use your GPU for scientific computing. http://boinc.berkeley.edu/gpu.php, 2009.

[ULB08] Ueng, S.-Z.; Lathara, M.; Baghsorkhi, S. S.; Hwu, W.-M. W.: CUDA-Lite: Reducing GPU Programming Complexity. In (Amaral, J. N. Ed.): Languages and compilers for parallel computing. 21st international workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008; revised selected papers. Springer, Berlin, pp. 1-15, 2008.

[Wi09] Windeck, C.: Nvidia-Chipsätze künftig auch nicht mehr für AMD-Prozessoren. http://www.heise.de/newsticker/meldung/Nvidia-Chipsaetze-kuenftig-auch-nicht-mehr-fuer-AMD-Prozessoren-877786.html, 2009.

[Wü09] Wüst, K.: Mikroprozessortechnik. Grundlagen, Architekturen, Schaltungstechnik und Betrieb von Mikroprozessoren und Mikrocontrollern. Vieweg+Teubner, Wiesbaden, 2009.

[XP07] Xiang, Z.; Plastock, R. A.: Computergrafik. Einführung in die theoretischen Grundlagen. mitp, Bonn, 2007.