FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Efficient Implementation of Finite Element Operators on GPUs

Severin Strobl

Studienarbeit

Efficient Implementation of Finite Element Operators on GPUs

Severin Strobl

Studienarbeit

Aufgabensteller:	Prof. Dr. C. Pflaum
Betreuer:	Prof. Dr. C. Pflaum
Bearbeitungszeitraum:	01.01.2008 – 02.09.2008

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 1. September 2008 .....

Abstract

Graphics processing units (GPUs) of the current generation are highly programmable using interfaces like NVIDIA's Compute Unified Device Architecture (CUDA) while offering a massive computational power for a relatively low price. The solution of systems of linear equations described by stencils representing finite element operators by iterative solvers is a application which can utilize these benefits of GPUs. This thesis describes the implementation of a parallelized Gauss-Seidel solver achieving up to 40 times the performance of a corresponding solver on standard CPUs. The solver is part of a library named GPUsolve, which offers the infrastructure to integrate this GPU-enabled application into object oriented C++ code.

Contents

1 Introduction

Over the last two decades graphics processing units (GPUs) have experienced an astounding evolution. From quite simple devices to bring 2D graphics to the user's display to highly complex, massively parallel high performance computing devices. Mainly driven by the needs of the game industry for faster and more detailed rendering of virtual 3D worlds, some years ago GPUs reached performance levels previously reserved to the main processor in a computer.

But GPUs not only became more powerful, also the possibilities to execute user supplied code on certain parts of the hardware increased. Over the time people began porting their memory or computation intensive codes to this new architecture. The field of general-purpose computing on graphics processing units (GPGPU) was born. First the interfaces for 3D graphics were used for this, while recently the main GPU hardware vendors began releasing special toolkits including own language extensions and compilers to simplify the programming of their hardware.

One of the many applications that might benefit from the enormous computational power of recent GPUs are iterative solvers for large systems of linear equations as they occur by the discretization of partial differential equations (PDEs) using the finite element method (FEM). Applying the FEM to PDEs allows the simulation of many physical problems like fluid dynamics, linear elasticity, heat transport or simulation of electromagnetic fields for example in solid state lasers.

For special cases the differential operators of the finite element method may be described using stencils which are generally easier to handle than the corresponding sparse matrices. This thesis focuses on the implementation details of an iterative method to solve such stencil based systems of linear equations using CUDA, NVIDIA's GPGPU technology. The aim was to create an easy to use library offering a fast iterative solver, namely a multi-color version of the Gauss-Seidel algorithm, for integration into finite element code. Although some parts are specific to this solver, most general optimization techniques for porting similar algorithms to GPUs are discussed. In order to achieve a clean and simple interface, the code was written in C++, making heavy use of the template programming concept of the language. The resulting library is called GPUsolve and offers a fast GPU accelerated iterative solver together with a highly object oriented infrastructure.

2 GPU Programming

In order to fully understand the potential and also the pitfalls when porting applications to GPUs, it is of some help to look at the underlying hardware as well as the different programming methods. This is only intended as a short overview introducing the basic concepts. A slightly more extensive description can be found in Owens et al. [2008], additional in-depth details in Pharr [2005, chap. 29].

2.1 GPU Hardware

Before GPUs were used as powerful parallel co-processors, their function was to render the abstract geometric objects created by various programs running on the CPU so they could be displayed to the user. While starting out as rather simple devices allowing the displaying of 2D data, they evolved into powerful and highly complex special purpose processors. At the time the first GPGPU efforts were made, the rendering pipeline of common devices consisted of several separated stages, as depicted in figure 1. This description only provides a rough overview over the very complex structure of today's hardware and focuses on the most fundamental features.

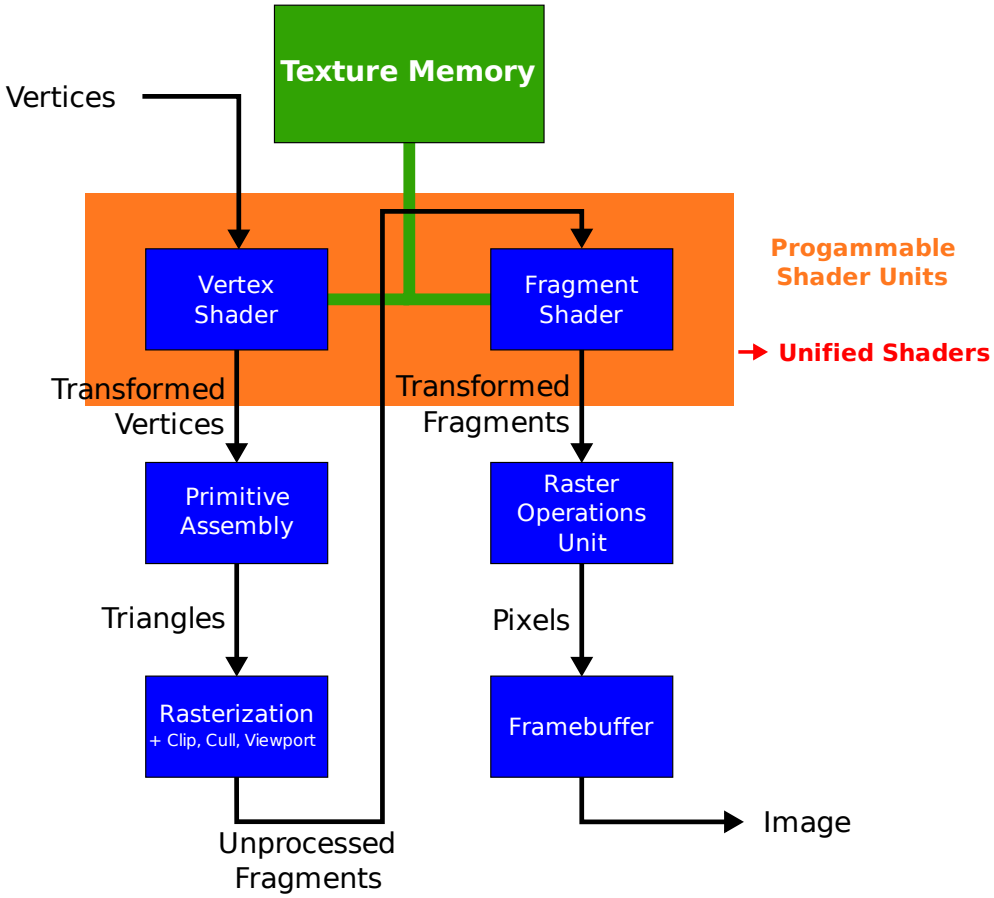


Figure 1: Rendering pipeline of current GPUs

The input to the GPU consists of vertices representing 3D objects in the user generated world. In the vertex processing steps, vertices can be transformed by user defined programs (Vertex Shaders). In the next steps the transformed vertices are converted to triangles. To

these some tests are applied to remove unneeded elements (clipping and culling) and mappings to the selected viewport are performed. After wards the triangles are fed to the rasterizer, which determines the single pixels of each primitive, producing so called fragments. These fragments are passed to the other programmable shading unit, the Fragment or Pixel Shader. Here colors from textures as well as additional complex effects like scene lightning are applied to each fragment. This shading unit is both highly programmable and parallel. As such it was the primary target for general purpose programming on graphics hardware. For further details on current GPU pipelines see Pharr [2005, chap. 30] and Owens [2007].

Both the programmable Vertex and Fragment/Pixel Shaders are connected to the texture memory and operate on independent input data in parallel offering basically the same instruction sets. Because of this similarity, the concept of the so called Unified Shaders was introduced. These processing units can serve as Vertex and Fragment Shaders and also as the newly introduced Geometry Shaders, which allow a more extensive manipulation and also the creation of new vertices in the graphics pipeline. The highly parallel processing units are not placed in separate parts of the GPU any more; instead one single central unit combines all the features on a higher abstraction level. The Unified Shading Unit consists of a group of parallel processing units (often called multiprocessors), each composed of several simpler stream processors along with special function units for memory access, instruction fetching and scheduling. While the first shaders worked in a single-instruction multiple-data (SIMD) fashion, later on support for branching was added. So the new streaming processors are best described as single-program multiple-data (SPMD) units [Owens et al., 2008]. For a detailed description of the multiprocessors found in current NVIDIA hardware, see section 3.2.

It was after this step to Unified Shaders, that GPU hardware vendors started to actively support general purpose programming on their hardware using this new essential part of current GPUs through special, not graphics programming centered interfaces. The programmer now does not have to restrict himself to one of the separated shading units, but he can use all the available power of the GPU. These efforts are described in part 2.3, beforehand the older method of shader programming is described.

2.2 Shader Programming

Before the hardware manufacturers joined the GPGPU efforts, the only methods to program the hardware shading units on GPUs were using the special assembly language ARB standardized by the OpenGL Architecture Review Board and later on the extended languages from the OpenGL and Direct3D APIs. Over the last years the focus shifted away from ARB to the higher-level interfaces. Although they offer almost the same range of functions, OpenGL is more popular because of its status as a free standard and its platform independency. These APIs serve only as a framework to communicate with the graphics hardware and to load and execute shader programs. The shaders themselves are written in special higher-level shading languages. These include the OpenGL Shading Language (GLSL) for use with OpenGL and Microsoft's High Level Shader Language (HLSL) for Direct3D. A shading language working with both APIs is NVIDIA's C for Graphics (CG) which comes with it's own optimized compiler, allowing output of GLSL and HLSL programs. All these languages offer data types like vectors with 3 or 4 components (RGB and alpha channel) and texture management intended for 3D graphics programming. General purpose programming with these shading languages is quite cumbersome, as all data has to be represented using textures and the interface is centered around typical graphics programming with its own data types and basic elements like vertices and matrix transformations. To overcome this drawback some higher-level languages and toolkits have been developed, some of which are described below.

2.2.1 BrookGPU

BrookGPU [Buck et al., 2004] is an implementation of the stream oriented programming language Brook on GPUs also developed at Stanford University. Brook is an extension to standard ANSI C and developed around the streaming concepts of data parallelism and arithmetic intensity. The so called computation kernel operates on one or more input streams (comparable to multi-dimensional vectors), performs arithmetic operations and generates output streams containing the results. The streams and kernels are mapped to the specific architecture of GPUs using textures and the SIMD possibilities of the hardware.

BrookGPU offers a multitude of back-ends: code generation for shaders using Direct3D and OpenGL is supported, as well as an interface to AMD's (former ATI's) Close To Metal (CTM), which is described in greater detail in section 2.3.2. For testing and debugging a CPU back-end also exists, making it possible to use the standard development and debug tools while writing applications.

2.2.2 Sh

The other popular project centered around higher-level shader programming is Sh. It is build on top of C++ using advanced object oriented language features to create a meta programming language. Sh started out as a research project at the University of Waterloo, evolved into a community driven project and is developed by RapidMind Inc. since 2004. The shader code is compiled and optimized at run-time, generating shaders using ARB or GLSL for GPUs. Sh itself is only a C++ library, no special compiler or language extensions are needed. It offers full support for user defined functions and types as well as classes and generic programming using templates, as all this functionality is contained in C++ already. Like BrookGPU, Sh also uses a stream concept for general purpose programming, but additionally supports graphics programming.

2.3 Hardware Vendor Approaches

After the introduction of unified shaders in graphics hardware, the two main vendors of dedicated GPUs, NVIDIA and ATI (now a part of AMD), started their own efforts to make GPGPU easier and more attractive to a larger group of developers. SDKs and low-level APIs for interaction with the hardware were developed and made available to the community.

2.3.1 NVIDIA CUDA

NVIDIA's GPGPU concept is named *Compute Unified Device Architecture* (CUDA), consisting of a low-level driver API to the hardware, the CUDA runtime library, various higher-level libraries providing BLAS and FFT functions as well as a SDK. The single components of CUDA are described in part 3. Using CUDA it is possible to program GPUs using standard C and specific libraries.

GPU programming using CUDA is thread-oriented. A grid of thread blocks is created, commonly consisting of several hundreds single threads. Each thread can be identified using a unique identifier, depending on the position of the thread in the block. A more verbose description of the programming model can be found in the next section. The threads can perform arbitrary functions, integer as well as floating point (single precision in current hardware only) operations are supported. The memory on the graphics card can be used to store both plain data and textures. The computation is done in so-called kernels, which are executed for each block in the grid independently. Communication between the blocks is only possible over global memory, threads in one block can use a portion of shared memory for data exchange.

In contrast to the standard graphics cards, NVIDIA offers specialized hardware (Tesla series) containing multiple GPU cores and more memory. Tesla products are available in the form of traditional extension cards, but also rack mountable versions containing up to four GPU cores are available. These hardware components are optimized and usable only for general purpose programming.

It has to be noted that the term *Tesla* is used in two contexts by NVIDIA: Both the new unified GPU architecture and the specialized versions of graphics hardware for GPGPU share this name.

2.3.2 AMD Stream

The AMD Stream Computing project evolved from ATI's Close To Metal (CTM). As AMD took over ATI, CTM was extended and eventually became the basis for AMD Stream.

The main components of the AMD Stream SDK are, descending from higher level to hardware:

- AMD Core Math Library (ACML): BLAS, LAPACK, FFT functions
- Brook+: optimized version of the Brook compiler for AMD hardware
- AMD Compute Abstraction Layer (CAL): low-level interface to GPU along with special intermediate language (based on CTM)

Additional development tools like the GPU ShaderAnalyzer and AMD CodeAnalyst were published. AMD also works closely with RapidMind (see section 2.2.2) to fully integrate their GPGPU framework into the commercial version of Sh.

AMD also offers special hardware for computations only: the FireStream line. They contain high-end GPUs and extended memory and also support double precision floating point operations in hardware in the newest versions.

2.3.3 Comparison

As AMD Stream uses an extended version of the Brook compiler, it is clearly stream oriented, in contrast to NVIDIA's CUDA thread oriented approach. On the higher levels, the differences are less obvious, as both approaches offer the typical numerical functionality of BLAS and FFT.

Common to both implementations are the not fully IEEE 754 compliant floating point operations: mainly rounding, division and handling of NANs differ from the standard. However it has to be evaluated for each project independently, if this poses a problem. The newest GPU series of both vendors now support double precision, however the rounding issues still seem to exist.

At the moment only NVIDIA's CUDA fully supports Linux, which may be based on the traditionally better driver support for alternative operating systems by NVIDIA.

3 NVIDIA CUDA

3.1 NVIDIA Hardware

The hardware of current NVIDIA GPUs (GeForce 8 Series and later) is centered around the up to 128 parallel streaming processors (SPs) in groups of texture/processor clusters (TPCs). Each of the TPCs consists of 16 streaming processors organized in two streaming multiprocessors (SMs), a SM controller, a texture unit with a cache and graphics related functionality. Each SM has 8 streaming processors with a portion of shared memory for communication, two caches, one for instructions, one for constant data, a multi-threaded instruction unit and two special function units. For a detailed description of the hardware and its usage for graphics and general purpose computing see Lindholm et al. [2008].

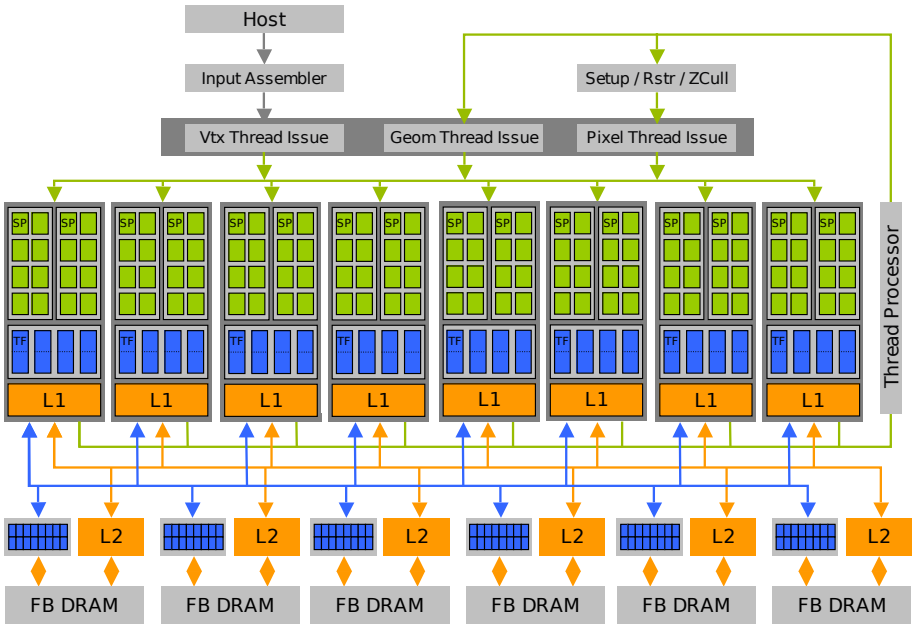


Figure 2: Diagram of the internals of the NVIDIA GeForce 8800 series GPU [Lindholm and Oberman, 2007]

All traditional operations of programmable shaders can be implemented using threads. Vertex shaders operate on one vertex at a time, pixel shaders on one pixel. The high performance of GPUs is of course only achievable, if a lot of these shader threads are executed in parallel. So the hardware has to support a huge amount of parallel threads and schedule them efficiently, so the streaming processors are all used to the best.

Each SM can manage up to 768 lightweight threads, providing fast creation, scheduling without overhead and efficient barrier synchronization. In order to achieve this, the threads are grouped into so-called *warps* of 32 threads respectively. These warps are processed in a single-instruction, multiple-thread (SIMT) manner. Each thread in a warp has its on state in terms of registers, but all members execute the same instructions. In order to support branching of threads in a wrap, single threads can be inactive during the execution of instructions. So different branches in a program are serialized with the respective threads being active or not, unless all threads in a wrap follow the same path. This implies, that the optimal performance is only reached if all threads in a wrap follow the same path in a program. However the Tesla architecture with its warps of 32 threads offers a highly superior performance for the execution of branching code than former hardware.

This new Tesla hardware architecture can be found in all current GPUs of NVIDIA. It is very flexible in terms of the actual number of streaming processors, as it is possible to have any amount of TPCs from 1 to 8, and possibly even more in future hardware. The overall structure stays the same, which means both shaders for graphics processing as well as computing applications will work on every GPU using this new architecture. The main differences between GPUs are reduced to the number of TPCs, the attached memory and of course the clock rates of the various chips.

During the work for this thesis, two GPUs from the GeForce 8 Series were used; a high-end GeForce 8800 Ultra and for development and debugging a mid-range GeForce 8600 GT. The main characteristics can be found in table 1.

	GeForce 8600 GT	**GeForce 8800 Ultra**
Streaming Multiprocessors	4	16
Streaming Processors	32	128
Core Clock Rate (MHz)	540	612
Shader Clock Rate (MHz)	1180	1500
Global Memory (GDDR3)		
Amount (MB)	256	768
Clock Rate (MHz)	700	1080
Interface (bit)	128	384
Bandwidth (GB/s)	22.4	104
Resources per Multiprocessor		
Registers	8192	
Shared Memory (KB)	16	
Constant Memory Cache (KB)	8	
Texture Cache (KB)	8	

Table 1: Hardware specifications for used GPUs

3.2 CUDA Programming Model

Performing computations using CUDA is done via *kernels*, user-written programs loaded onto the GPU and executed by the Tesla architecture. The kernels are based on threads, so they can be mapped to the hardware efficiently. As the high performance of GPUs can only be achieved when using large number of threads, the most important task when writing CUDA enabled programs is to break down the problem into many small tasks which can be executed in parallel. This is not possible for any given problem, but parallel algorithms are a well explored field in computer science.

To simplify the management of such large numbers of threads, CUDA offers the concept of grids and thread blocks. The computation domain is broken down into a one or two dimensional grid of blocks. Each of these blocks can currently contain up to 512 threads, organized in a one to three dimensional grid (see figure 3 for an example). The blocks are mapped to the SMs, the single threads to the SPs in one SM. The CUDA API offers means to identify both a block's position in the grid and a single thread's position in a block. This can be done using the build in variables `blockIdx` and `threadIdx` which both are vectors with three components containing the actual index. Using this variables it is possible to identify individual threads for handling special cases like boundaries of the computation domain. It

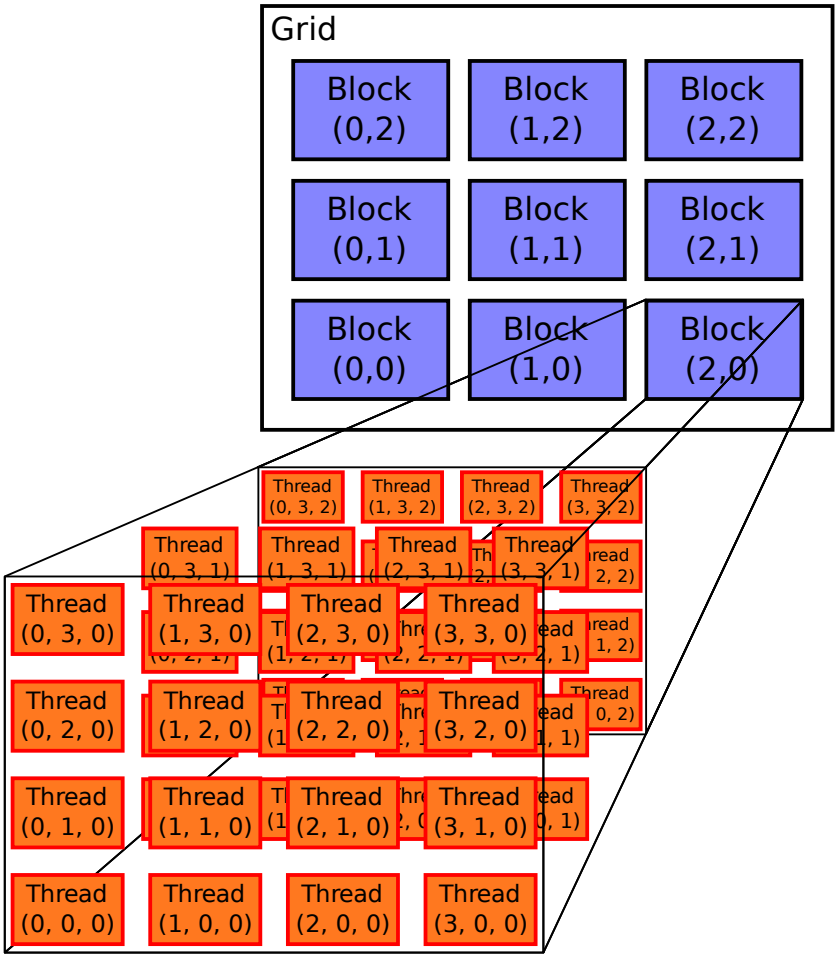


Figure 3: Grid of 3x3 blocks each with 4x4x3 threads

has to be emphasized once again, that the real potential of the GPU hardware only is reachable when using both multiple blocks and many single threads inside these blocks.

Synchronization between threads in one block is possible using the CUDA `__syncthreads()` primitive in a efficient way. Between threads of different blocks a explicit synchronization is not possible. But subsequent calls to the same kernel can be used to assure that all threads in all blocks have finished their work.

It is important to note that the developer has no means to influence the order in which single blocks or threads within blocks are processed. This is done by hardware schedulers which try to utilize the hardware in the most efficient way. Because of this the user has to take care of proper synchronization between the single threads.

3.3 Memory Model

The memory hierarchy on CUDA enabled hardware is somewhat different from the familiar one on the CPU side. The most important differences are probably the complete lack of any stack-like model for the hardware threads and the fact that only constant memory (and texture memory) is cached automatically.

Inside the SMs, a rather large number of registers is available. These currently 32 bit wide registers are very fast, but cannot be addressed directly. The NVIDIA compiler determines, how many registers per thread are needed and distributes the hardware registers according to this. The number of threads running in one SM may actually be limited by the amount of

registers in use. As on the CPU side, operations on data are only possible, if it is stored in registers. If the amount of registers in the hardware is not sufficient, the *local memory* may be used as an intermediate store. Despite its name, the local memory is not very efficient both in terms of bandwidth and latency as it is implemented in the off-chip DRAM. Whenever possible its usage should be avoided.

To allow the single SPs to communicate among one another, each SM contains a portion of 16 KB on-chip memory. Access to this memory region is almost as fast as to registers, provided the threads don't block each other. As the shared memory is arranged into 16 banks, it is possible for all threads to access data without generating bank conflicts. For details on this specific problem see NVIDIA Corporation [2008c, section 5.1.2.5]. As described in Harris [2007], achieving high performance of memory bandwidth limited algorithms often relies on using the shared memory as a sort of software managed cache.

Both the registers and the shared memory can not be accessed from the host side, but only from the GPU.

The main data storage on GPUs is the *global memory* located in external DRAM, which is capable to store several hundred MB of user data. The global memory supports both read-only and read-write spaces, as well as storage of textures. Both read-only data and textures are cached automatically within the GPU, but as the hardware has no protocols for cache coherency this is obviously not possible for read-write memory. The global memory can be fully accessed from both the GPU and the CPU making it the exchange point for all user data. Although the memory interface between the GPU core and the device memory is quite powerful, special steps have to be taken to ensure optimal performance (section 3.3).

In terms of the thread, block and grid model the various memory spaces are:

- thread
  - registers (read-write)
  - local memory (read-write)
- block
  - shared memory (read-write)
- grid
  - global memory (read-write)
  - texture memory (read-only)
  - constant memory (read-only)

3.4 SDK

NVIDIA offers several separated modules containing the necessary components to run and create CUDA enabled applications.

The basis for all general-purpose computing on graphics processing units using CUDA is the driver component, which is described section 3.5.1. The library containing this driver interface has to be installed along with some basic header files to enable the system to execute CUDA applications.

Atop the driver API the CUDA toolkit[1] needs to be installed. It contains quite a lot of different parts of the CUDA SDK and along with the driver component is sufficient to run compiled CUDA applications.

The CUDA runtime library (libcudart) is the interface for lower-level programming of NVIDIA GPUs (see 3.5.2). The specific function declarations are contained in a number of header files.

The included NVIDIA CUDA compiler named *nvcc* is actually more a wrapper around several different compilers, which is the reason for its official characterization as a compiler driver. The purpose of nvcc is to allow the developer to write C/C++ programs to run in parts on the GPU. The code that actually is transformed to Tesla compatible instructions is contained in kernels and only allowed to use the C subset of C++. The surrounding code however may make full use of all C++ features like object orientation or template programming.

The CUDA front end (cudafe) divides up the user code into the host and the device code. The parts of the code that run on the host side including all calls to the driver/runtime API are compiled using the system's standard C/C++ compiler, for example the GNU compiler suite. Also some preprocessing on all parts is done using the external compiler. The kernels containing the code targeted at the GPU (device code) are fed to the nvopencc. This compiler is based on the open source Open64 optimizing compiler. It transforms the C code into an intermediate language for the PTX (Parallel Thread Execution, ref PTX) virtual machine. The idea behind PTX is to create an *instruction set architecture* (ISA) independent of the actual hardware implementation Tesla. By this additional abstraction layer it is possible to keep the instruction set constant even if the underlying hardware changes in future releases. However PTX can be mapped almost directly to the current Tesla architecture. This mapping is done using ptxas, an assembler performing the transformations from PTX to real hardware instructions.

Also as a part of the CUDA toolkit, two additional libraries for numerical computations are installed. The first is CUBLAS, the CUDA version of the Basic Linear Algebra Subprograms. It offers BLAS level 1, 2 and 3 routines and is intended to be used with C or C++. However it is also possible to use it together with FORTRAN, from where characteristics like column-major and 1-based indexing have been adopted [NVIDIA Corporation, 2008a].

The second library is CUFFT [NVIDIA Corporation, 2008b], which brings GPU accelerated Fast Fourier Transformations for 1 to 3 dimensional data. This library is developed by NVIDIA directly and due to this is very highly optimized.

An important part in the toolkit is the documentation. The CUDA Programming Guide [NVIDIA Corporation, 2008c], various references for the CUDA libraries, the compiler [NVIDIA Corporation, 2008e], PTX [NVIDIA Corporation, 2008f] and the higher level libraries are available. Sine the 2.0 Beta version, also man pages for all library calls are included.

The third package contains the CUDA SDK code samples. This is a collection of over 50 sample applications. They offer a good starting point for beginners seeking for complete, optimized, mostly well documented CUDA code. A short extract from the many available examples:

- matrix multiplication
- eigenvalue calculation

[1] http://www.nvidia.com/object/cuda_get.html

- n-body simulation
- fluid dynamics
- image denoising

All projects make use of a common infrastructure from the CUDA Utility Library (CUTIL). It offers support for file handling in binary and the PPM image format. Some preprocessor macros are included to simplify error checking for CUDA calls. It also features functions to check CUDA kernel for shared memory bank conflicts.

To help new users in creating own applications using CUDA, there is a template project with a matching Makefile also making use of CUTIL. The user does not have to know the depths of the CUDA SDK to begin working with it and can concentrate on the task at hand.

On the NVIDIA homepage, some smaller, but never the less very useful tools are available. When writing CUDA enabled programs, it is often hard to determine which part of the code performs well and where some bottlenecks may lie. Here the CUDA Visual Profiler comes in handy. Using this tool it is possible to time memory copies, kernel calls and get exact information about the actual execution of the kernels. Counters for the number of serialized threads exist as well as for the number of misaligned memory accesses. When optimizing CUDA applications, the profiler can be of great help.

Also very helpful is the CUDA Occupancy Calculator[2], a spreadsheet able to determine how many blocks of threads may run in parallel on a specific GPU, depending on the number of resources (registers, shared memory) the threads use. With this tool one can estimate how well a specific implementation makes use of the capacity of the GPU.

3.5 CUDA API

The CUDA API consists of largely three parts: the driver API for direct interaction with the hardware, the runtime API for writing CUDA enabled user programs and various higher-level APIs for easy to use standard libraries.

3.5.1 Driver API

The direct interface to the GPU hardware is contained in the driver API. A wide range of functions are available on this level. It is possible to manage the installed devices and request information from the hardware concerning the features and details of the GPU like the amount of available memory or the *Compute Capability*, determining the functions supported by the hardware. Context management to prepare the hardware for and clean up after user programs is handled by this library. Also the loading and execution of user generated code on the GPU is handled via this API. Memory and texture management along with the handling of streams containing multiple GPU operations like memory copies and kernel launches can be done. Events usable for example to time kernel calls on the GPU can be generated and monitored. Also interfaces for the graphic APIs OpenGL and Direct3D are available.

Using the driver API is not very comfortable and error prone, as all initializations have to be done explicitly by the user. The runtime API is much easier to use and takes care of many low-level tasks, but still provides all needed features.

Since the 1.1 release of CUDA, the driver API is distributed together with the display driver and therefore automatically installed on systems with NVIDIA hardware.

[2] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

3.5.2 Runtime API

The runtime API offers basically the same functionality as the driver API. Device, memory, texture, stream and event management is possible, although the functions are somewhat simpler to use, as the user doesn't have to initialize the device explicitly and create an appropriate context. This is done automatically before to first library call. Also the code to be executed does not have to be loaded onto the device by hand, a simple call of the corresponding kernel is sufficient. As the functions in the driver and runtime layer are almost the same in their role, so are their names. To differ between the two versions, all functions in the driver API begin with the characters *cu*, while the equivalent counterparts in the runtime library start with *cuda*. For example `CUresult cuMemAlloc(CUdeviceptr* devPtr, unsigned int size)` is the driver API call analog to `cudaError_t cudaMalloc(void** devPtr, size_t size)` from the runtime API.

4 Stencil-based Iterative Solvers

4.1 Stencils as Finite Element Operators

The Finite Element Method (FEM) is probably the most widely used approach to solve physical problems described by partial differential equations (PDEs) like fluid simulation, heat transport problems or the simulation of electromagnetic fields. A detailed description of this technique as well as a proper mathematical formulation is beyond the scope of this work, but can be found in any standard textbook like Braess [2007].

The important property of the FEM in this context is that a large system of linear equations has to be solved using numerical algorithms. The system is described by a sparse $n \times n$ matrix (n being the number of discretization points), which describes the relationship and interaction between the various points in the domain represented as a grid. Depending on the type of finite elements (determined by the basis functions) and the method to generate the discretization grid, the number of entries per grid point in the system matrix varies. For typical basis functions and grids, the number of entries in a row is relatively small, as a single point in the grid is only connected to a limited number of other points, normally his direct neighbors.

For example when solving the Poisson's equation

$$\begin{aligned} -\Delta u &= f \quad \text{on } \Omega = (0,1)^2 \quad \text{with} \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

using regular triangles, linear basis functions and a mesh size h, the entries in the matrix are even the same as the ones created by applying the much simpler finite difference method to the same problem. For the derivation of this result see example 4.3 in Braess [2007]. Based on this the system matrix can be described by the well known 5-point stencil:

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}.$$

So for this finite element discretization, the entries in the system matrix occur at certain well known positions. This is a property also present for slightly more complicated problems. As long as the grid is structured so it is possible to determine the fixed number of neighbors for any grid point directly and the base elements are not too complex, the system matrix is sparse and all the entries are at known positions. In this case it is reasonable to not store the matrix as a whole using optimized data structure for sparse matrices (e.g. compressed row storage), but to allocate the entries in one row to the corresponding grid point in a stencil. The size of the stencil depends on the dimension of the domain, the used basis elements and the PDE itself. For the previous mentioned solution of the Poisson's equation in 2D, a single 5-point stencil is sufficient. This stencil is the same for all grid points in the domain. For the more general 3D case using hexahedra as subdivisions for the domain along with possible variable coefficients in the PDE, a 27-point stencil (figure 4) is needed for each point.

4.2 Stencils in UGBlock

The Unstructured Block Grid (UGBlock) part of the ExPDE (Expression Templates for Partial Differential Equations) project[3] is a library written in C++ for solving PDEs on block structured grids using finite elements. UGBlock supports the calculation of 27-point stencils from the local stiffness matrices representing the differential operators in the finite element form.

[3] http://www10.informatik.uni-erlangen.de/~pflaum/expde/Expde_Homepage.html

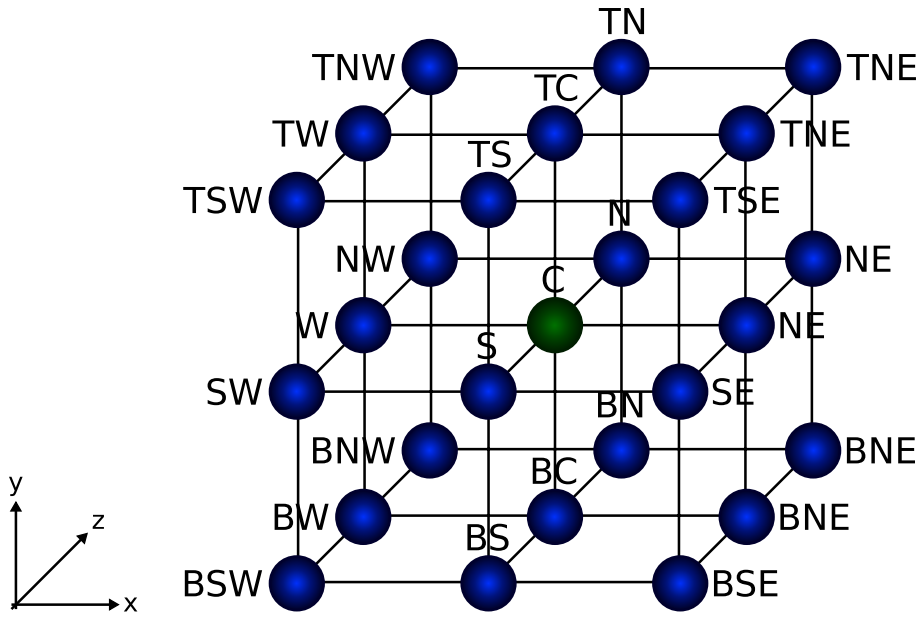


Figure 4: 27-point stencil for 3D case

UGBlock supports three different types of stencil: constant, semi-constant and variable stencils. A constant stencil is a single stencil, common to all points in the discretization domain. Semi-constant stencils are used if the discretization grid is equally spaced just in one dimension (e.g. a cylinder). For even more complex geometries or if a variable coefficient is part of the PDE, variable stencils are used, which include a different stencil for every point in the domain. Semi-constant stencils require more memory than constant stencils and variable stencils with one stencil per grid point need 27 times the amount of memory of the actual unknown vector. As both common CPU and GPU architectures make heavily use of memory caching, solvers using the simpler (and thus smaller) stencils achieve a significant higher performance as the stencil may be kept in a cache close to the CPU.

UGBlock was used as a test case during the development of GPUsolve as it offers a simple way to generate finite element operators as 27-point stencils. Furthermore the possibility to integrate a easy to use GPU accelerated iterative solver into existing code should be shown.

4.3 8-color Gauss-Seidel

4.3.1 Gauss-Seidel Method

The Gauss-Seidel method is one of the simplest and most popular iterative solvers for linear systems of equations. It is similar to the Jacobi method but it typically converges faster.

The system to solve is given as:

$$\mathbf{Au} = \mathbf{f}, \qquad \mathbf{A} \in \mathbb{R}^{n \times n}, \quad \mathbf{u}, \mathbf{f} \in \mathbb{R}^{n}.$$

The matrix $\mathbf{A}$ multiplied by the unknown vector $\mathbf{u}$ results in the so called right-hand side vector $\mathbf{f}$. It is important to note this matrix $\mathbf{A}$ often isn't explicitly implemented as a matrix in iterative solvers, but rather the previously described stencils are applied to $\mathbf{u}$ resulting in much better performance.

Using the matrix decomposition

$$\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U} = \begin{pmatrix} \ddots & & -\mathbf{U} \\ & \mathbf{D} & \\ -\mathbf{L} & & \ddots \end{pmatrix}, \tag{1}$$

where $\mathbf{D}$ denotes a diagonal matrix composed of the diagonal elements of $\mathbf{A}$, $\mathbf{L}$ the strict lower and $\mathbf{U}$ the strict upper triangular matrices of $\mathbf{A}$, the the Gauss-Seidel method can be written as:

$$\mathbf{u}^{(k+1)} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{u}^{(k)} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{f}. \tag{2}$$

Here $\mathbf{u}^{(k)}$ represents the approximate solution after k iterations, $\mathbf{u}^{(k+1)}$ analogical the improved new approximation.

For implementing the Gauss-Seidel method in software it is helpful to look at a different formulation. By component the algorithm can be written as:

$$u_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j=1}^{i-1} a_{ij} u_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} u_j^{(k)} \right), \qquad i = 1, 2, \ldots, n; \quad k = 0, 1, 2, \ldots; \tag{3}$$

From equation 3 the difference between the Jacobi and the Gauss-Seidel method is obvious. When computing the new approximation for the i-th component of the unknown vector $\mathbf{u}$, the already updated values from the same iteration $(k+1)$ are used for all elements $j < i$. The remaining elements with $j > i$ use the old value from the previous iteration k. Because of this it is not necessary to have two separated vectors with the old and new approximations for $\mathbf{u}$ and switch them after each iteration. Instead a single vector can be used and the updates are performed in place. Using this technique one does not even have to distinguish the cases $j < i$ and $j > i$, as the vector is the same.

4.3.2 Parallelization of the Gauss-Seidel Method

In the previous section the basic Gauss-Seidel method was described, but one important characteristic has been ignored. The algorithm presented in equation 3 can not be parallelized directly, because the update of the i-th component depends on the already updated components $\mathbf{u}_j^{(k+1)}$ with $j < i$. A way to overcome this flaw is to use coloring schemes to divide the system into independent parts. This approach can get almost arbitrary complex for dense matrices, but sparse matrices with specific structures as they arise from finite difference or finite element discretization of partial differential equations greatly simplify the process. As the presented implementation works only with constant, semi-constant and variable stencils, the system matrix is never assembled explicitly.

From here on we assume that the stencils are limited to the center point and the direct neighbors. This is also true for the stencils generated by UGBlock as described earlier. The number of colors needed for a parallel implementation of a stencil-based Gauss-Seidel method can be deducted from the simple 1D case. Here the update of each point using a three point stencil depends on the center point and the ones to the left and right (figure 5a). By using only two colors (traditionally red and black) applied alternately, one iteration over all grid points can be divided into two iterations over the red and black points respectively. Each of this half iterations can be performed in parallel, as only points of the respective other color are used (figure 5b shows the black half of a iteration).

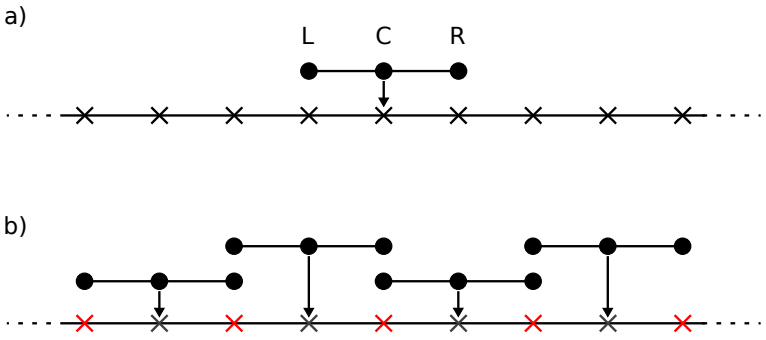


Figure 5: Red-black coloring for parallel Gauss-Seidel in 1D

In 2D the general case are 9-point stencils, which require two times as many colors as the one dimensional problem (figure 6). The often used 5-point stencil presented above permits to use a combination between the 1D and 2D case. Here it is sufficient to use only two colors resulting in the well known checkerboard pattern. The often used Red-Black Gauss-Seidel is a modification of the standard Gauss-Seidel method for just this special case.

From two to three dimensions four additional colors are needed resulting in a total number of eight colors. To keep things simple, GPUsolve uses only two colors (red and black) and four simple symbols as shown in figure 7 to describe the 8 groups of points. This is only a special notation, as it is easier to handle two groups of alternating colored symbols than eight different colors. Later on it will become clear why it is reasonable to think of the 3D case as multiple 2D layers put together.

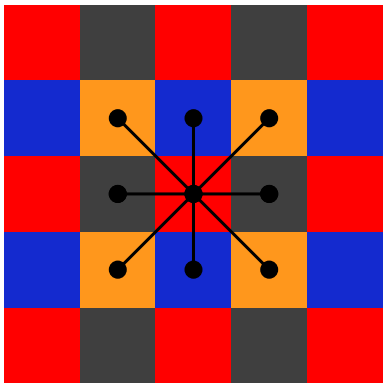


Figure 6: Scheme with four colors for parallel Gauss-Seidel in 2D

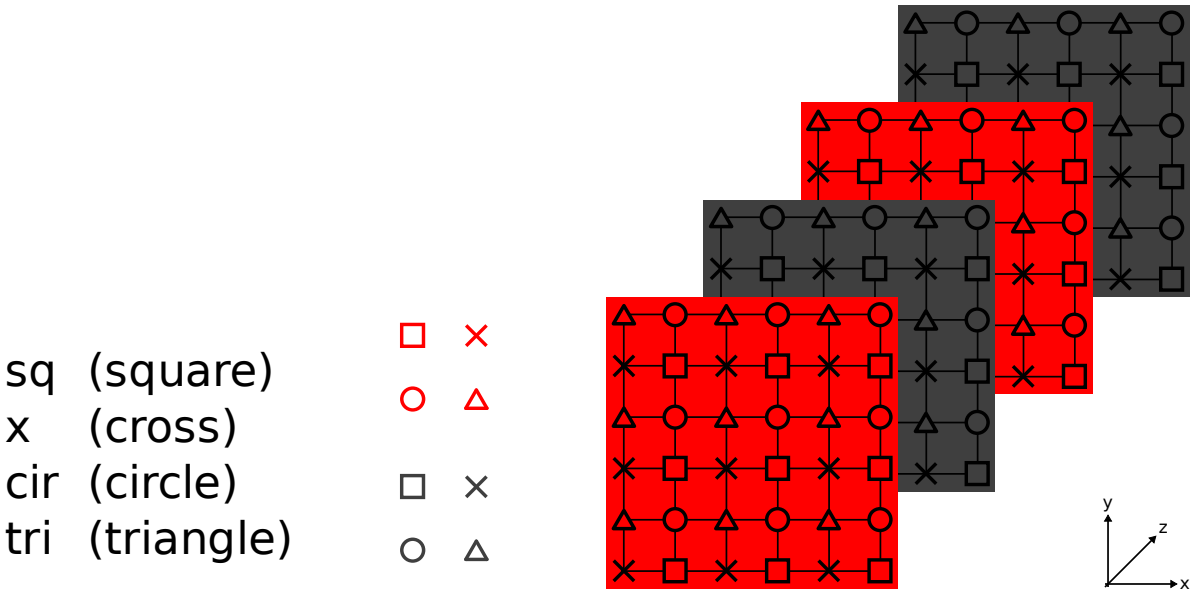


Figure 7: The 8 colors/symbols used in the multi-color Gauss-Seidel for 3D

5 Optimization Possibilities for CUDA

Prior to the discussion of the implementation of the needed infrastructure and the parallel Gauss-Seidel solver, a closer look at the GPU hardware characteristics is necessary to understand the optimization potential and some of the design decisions later on.

5.1 Memory Access

Although recent-high end graphics cards provide a memory bandwidth of up to 104 GB/s as in the case of the NVIDIA GeForce 8800 Ultra, only very specific access patterns to global memory deliver this potential to your application.
The main difficulty when learning GPU programming using CUDA is the different memory hierarchy. For current CPUs one always has a 3-level memory hierarchy consisting of CPU internal registers, caches (L1, L2, potentially L3) and global memory (RAM). Data requested by the CPU is automatically loaded to the caches according to some sophisticated algorithms depending on the exact layout and internal organization of the various caches. So by only reading or writing memory locations in a program the user benefits of the caching effects. Normally it is possible for a programmer to improve the use of caches using methods like blocking or loop fusion. So only in very special cases it is necessary to really manipulate the way data is loaded to the caches by performing software prefetching or similar optimizations. These interferences into the memory hierarchy are all but trivial and require a solid understanding of the specific hardware (CPU and memory). Because of this and the fact that optimizations of this kind are not portable even between CPUs of one generation, one seldom uses them.

When using CUDA the picture is completely different. As described earlier, the global memory on NVIDIA cards is not cached. So the middle layer in our memory hierarchy seems to be missing on the first glance. On the top we have the local registers in the various streaming multiprocessors and on the low end the device memory connected via a powerful 384 bit wide link running at clock rates of up to 1.5 GHz. When looking at the specifications of this memory connection, one might wonder why caches are at all needed, when the whole memory is accessible with a bandwidth an order of magnitude greater than on common CPUs.

The flaw lies in the way memory accesses are handled by the hardware both in times of latency and bandwidth. Each read or write access to device memory has a extremely high latency. NVIDIA's documents mention numbers between 400 and 600 clock cycles per memory operation. Although this can be hidden at least partly by running many (in the region of several hundred) threads, one problem remains: bandwidth. The impressive memory bandwidth can only be utilized when several threads each access consecutive regions in device memory whose start addresses additionally have to be aligned in some form. The important number here is the size of the so called *half-warp*, which is the first or second part of a warp as it was described in chapter 3.1. On current hardware a half-warp consists of 16 threads, but this may change in future hardware implementations. The memory accesses conform to these limitations are referred to as being *coalesced* in NVIDIA's jargon. Coalesced memory operations also reduce the latency dramatically: Only the warp's first thread's access takes the mentioned high latency, the remaining threads experience only a latency of one clock cycle.

In order to perform efficient, coalesced memory accesses several conditions have to be fulfilled:

- All threads of a half-warp access equal data types.
- All threads of a half-warp access consecutive memory addresses.
- The start address of the accessed region is aligned to 16 times the size of the accessed data type.

These various requirements along with typical memory access patterns are further detailed in Harris [2007]. It has to be noted however, that the newest generation of CUDA capable hardware (starting with Compute Capability 1.2) have far less restrictions for efficient memory accesses. As the cards used in the development for this work are only of Compute Capability 1.0/1.1, they are objected to the access restrictions presented above.

5.2 Usage of Shared Memory

As described in the previous section, efficient access to global memory is not trivial. A wide range of parallel algorithms including solvers for systems of linear equations at some point access the same input data multiple times, so caching these values in fast, on-chip memory may result in a significant performance increase. In the development of CUDA enabled programs it is often desirable to implement the missing cache for the global memory using the shared memory. By using this memory as a software managed cache it is possible to keep the global memory accesses coalesced and have the single threads running on one SM performing almost arbitrary complex access patterns on the cached data. However one has to take care that the threads don't produce too many bank conflicts. It is important to evaluate the different possibilities for each implemented algorithm, as the improvement in memory access time and bandwidth when using shared memory may produce a large amount of bank conflicts and thus slow down the application in the end.

5.3 Streaming Multiprocessor Resources

According to the specifications, the streaming multiprocessors of the Tesla architecture seem to provide a generous amount of hardware resources like registers, shared memory and support for concurrently running threads. Especially the number of registers seems to be high above all needs, also 16 KB of shared memory may seem sufficient. The point to remember is the high number of threads concurrently running on one SM. To fully utilize the hardware, NVIDIA recommends to run at least two thread blocks of 128, better 256 threads per SM. This leads to a number of 256 or 512 threads per SM.
Table 2 presents some numbers of the most important resources in the SM (the number of registers and shared memory available to each thread) for different numbers of threads running simultaneously. By looking at these figures it becomes quite clear some effort has to be put into minimizing the amount of shared memory and registers used per thread. The resources used by a specific kernel can be found in the corresponding .cubin file, generated by the nvcc compiler using the -cubin flag. The important numbers are in the *code* sections labeled *smem* and *reg*.

Resource	Number of threads			
	1	256	512	786
Registers	8196	32	16	10
Bytes of Shared Memory	16384	64	32	21

Table 2: Distribution of a SM's hardware resources according to the number of threads

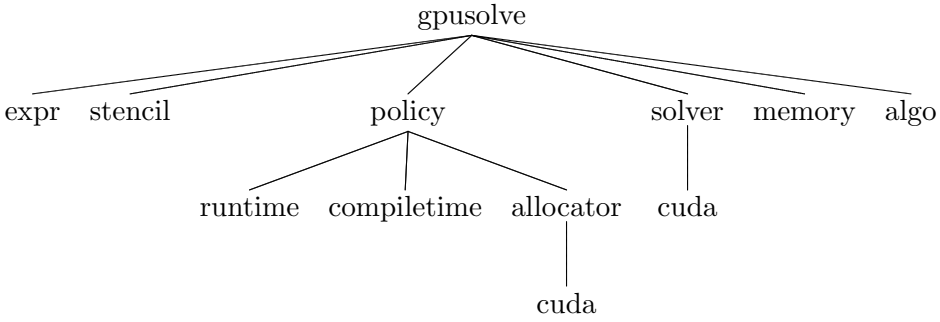
The CUDA Occupancy Calculator offers a nice way to estimate the number of threads that will be able to run at the same time. Using the CUDA Visual Profiler it is possible to validate this estimates on the real hardware and also find information about the various types of memory accesses performed by the kernels. Using these tools it is possible to minimize the resource usage of CUDA kernels quite well.

6 Implementation Details

6.1 Code Organization

GPUsolve makes use of the namespace support in C++. All components are contained in the `gpusolve` namespace. In this namespace the various parts are grouped in further namespaces according to their functionality. The naming of the classes in the single namespaces follows a common scheme: First a (preferable) short name describing the function of the class, followed by a additional sequence with the dimension (vectors, stencils, solvers) and the number of stencil points (stencils and solvers) and possibly a descriptor for the type of stencil: constant → C, semi-constant → SC, variable → V. This convention was inspired by the typical naming of Lattice Boltzmann methods. The variable stencil implementation for example is contained in a class called StencilD3Q27V. At the moment only D3 and Q27 are used, as only three dimensional problems with 27-point stencils are supported.

The most important namespaces in the implementation are:



In `gpusolve::expr` the classes with implementations of three dimensional vectors (VectorD3) both for the host and device side are contained. The namespace is named *expr*, as some simple expression templates (see section 6.2.3) are included. The `gpusolve::stencil` namespace holds the implementations of constant, semi-constant and variable (StencilD3Q27V) stencils for both CPU and GPU. The different policies (section 6.2.4) controlling special features of the code are in `gpusolve::policy` and the subordinated namespaces. The actual 8-color Gauss-Seidel implementations can be found in `gpusolve::solver`, the implementations using CUDA accordingly in `gpusolve::solver::cuda`. In the `gpusolve::memory` and `gpusolve::algo` namespaces all classes and templated functions to handle memory management and copy operations between the host and device side are arranged.

6.2 Infrastructure

6.2.1 Memory

In order to get data from the host system to the GPU and back, the global memory region on the graphics card has to be used. This memory can be accessed from both the CPU and the GPU for both reading and writing. It is managed like normal memory on the host through specific calls in the CUDA runtime API. Calls for allocating and freeing of memory portions as well as to transfer data between the host and the graphics card are available. This calls use the NVIDIA kernel driver to handle the hardware specific parts like using DMA buffers and communicating with the graphics card. The functions from the memory management API are very similar to malloc, free and memcpy as provided by the C standard library. As GPUsolve is implemented in C++, some wrapping had to be done for this calls.

The (de)allocation of memory should be possible by using the standard new and delete operators. Only two potential ways to overload these operators exist: Either globally or on a per class basis. The first approach is of course not feasible, as memory allocation on the CPU side should still be possible using the standard implementation. So the operators have to be specific to a certain class/struct. For this purpose a wrapper struct only containing one element of the type specified via a template parameter was created. Using a second template parameter, it is possible to select the allocator to use for instances of this type. The allocator is a simple class only consisting of the new and delete operators as (static) member functions. The implementation of the struct `memory::DeviceMemory` as well as a sample allocator from `policy::allocator::cuda::CUDADeviceAllocator` are detailed in listing 1.

Listing 1: Wrapper class and sample allocator for memory management on GPU

```
template<typename Data_Type, class Allocator>
struct DeviceMemory : public Allocator {
    Data_Type element;
};

struct CUDADeviceAllocator {
    static void* operator new(std::size_t size) throw (std::bad_alloc) {
        // allocate memory on GPU using runtime API
        [...]
    }

    static void* operator new[](std::size_t size) throw (std::bad_alloc) {
        // allocate memory on GPU using runtime API
        [...]
    }

    static void operator delete(void* p) throw() {
        // free memory on GPU using runtime API
        [...]
    }

    static void operator delete[](void* p) throw() {
        // free memory on GPU using runtime API
        [...]
    }
};
```

A short example on how to use the `memory::DeviceMemory` wrapper is shown in listing 2. All memory allocation in GPUsolve is done using this wrapper, so runtime error checks only have to be done in one place: the new and delete operators in the corresponding allocator. In the case of errors, just the same exceptions are thrown as by the standard operators, so there is no need for special treatment of GPU memory. As the memory on the graphics card is wrapped in `memory::DeviceMemory<Data_Type, Allocator>`, it is not possible to assign a pointer to this memory to a pointer of type Data_Type and use it like host memory by accident. So some type safety is added by this approach as well.

The NVIDIA device drivers use DMA (Direct Memory Access) for data transfers to and from the devices, instead of Programmed IO (PIO). This results in a great performance boost, as the actual copy operations are performed by the hardware using a DMA controller instead of the main CPU. DMA only works for memory regions whose pages have been locked (or *pinned*) to prevent the operating system from swapping. The CUDA API offers special functions (see cudaMallocHost(), cudaFreeHost()) to allocate and free memory and perform the necessary steps for locking the memory pages automatically. As some

benchmarks performed to shed some light on the performance improvement of using this special functions revealed a significant boost in transfer speed (section 7.1), support for memory management using the CUDA specific functions was included. A wrapper similar to `memory::DeviceMemory` named `memory::HostMemory` was implemented, along with two allocators: the `policy::allocator::cuda::CUDAHostAllocator` and the default counterpart `policy::allocator::StandardAllocator`, which uses the standard new and delete operators. When CUDA's functions for memory copies are called on memory allocated with the default new operator, for example from StandardAllocator, a fallback is performed: the API allocates an internal buffer and locks this memory and copies the data from or to main memory piecewise using the CPU and between the buffer and the graphics card using DMA.

Listing 2: Example for memory allocation on GPU using wrapper memory::DeviceMemory

```
1  using namespace gpusolve::memory;
2
3  // select a appropriate allocator
4  typedef policy::allocator::CUDADeviceAllocator Allocator;
5
6  // allocate array of 10 floats on GPU
7  DeviceMemory<float, Allocator>* data =
8      new DeviceMemory<float, Allocator>[10];
9
10 // ...and free it again
11 delete[] data;
```

6.2.2 STL-like Functions

The memory management is only one part, though. In order to move and initialize data in memory, the C++ STL provides some very nice functions like `std::copy` and `std::fill` from <algorithm>. But this functions are restricted to direct accessable memory. Memory allocated on the graphics card can only be manipulated using the CUDA runtime API. This C API is not very comfortable to use, especially when having to check for errors after each call. So the idea was to create wrappers around this calls to be able to manipulate the GPU memory though a STL like interface. Also the `HostMemory` structure used to describe memory allocated on the host side by special allocators requires such wrappers.

The first algorithm needed was the equivalent to `std::copy`, whose interface is specified in listing 3; it copies the elements in the range first to last (last excluded) to the destination described by result.

Listing 3: Prototype for std::copy

```
1  template<class InputIterator, class OutputIterator>
2  OutputIterator copy(InputIterator first, InputIterator last, OutputIterator
       result);
```

The STL functions are used for all host side copying between instances of `HostMemory`. For the GPU side three additional types of data transfers are needed: host-to-device, device-to-host and device-to-device. The first two are used to move data to and from the graphics card, the last is needed to copy data on the GPU, without transferring back to main memory in between. Three templated functions where implemented, using the previously described `memory::DeviceMemory` wrapper to identify pointers to GPU memory. A sample implementation of the copy algorithm for the device-to-host case is shown in listing 4. For all combinations of memory copies between host (`memory::HostMemory`) and device memory (`memory::DeviceMemory`) corresponding functions were added.

Aside from the `std::copy` imitation, also counterparts for `std::fill` and `std::fill_n` were implemented.

Listing 4: Sample implementation of gpusolve::copy for device-to-host transfers

```
using namespace gpusolve::memory;

template<typename T, class Device_Allocator, class Host_Allocator>
DeviceMemory<T, Device_Allocator>* copy(HostMemory<T, Host_Allocator>* first,
     HostMemory<T, Host_Allocator>* last, DeviceMemory<T, Device_Allocator>*
    result) {

    size_t size = last - first;

    if(size) {
        cudaError_t success = cudaMemcpy(result, first, size * sizeof(T),
            cudaMemcpyHostToDevice);

        if(success != cudaSuccess) {
            [...]
        }
    }

    return result;
}
```

6.2.3 Data Structures

The most important classes for data storage in GPUsolve are `VectorD3` and the stencil implementations `StencilD3Q27C` (constant stencils), `StencilD3Q27SC` (semi-constant stencils) and `StencilD3Q27V` (variable stencils). All of them make heavy use of the C++ template mechanism. Template parameters are used to differ between instances in host memory and GPU memory, to support arbitrary data types and policies (section 6.2.4).

Vectors `VectorD3` represents a 3 dimensional array used for any variable in the context of GPUsolve. Specialized constructors and copy operators were implemented so copying values between the host memory and the device memory can be done by simple assignments. For the CPU side a number of simple expression templates [Härdtlein, 2007] were implemented. At the moment only basic vector operations like addition, subtraction or dot-product are implemented. All `VectorD3` specializations are derived from the `Expr` class, which is a wrapper around expressions composed mainly of instances of VectorD3.

The memory layout of the `VectorD3` implementation differs between the CPU and GPU versions. On the CPU side, the data is stored in a straight forward linearized one dimensional array. As the Tesla architecture requires memory accesses to be aligned, the memory layout had to be altered slightly. The three dimensional vector is also linearized, but the size in the x-direction is padded to multiples of 16 to keep all memory accesses coalesced. However a optimized CPU version is available via a compile-time policy. This version also uses padding to achieve the same memory layout as the GPU counterpart, resulting in easier and faster copy operations.

Stencils The stencil classes are used to store the 27-point stencils representing the finite element operators. Although the classes are named `StencilD3Q27`, they actually store multiple stencils, at least the semi-constant and variable stencil versions. They offer a object oriented interface, similar to a vector, with an additional parameter to select one of the 27 points.

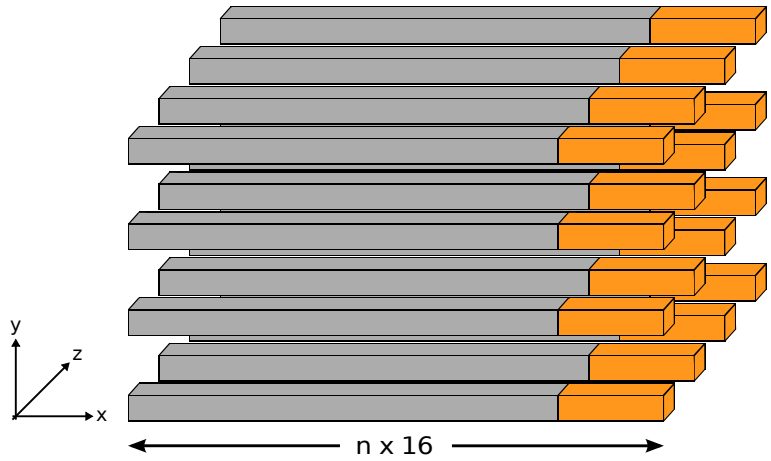


Figure 8: Padding in x-direction in GPU version of VectorD3

Like the `VectorD3` implementations various assignment operators were created to simplify the task of assembling and copying stencils. Only the stencil classes for the host side allow direct access to the individual entries, as it is not possible to access single values in the graphics card's memory efficiently. So the stencils are to be composed on the host side and transferred to the GPU by just performing assignments. The operators take care of transforming the stencils from the CPU side to the highly optimized memory layout presented in section 6.3.2.

Template Parameters As these basic classes have quite a number of template parameters, a problem arose: any code which uses these classes also has to handle the template parameters. This could easily lead to functions with 5 to 10 template parameters, which of course is not manageable any more. To overcome this, for each of the vector and stencil classes two simpler interfaces were created. For `VectorD3` these classes are called `VectorD3Base` and `VectorD3Type`. Most parts of GPUsolve including the actual Gauss-Seidel solver only use these types. `VectorD3Base` offers a simple interface to some member functions useful in the context of a multidimensional array, like the size of the single dimensions. As the virtual functions of these base classes may hinder optimal performance, only non-critical member functions are contained in `VectorD3Base`. When full access to the `VectorD3` members is needed, `VectorD3Type` is the optimal interface. Using the type information placed in it via a template parameter by `VectorD3`, it is possible to determine the exact type of the `VectorD3` including all policies. By performing a dynamic cast to the type determined from `VectorD3Type` it is possible to restore all type information from the more general object and use it as a `VectorD3`. The data type and device (CPU on host side or GPU) always have to be specified, so it is possible to determine the most important type information at compile time and use the optimal code segments for each object. For the stencil classes analog interfaces like `VectorD3Base` and `VectorD3Type` exist. The basic relationships between the mentioned classes are depicted in figure 9.

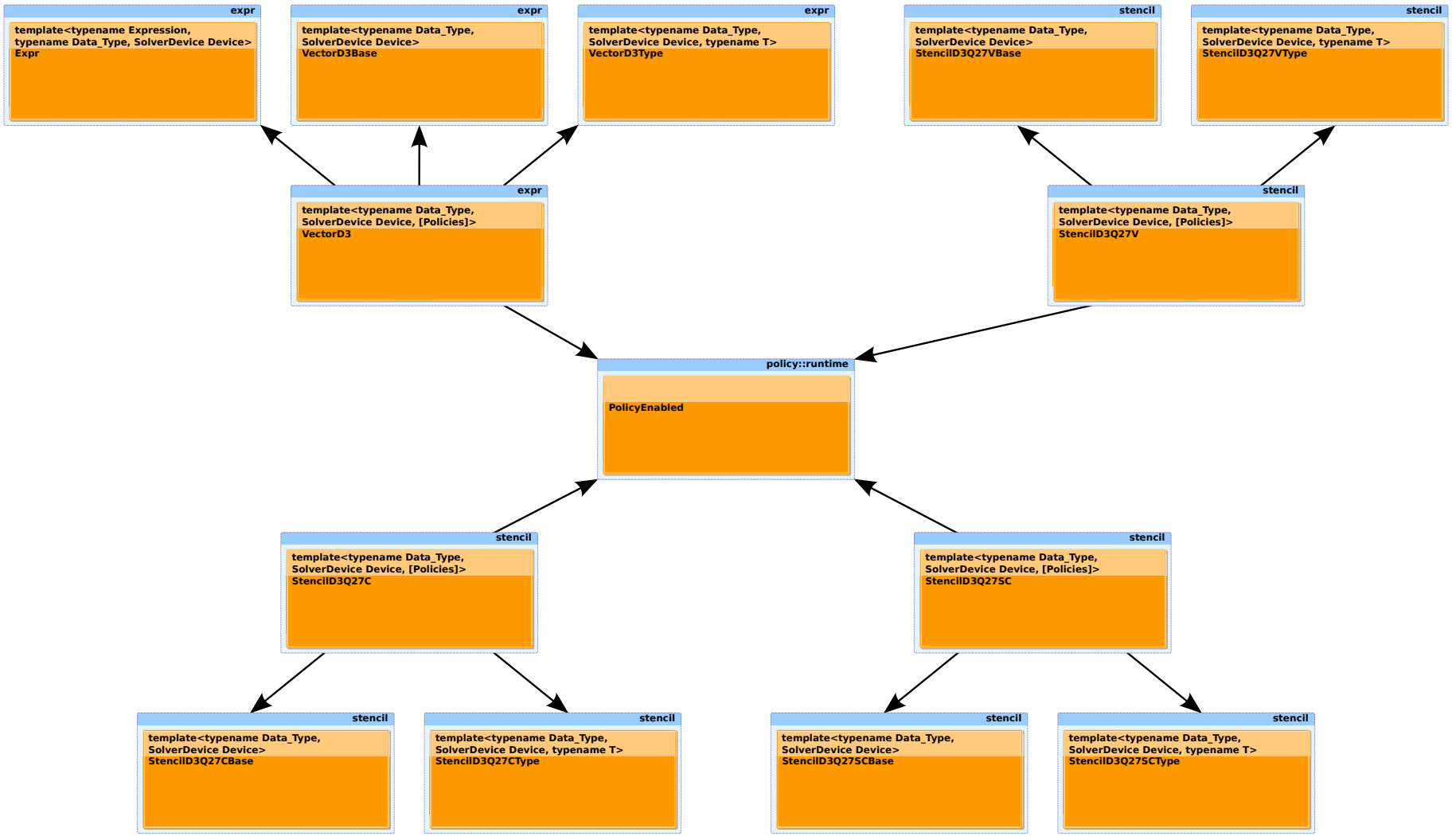


Figure 9: Basic vector and stencil classes in GPUsolve

6.2.4 Policies

To support high performance as well as debugging and flexibility in terms of basic object properties the concept of policies was integrated into GPUsolve. Basics on policies in the context of programming using C++ templates are described in Vandevoorde and Josuttis [2003]. The goal is to be able to specify exactly for which parts of the code checks for example of array boundaries or timings of memory copy operations are to be performed. For the CPU version of VectorD3 it is possible to activate a specialized memory layout using more memory but offering a huge performance boost for copy operations to or from the graphics card. Another option is the selection of different memory allocation methods as mentioned earlier. As far as possible the policies describing this behavior should be not static, but changeable at runtime. For certain policies like array boundary checks it is not possible to support this efficiently. Likewise it doesn't make sense to change the memory allocator during an objects lifetime. Due to this two classes of policies were created: compile-time and runtime policies. Compile-time policies make use of the C++ template mechanisms and are evaluated by the compiler imposing no overhead to the program at runtime. The compile-time options are implemented using the concept of *named template parameters* [Vandevoorde and Josuttis, 2003, chap. 16.1] so the user does not have to take care of details like the correct order of the parameters or specifying additional default parameters. The runtime policies can be changed during the execution of the program, although the default values are configured at compile time.

Currently three template parameters are used to specify compile-time policies. Two are used to select a memory allocator for memory on the host and the GPU side. It should not be necessary to change the defaults for these two options, but it is of course possible for certain applications. The third parameter actually specifies a class containing constants which control specific performance and debugging related features. Currently flags controlling boundary checks of array accesses and the use of the special memory layout for optimized copies are implemented. The class specified here is the actual compile-time policy and could be extended to include more options at any time. Using this policy it is possible to configure each instance of an object individually. This is a great enhancement in contrast to the often used preprocessor functionality of simple `#define`s which of course affects all objects of a type. Using the policy enabled classes like `VectorD3` and predefined or user generated policy classes it is possible to create both fast and debug-friendly code. As all options of the selected compile-time policy are known to the compiler during the optimization steps, disabled safety checks can be removed resulting in higher performance at runtime. Listing 5 shows the construction of three vectors making use of different allocators and policies.

Listing 5: Compile-time policies in GPUsolve

```
1  using namespace gpusolve::policy;
2
3  // create a vector in main memory using a special allocator
4  Vector3D<float, device::cpu,
5      HostAllocator_is<allocator::cuda::CUDAHostAllocator> > u(10, 10, 10);
6
7  // enable debug functions like boundary checks for array accesses
8  Vector3D<float, device::cpu,
9      CompiletimePolicy_is<compiletime::DebugPolicy> > v(10, 10, 10);
10
11 // create vector in graphics memory using defaults
12 Vector3D<float, gpu> w(10, 10, 10);
```

Runtime policies are composed of two parts:

- PerformancePolicy
- InfoPolicy

These policies enable or disable various optional features in the code and also allow the user to specify a target to which generated log messages are written. The `PerformancePolicy` controls for example whether kernel calls and memory copies of vectors and stencils are timed. Furthermore statistics like calculations of the reached number of Flops in certain kernels can be activated. The `InfoPolicy` mainly serves to identify bottlenecks and unneeded instantiation and coping of objects. By using it, additional information on the life time of objects is made available, as well as low-level details about memory usage and alignment. All groups of messages can be activated independently via special flags. A quite self explaining example is presented in listing 6.

Listing 6: Sample runtime policy in GPUsolve

```
1  using namespace gpusolve::policy::runtime;
2
3  DefaultPolicy& policy = DefaultPolicy::Instance();
4
5  policy.GetPerformancePolicy().SetLogTarget("perf.log", std::ostream::trunc);
6
7  policy.Enable(PerformancePolicy::all);
8  policy.Disable(PerformancePolicy::time_kernel_call);
9
10 policy.Enable(InfoPolicy::all);
11 policy.Disable(InfoPolicy::mem_alignment);
```

6.3 8-color Gauss-Seidel

6.3.1 Blocking

The idea of using the 8-color Gauss-Seidel algorithm was to be able to perform computations in parallel. To do this our 3D grid has to be broken down into small blocks each of which can be processed at the same time. To identify the exact layout of these blocks, it is necessary to take a closer look at the data dependencies.

When analyzing the update of one grid point one sees that quite a lot of data is needed:

- the adjacent points from **u** for all 3 dimensions (26 values)
- value of the right-hand side vector **f** (1 value)
- the stencil to apply to this point (27 values)

So a total of 54 values each of size 4 bytes (for single precision) resulting in 216 bytes have to be accessed to perform the computation for one single updated point.

As figure 10 shows, all of the values of **u** which are needed to perform this update, are also needed when updating the neighboring points of the same color. So these values should of course be held in some sort of cache. The value of the right-hand side is only needed for this single point and it is constant and as such never altered by any thread during the whole iteration. The stencil is also constant, but may change from point to point, depending on the type of stencil used (constant, semi-constant or variable). In the most basic case, where the stencil is constant over the whole grid, all threads working in a block use the same stencil and it would be a good idea to cache it.

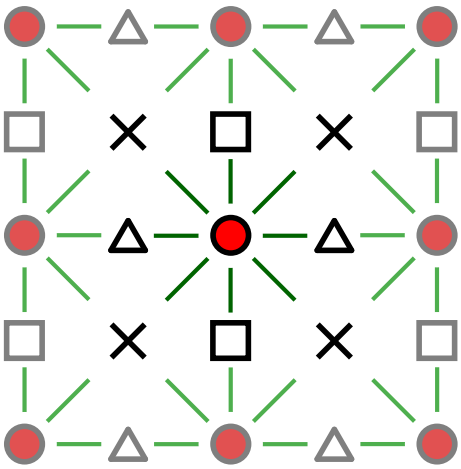


Figure 10: Update of single layer with corresponding stencil components

The Tesla architecture offers two different approaches for caching data from global memory. On the one side, it is possible to implement ones own caching algorithm using the shared memory, or on the other hand treat the data as a texture and rely on the texture cache implemented in hardware. But textures are read-only as they are a part of the traditional graphics programming concept. This leaves us with only one use for textures: to store read-only, never changing data as the right-hand side vector $\mathbf{f}$ or the stencil. The caching of the unknown vector has to be done by software using the shared memory region in the SMs. Here another difficulty arises in the form of the limited size of the shared memory (16 KB). As it is crucial to have several thread blocks running on one SM at the same time to hide the memory latency, the remaining amount of shared memory is dramatically reduced further. When reading from global memory only coalesced reads are efficient, so at least 16 threads (a half-warp) have to read consecutive addresses. This means the cached blocks of $\mathbf{u}$ have to be at least 16 points wide in x-direction. But as only every second point in each dimension is updated during one sweep and all threads should perform the same operations as far as possible, it is far more efficient to have each thread read 2 points per dimension, so in case of 3D a total of 8 points. When each thread reads 2 points in the x-direction and two points are needed for the boundary, the block size has to be extended from 16 to 34 to keep the reads coalesced and all threads busy. In order to keep the overhead of the boundary reasonable small, the block size for the other dimensions should be at least 8, better 16, each with 2 additional points for the boundary. Clearly this is not possible as it would require to much shared memory:

$$\text{blocks of } 34 \times 10 \times 10 \Rightarrow 3400 \text{ values of 4 bytes} \approx 13.3 \text{ KB}$$

or even

$$\text{blocks of } 34 \times 18 \times 18 \Rightarrow 11016 \text{ values of 4 bytes} \approx 43.0 \text{ KB}.$$

To solve this dilemma a approach presented by Williams et al. [2006] and Brandvik and Pullan [2008] is used. Instead of whole blocks only two dimensional slices of the 3D grid are loaded into the cache (figure 11). This way the slices can be big enough to compensate the loading of the boundary points and still only use moderate amounts of memory to have multiple thread blocks run on one SM at the same time. The blocking size in x-direction is determined by the hardware and currently fixed to 32 interior points with 2 additional boundary values. In the y-direction a blocking size of $16 + 2$ points is used. The blocking size in the z-direction is arbitrary, as long as it's a multiple of 2. The number of threads used is determined by these

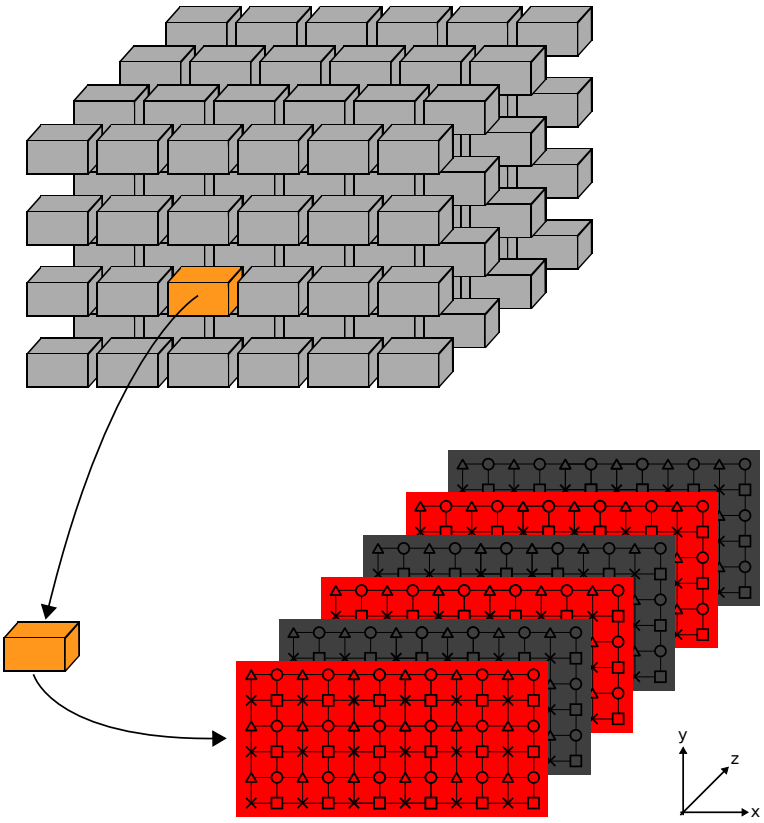


Figure 11: 2D slices in a block from the 3D grid

dimensions and is set to $16 \times 8 = 128$. This number is quite low to keep all SMs fully occupied at all times, but as the implemented algorithm is memory bound this poses no problem. The process of loading the slices of $\mathbf{u}$ into the shared memory is somewhat complex, due to the boundary values and the required alignment for coalesced memory accesses. Figure 12 shows how one 2D slice is broken down into various parts matching the number of involved threads. It has to be noted once again that the thread blocks are only two dimensional as the whole grid is processed in 2D slices. The 2D block loaded to shared memory has 34×18 points, so each thread has to perform multiple load operations. The four larger blue areas in figure 12 each are the size of a thread block (16×8), the five smaller areas mark loads only performed by some of the threads. For example the small region on the lower right-hand side is loaded by threads whose x index is smaller than 2. Also apparent from this picture is how the boundaries in x- and y-direction are loaded to achieve fully coalesced memory accesses.

To further improve the performance by reducing the number of required synchronization points, double buffering using two shared memory regions alternately is used. With this approach only one synchronization between the different threads is required after loading on slice and performing computations on a previously loaded slice of $\mathbf{u}$.

The blocking size in z-direction is not fixed and can be determined by the user or automatically based on the dimensions of the problem. First no z-blocking was done, but especially for small problems the performance suffers from too many SMs being idle due to a too low number of blocks. The impact of various block sizes is examined in section 7.2.

A minor difficulty arises from a limitation in the CUDA libraries: a grid of thread blocks can only be two dimensional. As the presented blocking scheme uses blocks in three dimensions, a mapping from 3D to 2D has to be performed.

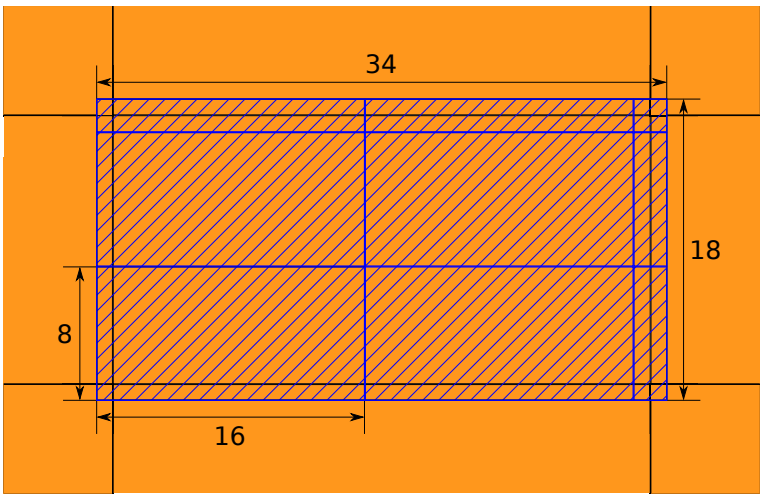


Figure 12: Loading of a 2D slice of the grid by a thread block

6.3.2 Data Layout

The data needed to perform a Gauss-Seidel iteration consists of three parts: the unknown vector $\mathbf{u}$, the right-hand side $\mathbf{f}$ and the stencil representing the matrix A. As described earlier, three versions of the algorithm were implemented to support constant, semi-constant and variable stencils. In the most basic version, only one single stencil is applied to all data points, so the greatest part of the memory is used for the vectors $\mathbf{u}$ and $\mathbf{f}$. The memory layout for these two vectors is the general one described earlier implemented in the `expr::VectorD3` template. The vector $\mathbf{f}$ is stored in this normal layout, the corresponding memory region is bound to a texture and all accesses in the kernel are carried out through the texture interface.

Constant Stencils As all threads performing updates on $\mathbf{u}$ use the same 27 point stencil, this stencil is loaded into shared memory at the beginning of each kernel call. There should not be any bank conflicts when different threads read the various stencil points as the hardware supports broadcast style memory reads. As long as all threads read the same value from shared memory, only one read instruction is needed and the result is delivered to all threads automatically. The stencil point for the central point (C in figure 4) is inverted after loading, as only the inverse value is needed (eq. 3).

The constant stencil and the right-hand side vector $\mathbf{f}$ are accessed via the texture interface, making use of the automatic caching for these constant values.

Semi-Constant Stencils The handling of semi-constant stencils is slightly more complicated. It is assumed that the stencils are constant in x-direction, i.e. all threads with the same x component in their `threadIdx` share the same stencil. The amount of shared memory is sufficient to allocate a buffer for all needed stencils. The size of this buffer is determined by the number of threads in y-direction. In global memory the stencils are padded to 32 values to keep the memory accesses aligned. The wasting of $(27 - 32) \times 4$ bytes $= 20$ bytes per stencil is justified by the increased performance due to the faster memory accesses. For the GPU version of the semi-constant stencil, the assignment operator takes care of the padding to match the optimized memory layout.

Variable Stencils When using variable stencils the main data consumer is not the unknown vector $\mathbf{u}$ any more, but the stencils for the data points itself. Concretely $\mathbf{u}$ and the right-hand

side **f** together are using two floating point values, against 27 stencil values for one single point.

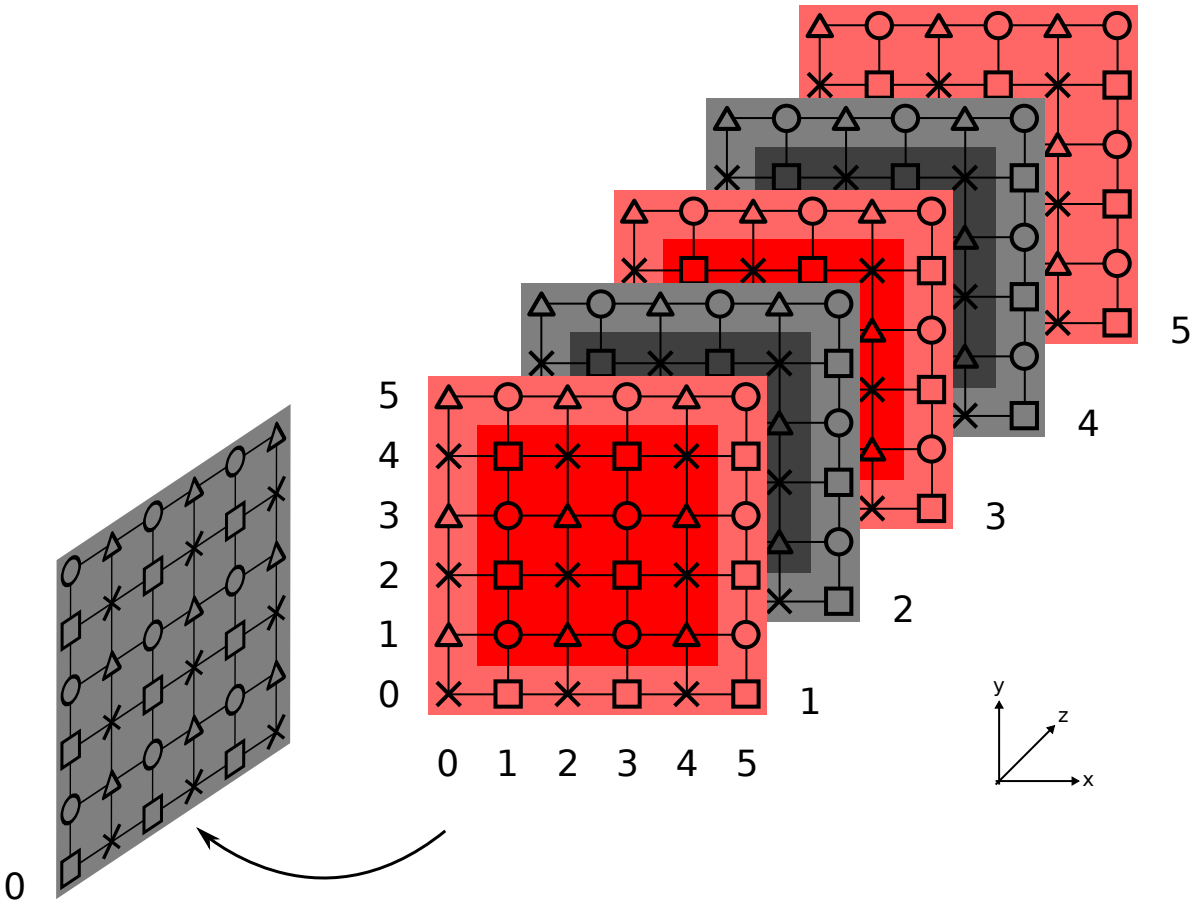


Figure 13: Model problem with 4 interior grid points per direction and boundary points (light red/gray)

The first approach was to use a texture for the stencil data, as it is read only and exhibits good data locality. Unfortunately the information release by NVIDIA about hardware details of the cache like size, associativity and overall management is almost not existing. Although multiple tests where performed it could not determined why exactly the performance suffers so much when using a texture for stencil storage. It may be that the latency of the cache was too high if each value was only requested once or the memory layout of the stencil in combination with the access pattern just did not match the cache implementation.

The remaining alternative was to use standard global memory for storing the stencils. To keep the accesses efficient for all threads an appropriate memory layout had to be found. The final storage scheme is somewhat complicated to set up, but usually the stencils are only computed and assembled once but applied in a greater number of iterations.

The stencils are separated by color, so the first thread in each block (position $(0, 0)$) accesses an aligned memory address independent of the currently updated color. Also this scheme does not include (unneeded) stencils for the boundary points, as this would break the alignment. As the stencil is applied to **u** layer by layer, each thread needs 9 stencil points to complete one layer. Due to the very limited amount of shared memory and registers it is not possible to

have each thread load these 9 values in one sweep and perform the calculations afterward. The single stencil points for the threads working point have to be loaded one by one as they are needed. As the threads should only perform coalesced memory accesses, it is necessary that the values for e.g. the bottom center point of all stencils are stored in consecutive memory addresses. The final layout is presented in figure 14.

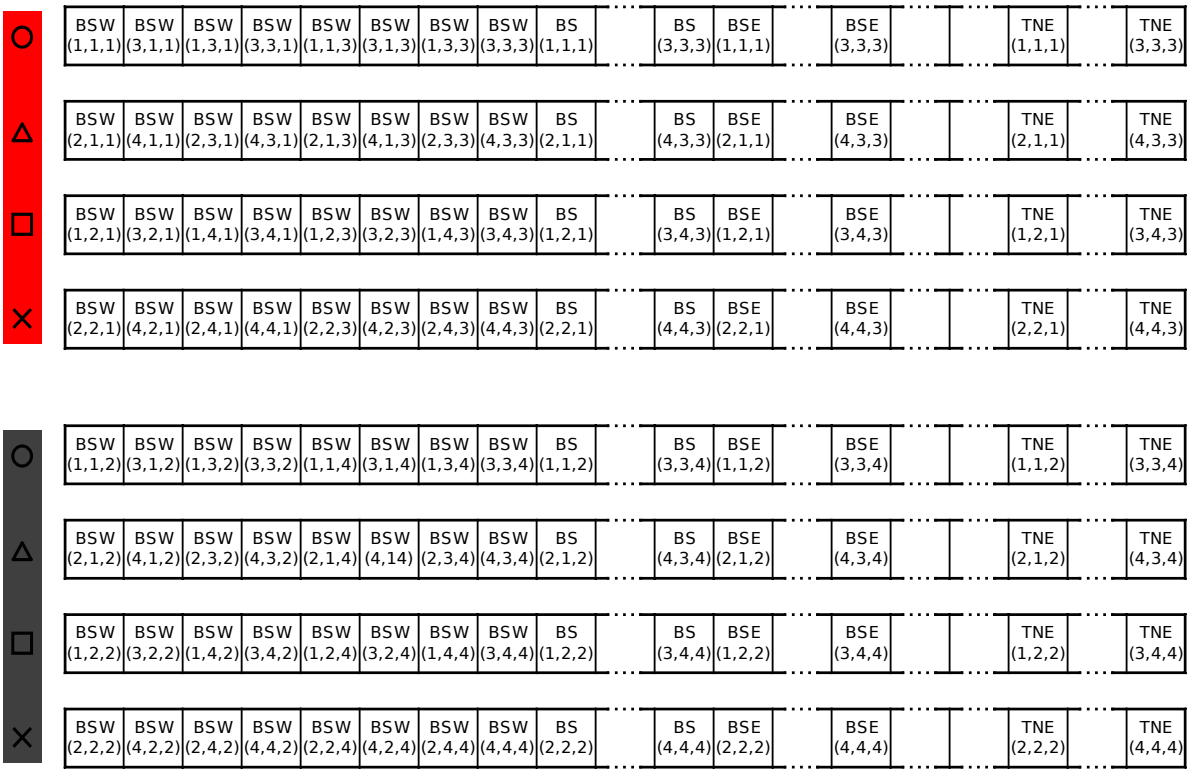


Figure 14: Storage layout for variable stencils for model problem with corresponding grid points annotated (x, y, z)

The CPU version of the variable stencil uses a traditional memory layout where all stencil points for one grid point are stored together in one block. With the offered operators a variable stencil can be set up intuitively. After that it can be converted to a GPU optimized stencil by simple assignment to a instance of the GPU version of `StencilD2Q27V`. For testing purposes a variable stencil can also be initialized from a constant or semi-constant stencil.

6.3.3 Color Handling

The color for the current sweep is supplied to all kernels via a template parameter. As the actual color is used to adjust some offsets and to prevent the loading and storing of unneeded grid points, the compiler creates eight slightly different versions of the kernels. As all this happens at compile time, no overhead is imposed due to the different colors during runtime. For some colors the number of needed registers is even reduced, as some dead branches can be eliminated by the compiler.

6.3.4 Algorithm Outline

The interactions between the various parts described in the last few sections are presented in the following sketch of the corresponding algorithm.

The most complex part of the kernel is the calculation of the memory offsets for each involved thread both in shared and global memory depending on the position of the thread in the block ($threadIdx$), the position of the block in the grid($blockIdx$) and the color of the active sweep. All indices are calculated prior to the main loop (starting at line 6), only the offsets for **u** and **f** are updated after every processed 2D slice, as they depend on the z position. The synchronization of the single threads is performed via the `__syncthreads()` call, which synchronizes all threads in a block within 4 clock cycles if no waiting is required.

The two remaining versions for semi-constant and variable stencils are very similar to algorithm 1. The only differences is the handling of the stencils. For semi-constant stencils the current stencils shared by all threads with the same x index are loaded at the beginning of the main loop. For variable stencils the offsets for all the points in the first thread's stencil in global memory are calculated in the start-up phase. Together with an index describing the current layer in z-direction it is possible for each thread to calculate the exact address of the needed stencil components. Due to the specialized memory layout presented above all the memory accesses by the threads are aligned and coalesced.

The benefit of the double buffering of the unknown vector **u** is obvious from e.g. lines 7-9. Only after both loading one layer and applying the stencil to another layer the threads in the block synchronize.

Data: unknown vector **u**, constant stencil **s**, right hand-side vector **f**
Input: block position $blockIdx$, thread position $threadIdx$, color c
Result: updated unknown vector **u**

```
1  calculate offsets;
2  allocate shared memory for u and s;
3  load s into shared memory;
4  determine zStart and zEnd;
5  load layer zStart − 1 of u into shared memory;
6  for l = zStart; l < zEnd; l = l + 2 do
7  |  load layer l of u into shared memory;
8  |  apply stencil to layer l − 1;
9  |  synchronize;
10 |  load layer l + 1 of u into shared memory;
11 |  load f for layer l;
12 |  apply stencil to layer l;
13 |  synchronize;
14 |  apply stencil to layer l + 1;
15 |  calculate updated u and store it in shared memory;
16 |  synchronize;
17 |  write back updated values of u;
18 |  update offsets;
19 end
```

Algorithm 1: CUDA kernel for a Gauss-Seidel iteration with constant stencils

6.4 Integration into UGBlock

To connect GPUsolve and UGBlock, only operators had to be added to UGBlock's vector and stencil matrix implementations. With these operators it is possible to convert between

the different representations of three dimensional vectors and stencils. After the transfer to GPUsolve the stencils generated in UGBlock representing the finite element operators are used to calculate a approximate solution for the PDE. Using these stencils it was possible to really test the internals of the various solvers and to compare the results to a completely different implementation of the Gauss-Seidel algorithm.

7 Results

7.1 Infrastructure

To evaluate the performance of the basic data structures, some tests copying various versions of `VectorD3` were performed. All memory benchmarks were executed on the GeForce 8800 Ultra in `fauia116`.

The results for transfers between instances of `VectorD3<cpu>` and `VectorD3<gpu>` are displayed in figure 15. Vectors of sizes of $500 \times 400 \times 400$ (aligned to $512 \times 400 \times 400$) were used, resulting in memory usage of up to 312.5 MB for the optimized CPU and the GPU version. On the CPU side all possible optimizations were tested: locking of the memory pages of the vector and the GPU-like memory layout. As a reference, the bandwidthTest program from the CUDA SDK was used to determine the maximal achievable transfer rate for the test system.

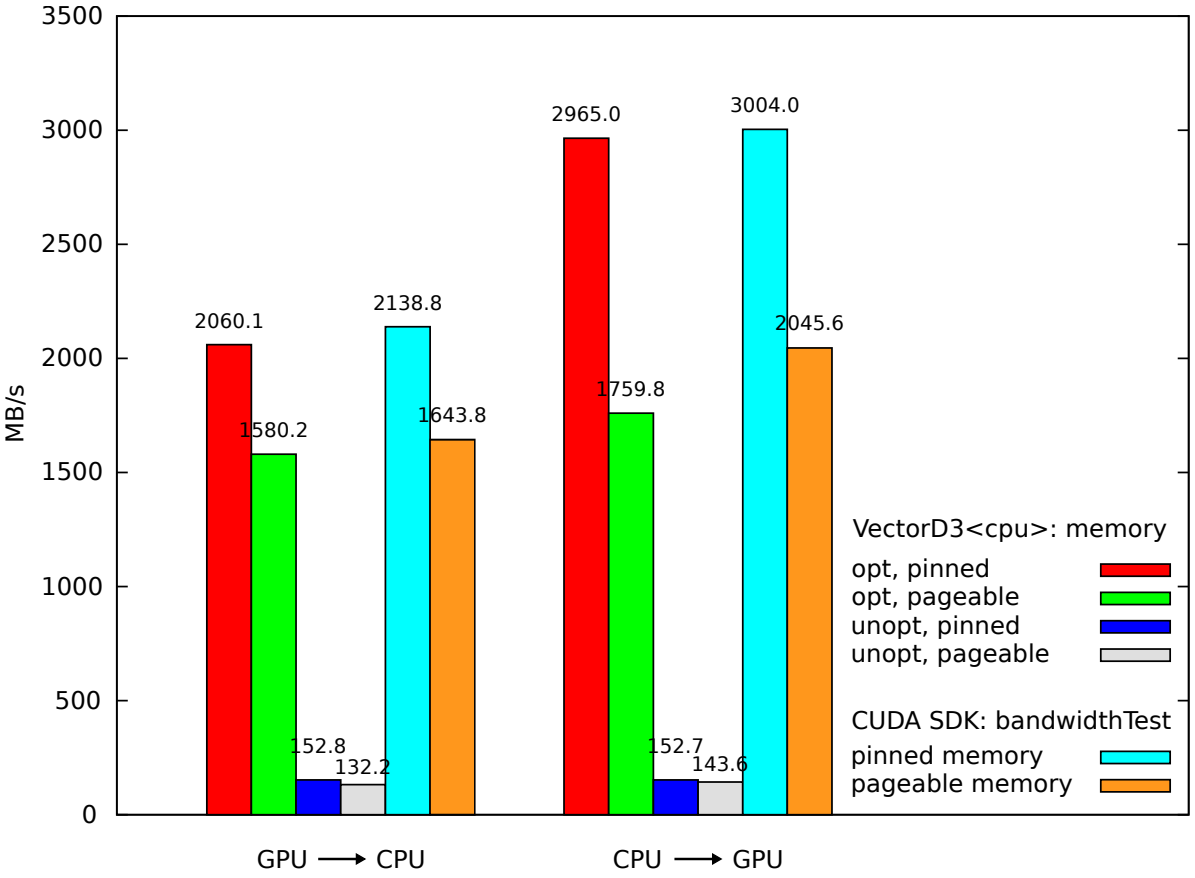


Figure 15: Memory bandwidth of device-host and host-device transfers

The results are quite clear: the fastest transfer from GPU to main memory is reached when both the optimized memory layout and page locking are activated (opt, pinned). In this case the performance of the benchmark from the CUDA SDK was almost reached. When disabling the page locking mechanism (opt, pageable), the performance drops by almost 500 MB/s which is not that bad, considering the fact that all memory transfers have to be performed on the CPU as well, as only a small page locked memory region is used for the DMA transfers. If the optimized memory layout of the CPU version of `VectorD3` is disabled, the transfer time increases dramatically. The reason for this strong bump lies in the highly increased number of system calls needed to transfer the data. For each section (determined by the size in x-

direction) of the vector a system call and a DMA transfer are necessary. So instead of one call to the CUDA API, $400 \times 400 = 160.000$ calls are performed. When both pageable memory and the not optimized memory layout are used, the transfer rate decreases further, however not as strongly as in the case of the optimized memory layout. In the reversed direction, from host to device memory, the trend is the same. Page locked memory regions can generally be copied faster, the optimized memory layout with its reduced number of system and DMA calls outperforms the simple version. Yet it can be noted, that all memory operations achieve a higher performance than in the GPU to CPU direction. This fact is obvious especially for the CUDA bandwidthTest with pinned memory. The reason for this is probably the optimization of the GPU for transfers from main to device memory, as this is the default case for typical GPU applications like 3D games.

Based on this benchmarks the optimized version of `VectorD3` performs quite well when compared to the performance of bandwidthTest from the CUDA SDK. If vector data is only copied to and from the GPU rarely (e.g. only after several 100 iterations of a solver), the unoptimized versions of `VectorD3` are sufficient. If however the data is copied frequently (e.g. after each iteration), page locked versions of `VectorD3` with the optimized memory layout should be used as the transfers may become a bottleneck otherwise. It also has to be noted that the fastest version (opt, pinned) obtains almost two times the performance of copies in main memory (about 1.5 GB/s).

7.2 8-color Gauss-Seidel

In order to analyze the performance of the implemented 8-color Gauss-Seidel algorithms, a simple test case was created. The Poisson's equation

$$\begin{aligned} -\Delta u &= f \quad \text{on } \Omega = (0,1)^3 \quad \text{with} \\ f &= 3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z) \quad \text{and} \\ u &= f \quad \text{on } \partial\Omega \end{aligned}$$

was used, so the solution could be determined analytically as

$$u = \sin(\pi x)\sin(\pi y)\sin(\pi z).$$

Using finite differences, the standard 7-point stencil shown in figure 16 is calculated. This stencil is used in all test cases, for the semi-constant and variable stencils, the same stencil was used multiple times. The remaining 20 points in the 27-point stencil were set to zero. Although the stencil is not fully filled, the performance measurements are accurate, as all stencil points had to be processed, independent of their value.

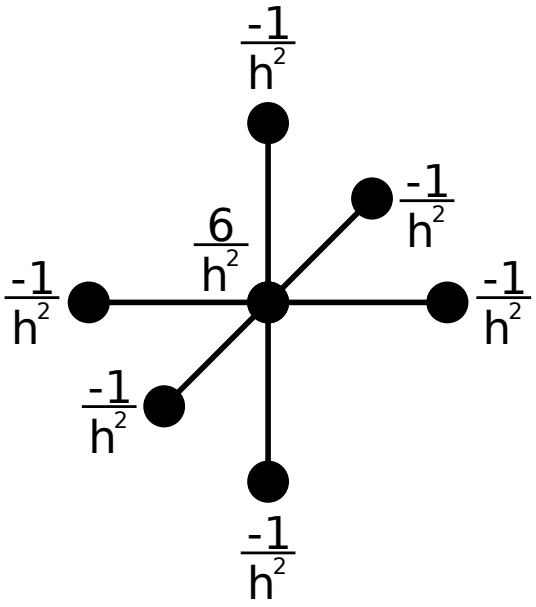


Figure 16: 7-point stencil used for testing the Gauss-Seidel implementation

The first series of tests was performed to identify the optimal value for the blocking in z-direction. To do this, all three versions of the solver were executed various domain sizes on both available graphic cards.

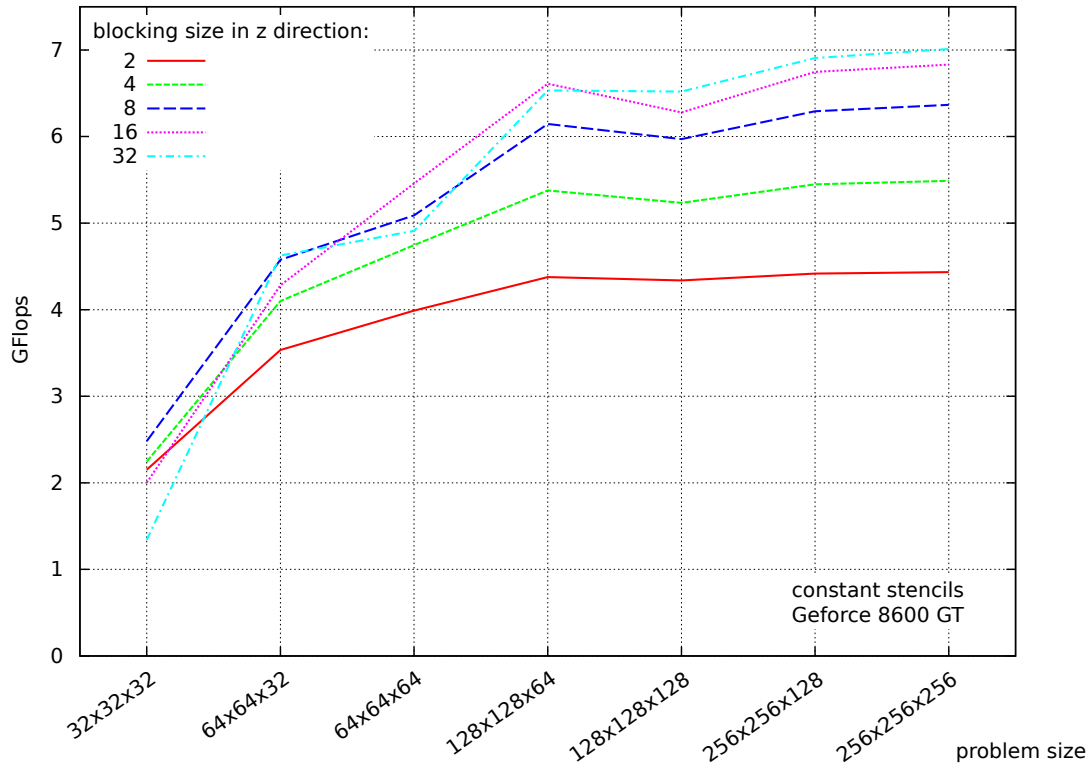


Figure 17: Performance of the Gauss-Seidel solver for constant stencils with various blocking sizes on the GeForce 8600 GT

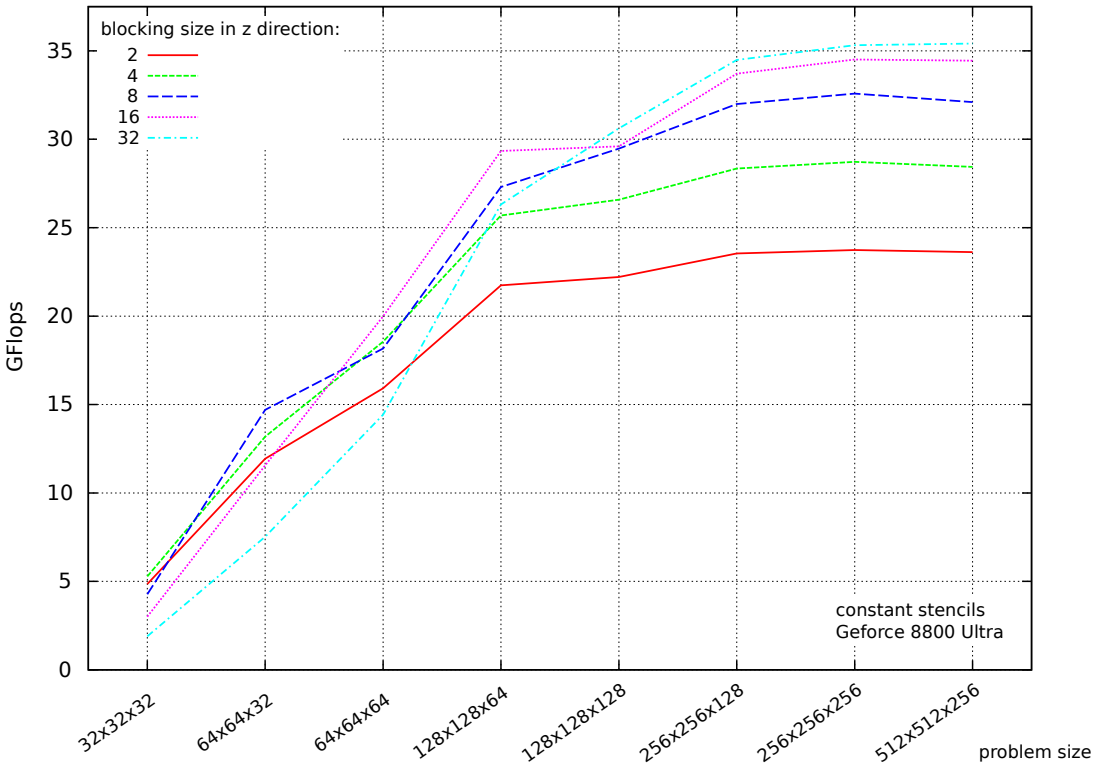


Figure 18: Performance of the Gauss-Seidel solver for constant stencils with various blocking sizes on the GeForce 8800 Ultra

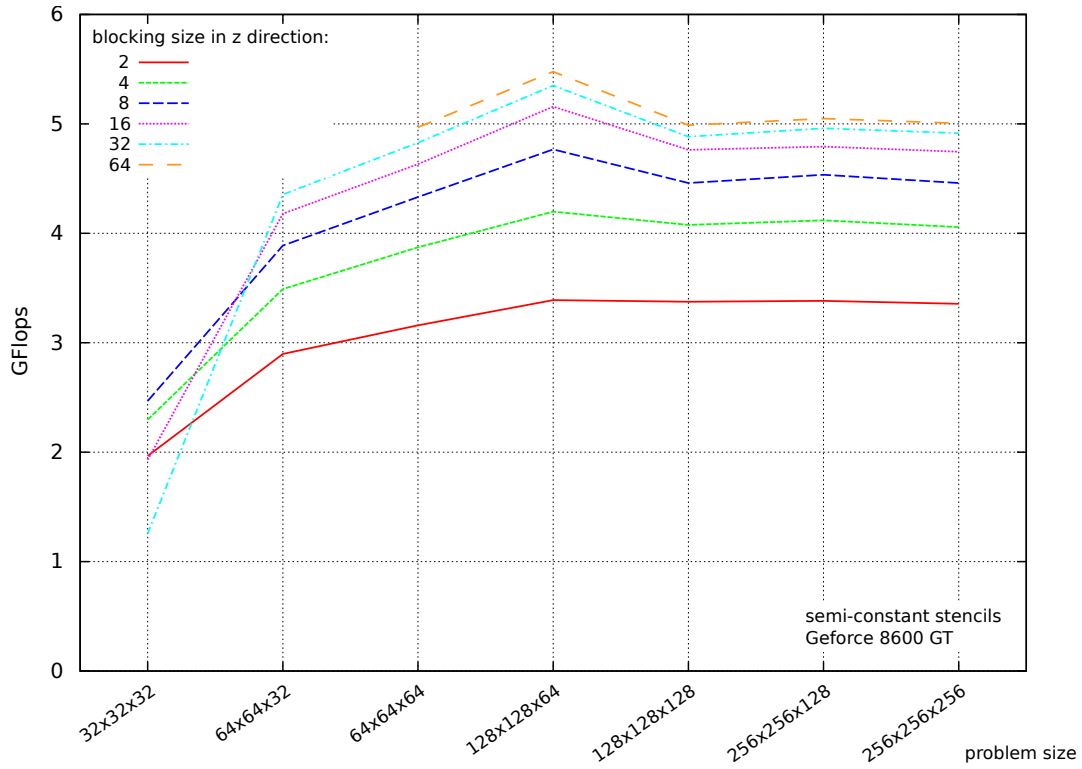


Figure 19: Performance of the Gauss-Seidel solver for semi-constant stencils with various blocking sizes on the GeForce 8600 GT

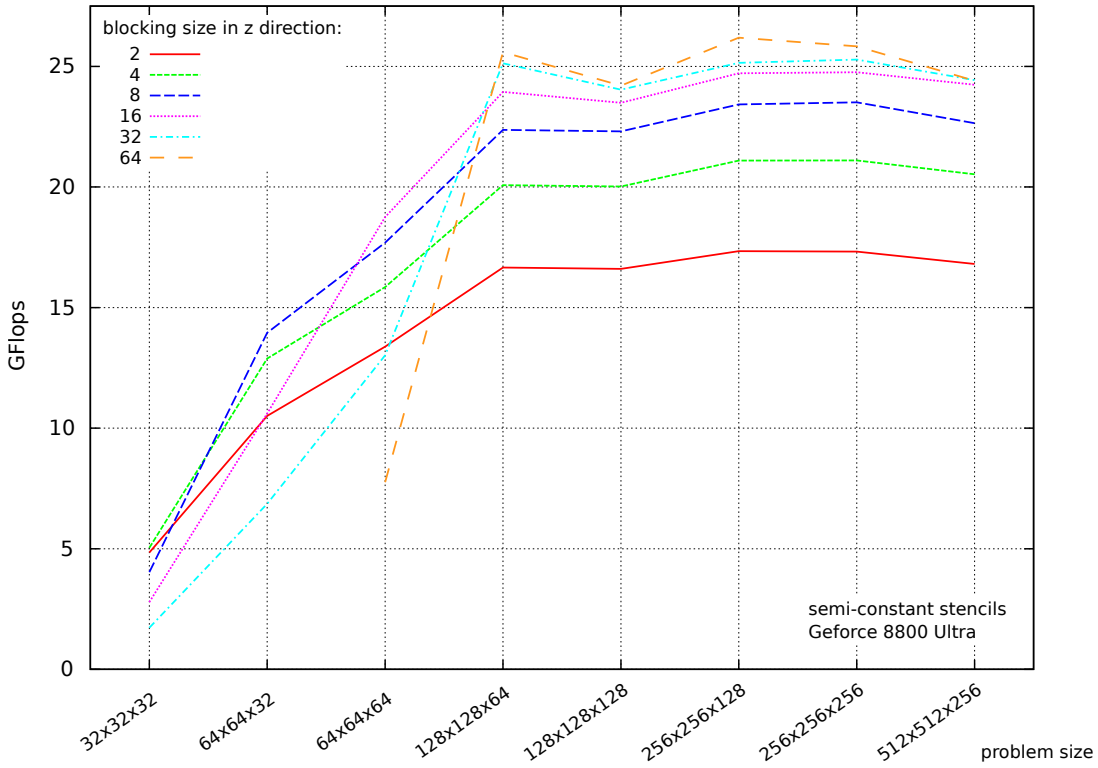


Figure 20: Performance of the Gauss-Seidel solver for semi-constant stencils with various blocking sizes on the GeForce 8800 Ultra

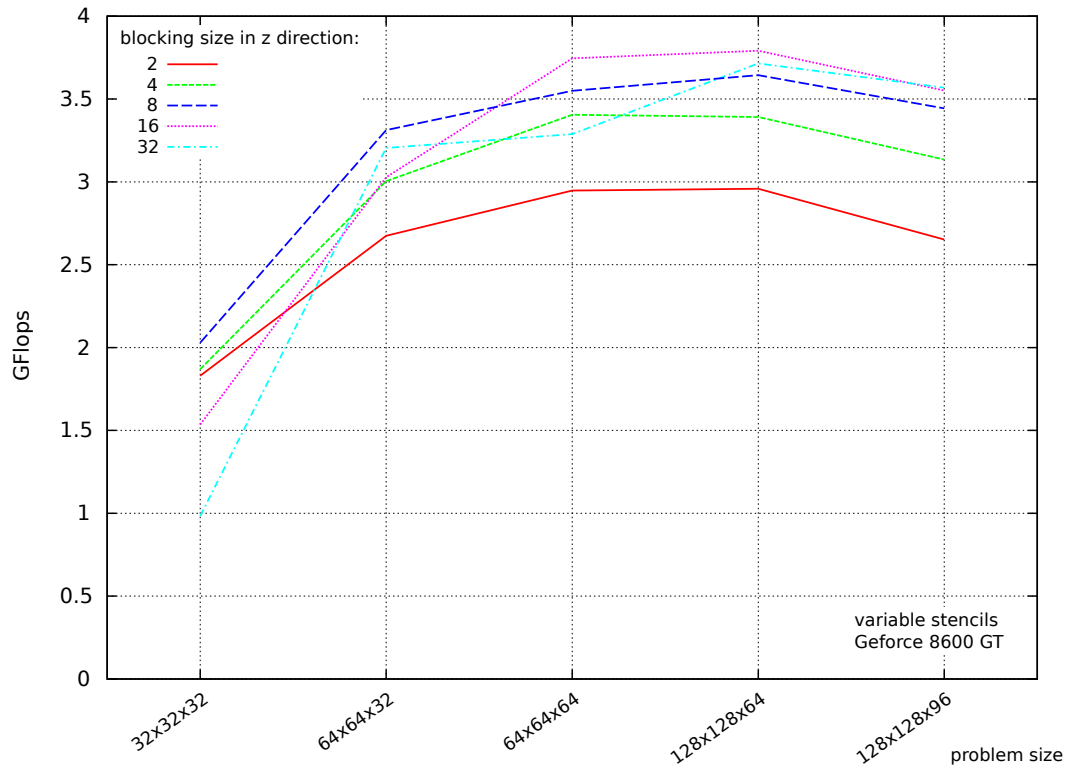


Figure 21: Performance of the Gauss-Seidel solver for variable stencils with various blocking sizes on the GeForce 8600 GT

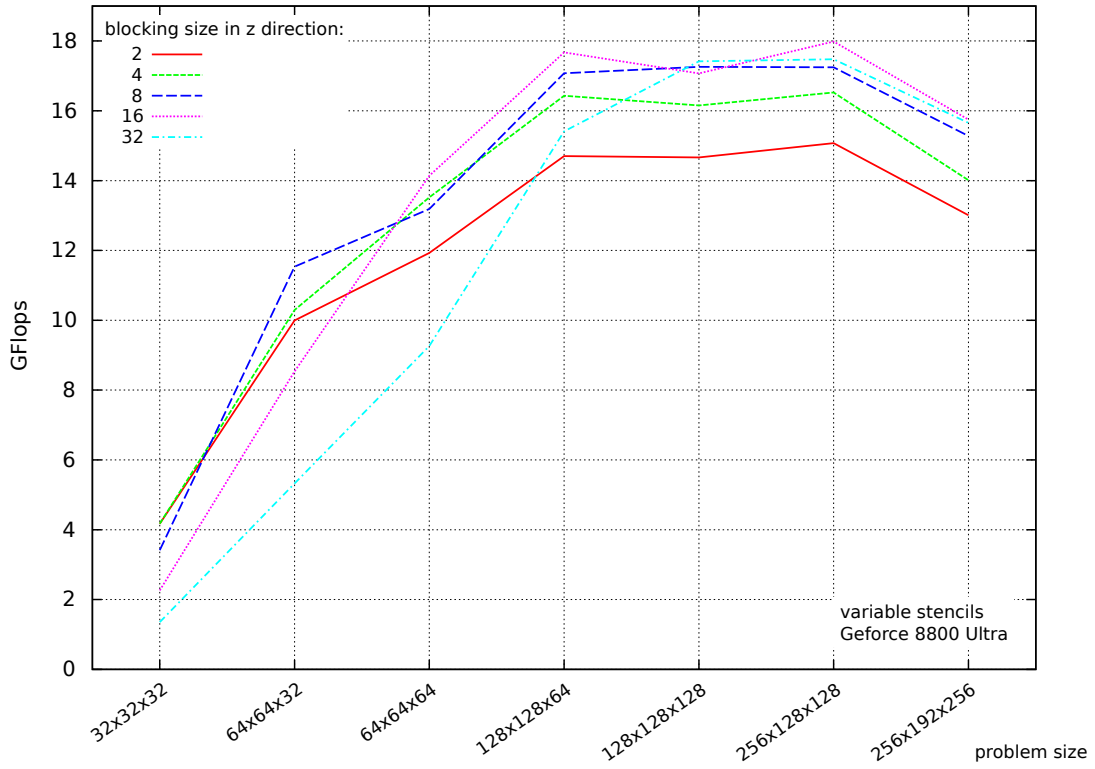


Figure 22: Performance of the Gauss-Seidel solver for variable stencils with various blocking sizes on the GeForce 8800 Ultra

As it was to be expected for small domains in the z dimension, reduced z-blocking leads to a significant higher performance, even if the calculation of all offsets has to be done more often. The effect was even higher on the GeForce 8800 Ultra, as this card has a higher number of streaming multiprocessors. This fact is also the reason why for slightly bigger problems ($64 \times 64 \times 32$) the performance with larger block sizes gained weight faster on the GeForce 8600 GT. Here only 2 SMs had to be provided with blocks, whereas the 8800 Ultra with its 8 SMs still was partly idle. This trend holds for both cards. As soon as all SMs have enough blocks to process the performance stays more or less constant. However it is interesting that for the largest problems in the semi-constant and variable case the performance declines quite noticeable on both GPUs. Presumably the memory controller in the GPU has a hard time dealing with this case, as the memory reads are scattered over a larger area. From the various results it is evident that no simple rule for all GPUs, problem sizes and stencil types can be be given to calculate the optimal block size for all cases. For constant stencils a forth of the domain size in z-direction seems to be a good call with 32 being the upper limit. For semi-constant and variable stencils slightly larger block sizes tend to be preferable.

To evaluate the performance of the GPU accelerated Gauss-Seidel implementations, the results are compared to the CPU versions of the corresponding solvers. All tests were performed on the `fauia116` system, which holds two AMD Opteron 2222 dual core CPUs (3.0 GHz, 2 MB L2 cache) and 16 GB of DDR2 RAM. As the host versions of the solvers include support for OpenMP, all solvers were tested with both one and four threads active. It has to be noted, that the performance of the CPU versions of the solvers is not the ultimate possible, as the focus of this thesis was on the GPU side. However the general trends are surely valid. All tests were performed using single precision floating point numbers, as the used GPUs do not yet support double precision. For the GPU results, the optimal z-blocking values were used. As the GeForce 8600 GT only has 256 MB of memory, the biggest test cases could not be run for this GPU.

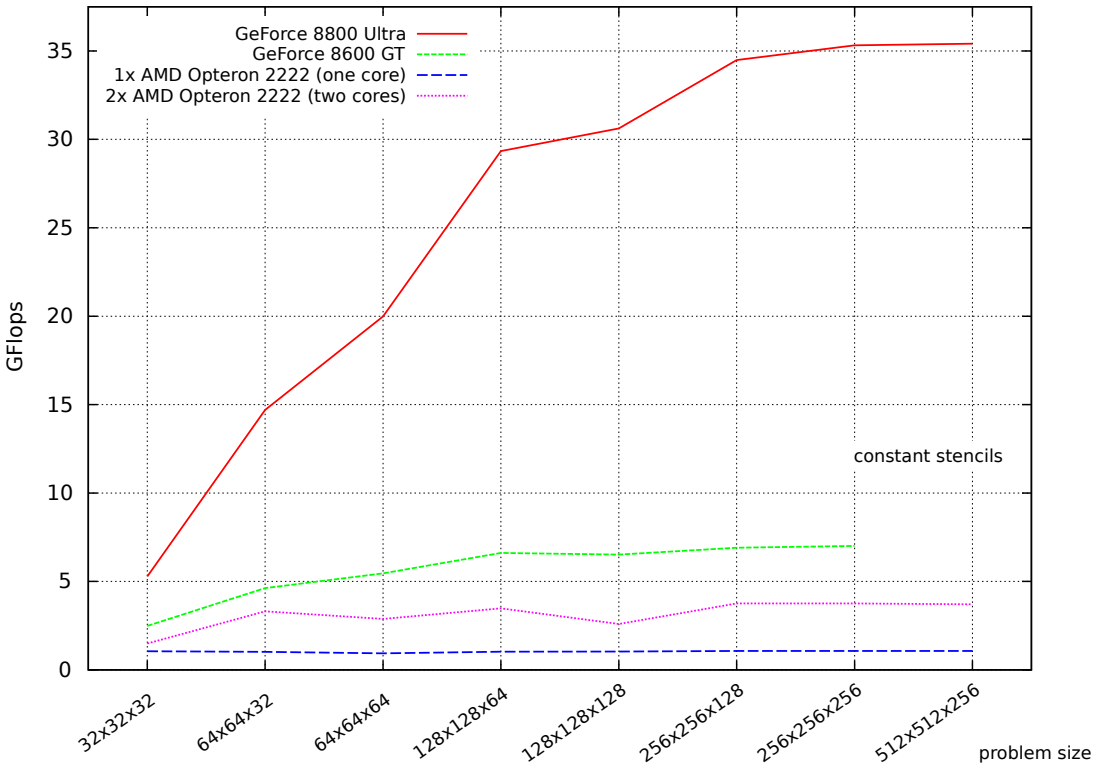


Figure 23: Performance of the Gauss-Seidel solver for constant stencils

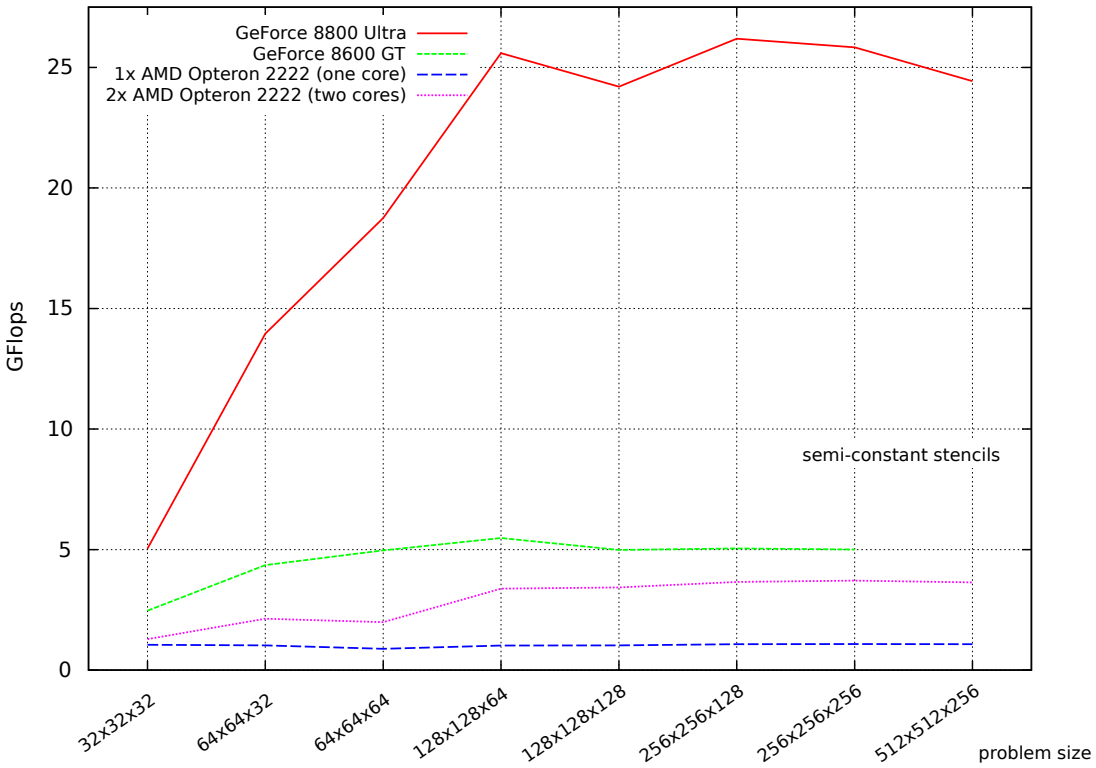


Figure 24: Performance of the Gauss-Seidel solver for semi-constant stencils

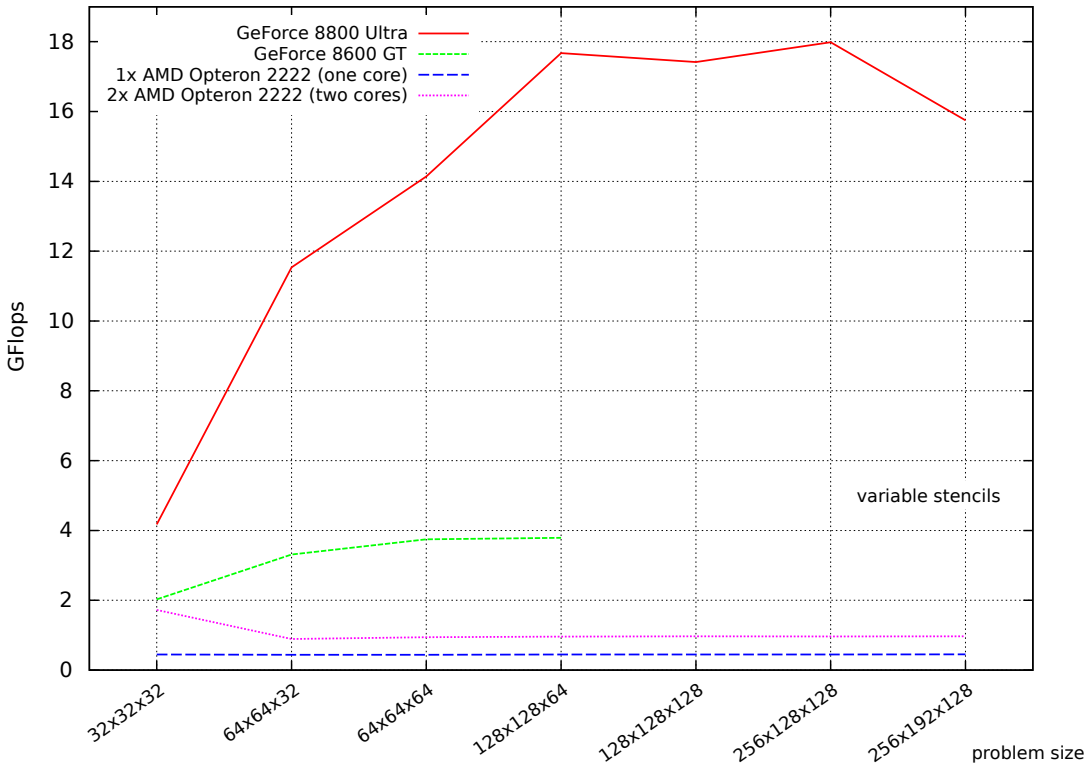


Figure 25: Performance of the Gauss-Seidel solver for variable stencils

The Gauss-Seidel implementations on the GPU easily outperform the CPU versions for all tests. While the difference is not that dramatically for small domains, as soon as the GPUs run under full load, the performance is out of reach for any CPU. To make the comparison easier, the results of the GPUs were normalized to the performance of one CPU core in the following table.

Domain size	**GeForce 8600 GT**	**GeForce 8800 Ultra**	**Opteron 2222 (1 core)**	**2x Opteron 2222 (4 cores)**
32x32x32	2.36	5.03	1.00	1.42
64x64x32	4.52	14.37	1.00	3.23
64x64x64	5.86	21.47	1.00	3.10
128x128x64	6.43	28.52	1.00	3.38
128x128x128	6.31	29.64	1.00	2.52
256x256x128	6.46	32.25	1.00	3.51
256x256x256	6.58	33.14	1.00	3.52
512x512x256	-	33.10	1.00	3.47

Table 3: Normalized performance of the Gauss-Seidel solver with constant stencils

The GPUs show a impressive performance, especially the GeForce 8800 Ultra. But even the 8600 GT, which was considered a mid-range card some time ago achieves about 5 to 8 times the performance of a 3.0 GHz CPU. Interestingly enough the GPUs reach the highest normalized performance for the Gauss-Seidel with variable stencils. So the quite complex memory layout for these stencils clearly pays of. Here the GPUs can make full use of their superior memory bandwidth. To get a feeling for the efficiency of the implemented algorithms in regard to the memory throughput, another series of test was performed. The domain size

Domain size	GeForce 8600 GT	GeForce 8800 Ultra	Opteron 2222 (1 core)	2x Opteron 2222 (4 cores)
32x32x32	2.35	4.80	1.00	1.23
64x64x32	4.26	13.65	1.00	2.08
64x64x64	5.61	21.17	1.00	2.25
128x128x64	5.37	25.08	1.00	3.31
128x128x128	4.87	23.60	1.00	3.34
256x256x128	4.71	24.41	1.00	3.41
256x256x256	4.63	23.88	1.00	3.43
512x512x256	-	22.72	1.00	3.38

Table 4: Normalized performance of the Gauss-Seidel solver with semi-constant stencils

Domain size	GeForce 8600 GT	GeForce 8800 Ultra	Opteron 2222 (1 core)	2x Opteron 2222 (4 cores)
32x32x32	4.51	9.28	1.00	3.83
64x64x32	7.57	26.36	1.00	2.04
64x64x64	8.51	32.11	1.00	2.14
128x128x64	8.45	39.39	1.00	2.14
128x128x128	-	38.70	1.00	2.15
256x128x128	-	40.03	1.00	2.15
256x192x128	-	34.72	1.00	2.14

Table 5: Normalized performance of the Gauss-Seidel solver with variable stencils

for these benchmarks was set to $256 \times 256 \times 256$ for constant stencils (z-blocking: 32) and to $128 \times 128 \times 64$ for semi-constant stencils (z-blocking: 64) and variable stencils (z-blocking: 16).

Stencil type	GeForce 8600 GT			GeForce 8800 Ultra		
	read	write	combined	read	write	combined
constant	5.75	2.02	7.77	28.99	10.17	39.15
semi-constant	5.46	1.57	6.68	24.18	7.44	31.62
variabel	10.61	1.06	11.67	49.92	4.99	54.91

Table 6: Memory throughput of the Gauss-Seidel implementations in GB/s

The memory throughput of the GPUs is quite impressive. Notably is the significant higher performance when using variable stencils. This leads to the assumption that the overall performance of the Gauss-Seidel with constant and semi-constant stencils is limited not by the memory accesses but by the performed computations. However the achieved GFlops rates are well below the possibilities of the hardware. So there might still be room for some improvements in this region.

8 Conclusion

The performance comparisons show the efforts to port the Gauss-Seidel method as a stencil-based solver to a current GPU architecture are rewarding. Despite the initial difficulties a basic version of the GPU implementation can be produced relatively easy using tools like NVIDIA's CUDA. By putting more work into the optimization the potential of today's GPUs can be used for scientific computing on a scale unreachable by standard CPUs. As even a problem like the multi-color Gauss-Seidel with its quite complex memory access patterns can be accelerated considerable, it is tempting to try to port other numerical algorithms to GPUs. The hardware becomes more powerful with each revision, by now even double precision floating point operations are fully supported. Also the software side advances: CUDA becomes more mature with each version, the documentations is improved and the GPGPU community grows.

Although the results so far are promising, GPUsolve still could be improved in many ways. It would be interesting to include a different class of solver like the Conjugate Gradient method, which makes use of many vector operations. Also support for multiple GPUs in a system as well as for more than one host system each with one or two GPUs might offer even higher performance. Another approach may lead to the combined power of the GPU and CPU(s), both sides working on the appropriate part of a problem at the same time.

It has been shown by the implementation of GPUsolve that it is possible to port algorithms for scientific computations to GPU architectures like Tesla and create very efficient iterative solvers. The problem of stencil-based finite element operators first seemed not very well suited for GPU programming, especially the handling of the boundary values in the single blocks was a challenge. But in the end appropriate memory layouts were implemented, resulting in high performance. Also the integration of CUDA code into a library making extensive use of C++ template programming was successful, resulting in a user-friendly, object oriented interface.

References

Dietrich Braess. *Finite Elemente*. Springer, Berlin, 2007. ISBN 3540724494.

Tobias Brandvik and Graham Pullan. Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware. January 2008.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

Jochen Härdtlein. *Moderne Expression Templates Programmierung - Weiterentwickelte Techniken und deren Einsatz zur Lösung partieller Differentialgleichungen*. PhD thesis, FAU Erlangen, 2007.

Mark Harris. SUPERCOMPUTING 2007 Tutorial: High Performance Computing with CUDA - Optimizing CUDA. Course Notes, November 2007. http://www.gpgpu.org/sc2007/.

Erik Lindholm and Stuart Oberman. HOT CHIPS 19 - NVIDIA GeForce 8800 GPU. Conference Talk, August 2007. http://www.hotchips.org/archives/hc19/.

Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008. ISSN 0272-1732.

NVIDIA Corporation. *CUDA - CUBLAS Library*, 2.0 edition, March 2008a.

NVIDIA Corporation. *CUDA - CUFFT Library*, 2.0 edition, April 2008b.

NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2.0 edition, June 2008c.

NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Reference Manual*, 2.0 edition, June 2008d.

NVIDIA Corporation. *The CUDA Compiler Driver NVCC*, 2.0 edition, April 2008e.

NVIDIA Corporation. *NVIDIA Compute - PTX: Parallel Thread Execution*, 1.2 edition, April 2008f.

John Owens. Siggraph 2007 GPGPU Course - GPU Architecture Overview. Course Notes, August 2007. http://www.gpgpu.org/s2007/.

John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

Matt Pharr. *GPU Gems 2*. Addison-Wesley, Boston, 2005. ISBN 0321335597.

David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley, Boston, 2003. ISBN 0201734842.

Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-302-6.