GPGPU-based Multigrid Methods

Leopold-Franzens-University of Innsbruck, Austria
Department of Computer Science
Infmath Imaging Working Group

Master's Thesis

Supervised by: Univ.-Prof. Dr. Otmar Scherzer

Peter Thoman
peter@metaclassofnil.com

Innsbruck, July 30, 2007

Abstract

Multigrid methods are evaluated for their suitability towards a GPU implementation. A Jacobi-based variant of a multigrid solver for the 2D Poisson equation built on C++, OpenGL and GLSL is presented. The performance of various implementation techniques is benchmarked and interpreted, a number of optimization strategies are tested and the final results are compared across different hardware platforms and to a traditional CPU-based implementation.

Contents

Introduction

Since the advent of high performance Graphics Processing Units (GPUs) there have been numerous efforts to use their capabilities in ways different from the original intention. In their latest versions they provide high-performance highly parallel Single Instruction Multiple Data (SIMD) floating point computational resources. While using GPUs for non-rendering tasks has some history even when they were limited to fixed-function operations, their utility greatly increased with the introduction of user-programmable shading hardware [14]. Since then, GPUs have been used as low-cost stream processors [15] in a variety of applications – both graphics-related ones like image processing [17] and unrelated numerical operations like simulating fluid dynamics [7].

In this work, GPUs will be employed to implement a multigrid method [2] for solving boundary-value problems (BVPs) in elliptic partial differential equations (PDEs). This allows for a wide variety of practical applications, but the focus of this thesis is establishing the performance potential of such a method on current hardware, especially vis-à-vis conventional CPU-based implementations.

The first chapter will provide some background regarding the mathematical foundations of multigrid methods on the one hand, and a summary of the current state of general purpose GPU (GPGPU) programming on the other. Finally, an overview of previous works in the field will be given. After these basics have been established, our implementation will be described in detail, including motivation for all design choices, mostly in the form of benchmarks.

As mentioned above, the focus in this work is on the performance potential of multigrid methods on the most recent GPU architectures. Thus the second chapter will concern itself with evaluating the benchmark results of various architectures, and comparing them to results obtained via conventional CPU-based implementations. Additionally, some optimization strategies will be presented and have their effectiveness tested.

In the final chapter possible future research in this area will be listed,

which can be divided into two groups: one adopting to advances in graphics hardware and its programming paradigms, and the other implementing different variations and extensions of the multigrid idea on GPUs.

Chapter 1

Background

To implement a numerical PDE solver based on the multigrid method on GPUs it is essential to first understand that method, and for that reason the first part of this chapter will explain its basics. It will also identify the central building blocks of the algorithm that will be mapped to a SIMD implementation in the second chapter, and present the model problem that will be treated throughout this work.

While GPGPU is no longer as obscure a field as it was a few years ago, it still seems prudent to provide a short summary of the how and why of it. This will be accomplished in the second section of this chapter. In the end, an overview of previous works explicitly in the field of GPGPU multigrid will be listed and described.

1.1 Multigrid Methods

Multigrid Methods are popular as they allow the fast, numerical, iterative solving of systems of equations. They use a combination of classical iterative solvers – Jacobi or Gauss-Seidel, for example – with a hierarchy of discretizations to achieve this goal. An introduction to the methods is found in Briggs et al. [2], and a comprehensive treatment is given in the book by Trottenberg, Oosterlee and Schüller [19].

One of the most important theoretical properties of multigrid methods is their complexity class. Full multigrid implementations are $O(n)$ in both space and time and thus among the most efficient solvers available. As we are focused on the efficiency aspects of *implementations* here, such properties will not be explored further.

1.1.1 The Model Problem

Elliptic PDEs are the most common application of multigrid methods. Therefore, we chose the prototype of such equations, the discrete two-dimensional Poisson equation with Dirichlet boundaries in the unit square $\Omega = (0,1)^2$, as our test case. This problem is given by

$$\begin{aligned} -\Delta_h u_h(x,y) &= f_h^{\Omega}(x,y) \\ u_h(x,y) &= f_h^{\Gamma}(x,y) \quad \text{for} \quad ((x,y) \in \Gamma_h = \partial\Omega_h) \end{aligned}$$

with boundary conditions $f_h^{\Gamma}(x,y)$ and discretization width $h = 1/n$, $n \in \mathbb{N}$ being the number of grid points in each direction.

Using the standard $O(h^2)$ numerical five-point approximation of $-\Delta_h$ we arrive at

$$\begin{aligned} -\Delta_h u_h(x,y) &= 1/h^2[4u_h(x,y) - u_h(x-h,y) - u_h(x+h,y) \\ &\quad -u_h(x,y-h) - u_h(x,y+h)] \\ &= 1/h^2 \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}_h u_h(x,y) \end{aligned} \tag{1.1}$$

as our final formula. The *stencil notation* employed in (1.1) will be used in future formulas, as it maps well to the hardware implementation.

1.1.2 The Multigrid Algorithm

Many traditional iterative solvers like the ω-Jacobi or Gauss-Seidel methods exhibit the following interesting behavior: high frequency error components are reduced much faster than low frequency error components. As shown in Figure 1.1, this means the error becomes *smooth* in very few iterations but may take a large number of iterations to be reduced globally.

To understand this behaviour, the eigenvectors and eigenvalues of the iteration matrix should be examined. Figure 1.2 shows these values for ω-Jacobi and a selection of relaxation parameters. As shown, the selection of the ω parameter can be used to influence which error frequencies should be dampened most rapidly.

From this observation the multigrid method has been derived. By iterating on different discretization widths, all frequency components of the error are reduced efficiently. This is achieved via the Coarse Grid Correction (CGC) process:

1. Start with a few smoothing steps – that is, steps of the iterative solver. (Presmoothing)

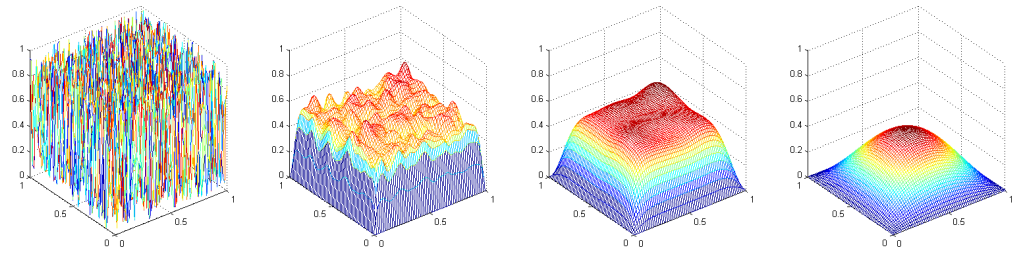

Figure 1.1: Smoothing properties of the Jacobi method. From left to right: error after 0, 10, 100 and 1000 iterations

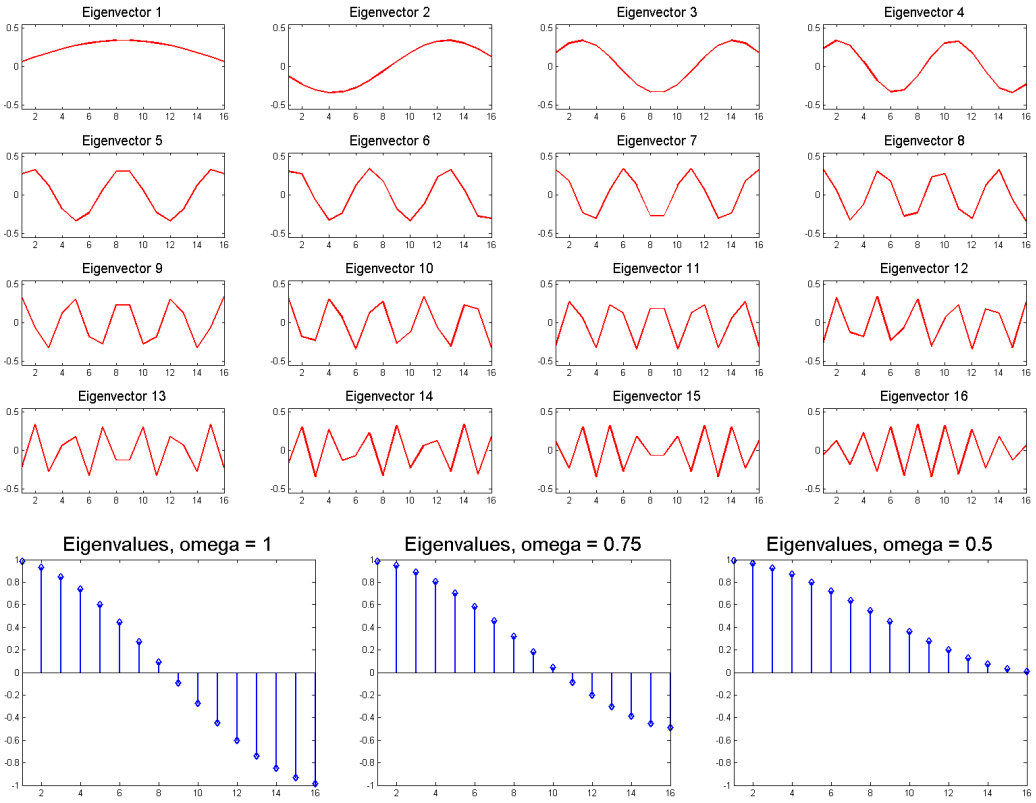

Figure 1.2: Eigenvectors and Eigenvalues of the ω-Jacobi iteration matrix on 16 nodes.

2. Calculate the residual and transfer it to a coarser grid.

3. Solve for the residual on the coarser grid.

4. Transfer the solution back to the finer level, and add it to the existing estimate.

5. (Optional) Reduce errors introduced by the transfer process by additional smoothing. (Postsmoothing)

Formally, iteratively solving an equation $Lu = f$ using this method can be described as follows:

$$\begin{aligned} \bar{u}^m &= \text{SMOOTH}^{n_1}(u^m, L, f) \\ r^m &= f - L\bar{u}^m \qquad (1.2) \\ r_c^m &= Rr^m \\ L_c v_c^m &= r_c^m \qquad (1.3) \\ v^m &= Iv_c^m \\ \breve{u}^m &= \bar{u}^m + v^m \\ u^{m+1} &= \text{SMOOTH}^{n_2}(\breve{u}^m, L, f) \end{aligned}$$

With m being the iteration index, R and I restriction and interpolation operators, r the residual, v the correction, SMOOTH the smoothing method used and n_1, n_2 the number of pre- and postsmoothing steps, respectively.

The remaining question is how to solve the equation for the residual in (1.3). As it is of the same form as the original problem, but smaller, the obvious answer is to use the same method recursively until reaching some lowest level where the solution can be derived directly. This approach yields the basic multigrid methods. Figure 1.3 illustrates the process on a simple example.

What are the advantages of this approach? Primarily, the frequencies at coarser grids are progressively "stretched", so errors of all frequencies get reduced efficiently. Additionally, the problem size is reduced to $\frac{1}{4}$ with each coarsening (for 2D problems), which directly impacts the computational effort required by the solver. In the next section, we will examine the individual steps of this algorithm in more detail.

1.1.3 Components of the Multigrid Method

Multigrid methods are very flexible and can be applied on a wide variety of structures in numerous ways. In this section, the choices made for the individual components of the GPU implementation are presented and justified.

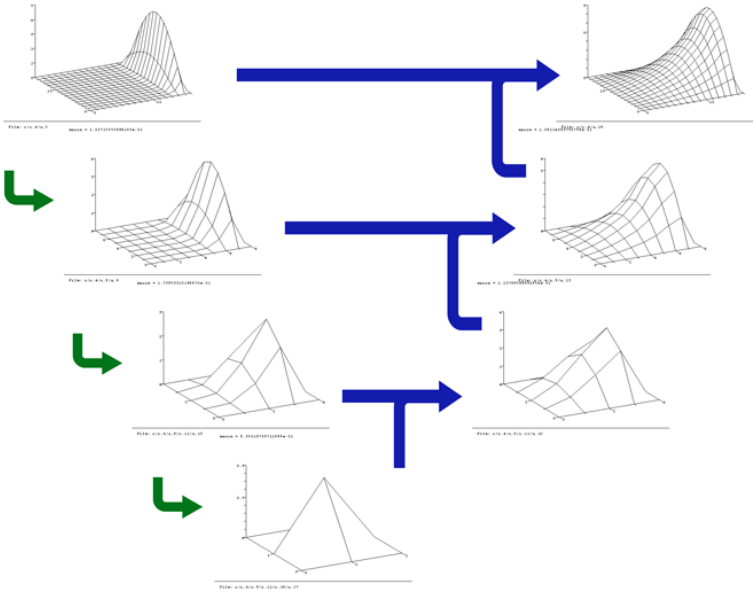

Figure 1.3: Multigrid method example, arrows show the flow of data. Solution images from Rüde [16].

Where possible, the methods most suited to achieving optimal performance in a GPGPU setting will be identified. The components and operations discussed are the following:

- Grid and coarsening type
- Smoothing/Relaxation method
- Residual calculation
- Restriction (fine to coarse transfer)
- Interpolation (coarse to fine transfer)
- Cycle type

Grid and Coarsening Type

The most common grid coarsening types are shown in Figure 1.4. Semi-coarsening is only advantageous for specific applications, which leaves the choice between standard and red/black coarsening. While the latter allows using the very efficient Gauss-Seidel Red/Black (GS-RB) smoother (see next section), it is not as well suited to GPUs because of their cache and memory layout. We use standard coarsening.

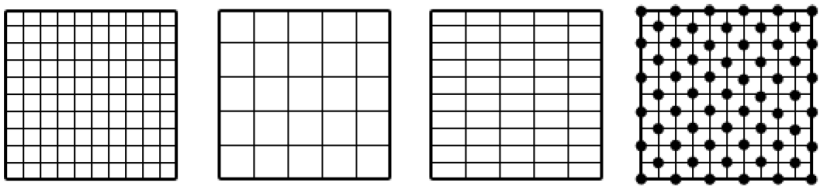

Figure 1.4: Grid coarsening – from left to right: original grid, standard-, semi- and red/black-coarsening

Smoothing/Relaxation Method

The Jacobi method with a relaxation parameter (ω-JAC) or the Gauss-Seidel method with either lexicographical (GS-lex) or red/black sorting of grid points are the common choices. On GPUs we are most interested in the parallelization properties of these methods, as summarized by Trottenberg et al [19]. Their findings are shown in table 1.1. From these numbers,

Method	**Smoothing factor**	**Smoothing**	**Parallelization**
ω-JAC, $\omega = 1$	1.00	None	N
ω-JAC, $\omega = 0.5$	0.75	Unsatisfactory	N
ω-JAC, $\omega = 0.8$	0.60	Acceptable	N
GS-LEX	0.50	Good	$\sqrt{N}$
GS-RB	0.25	Very Good	$\frac{1}{2}N$

Table 1.1: Parallelization possibilities versus smoothing properties of iterative solvers

the only acceptable candidates for a – inherently highly parallel – GPU implementation are 0.8-JAC and GS-RB. While the latter has better smoothing properties, the parallelization advantages of 0.8-JAC compounded with the additional cache and addressing complexities and related overhead introduced by GS-RB lead us to chose the Jacobi-based method. It can be easily implemented using a five-point stencil on the standard grid.

Residual Calculation

Calculating the residual numerically is very similar to applying one step of Jacobi smoothing, the formula is provided in (1.2). Given the choices already made, there are no different methods to consider here.

Restriction (fine to coarse transfer)

The intuitive restriction operators for standard coarsening are given by the

stencils in formula (1.4).

$$\begin{aligned} \text{Injection:} \quad & 1 \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \text{Half Weighting:} \quad & \tfrac{1}{8} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \\ \text{Full Weighting:} \quad & \tfrac{1}{16} \quad \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \end{aligned} \tag{1.4}$$

All of these have been implemented, but in benchmarks and experiments full weighting is used exclusively unless otherwise mentioned. While the simpler methods are comparatively faster, their speed advantage is usually inhibited by the scaling errors they introduce.

Interpolation (coarse to fine transfer)

Interpolation is usually done by applying bilinear filtering, as specified by the distributive stencil in formula (1.5).

$$\text{Bilinear Filtering:} \quad \tfrac{1}{4} \quad \left] \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix} \right[\tag{1.5}$$

Cycle Type

The cycle type of a MG method is determined by the amount of times a coarse grid correction is performed before interpolating the result back to the upper level. The three main cycle types found in literature are shown in figure 1.5.

It will be shown in later chapters that – due to the high degree of parallelization and high overhead costs – iterations on coarse grids are very wasteful on modern GPUs. As W and F cycles both require more computations on very coarse grid levels without providing significant advantages the GPU implementation focuses on V cycles.

1.2 General Purpose GPU Programming

Interest in using the vast computational capabilities of GPUs for many numerically intensive problems has been on the rise for the past 5 years, since

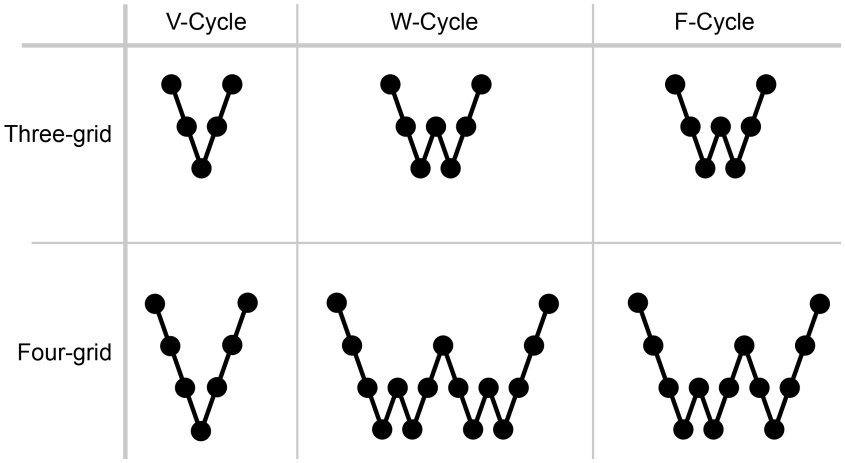

Figure 1.5: Multigrid cycle types

the introduction of user-programmable shaders [14]. A short history of and introduction to the field can be found in [17]. Very recently, the major graphics card manufacturers NVIDIA and ATI (now AMD) have officially recognized these efforts by providing APIs tailored to this kind of usage, named CUDA [5] and CTM [10] respectively. While neither of these are used for this implementation, a look into the advantages and disadvantages compared to traditional GPGPU development can be found in section 2.5.

This part of the document is not intended to provide details on all intricacies of GPGPU development. It will, however, give a short summary, provide some performance data and look into reasons that motivate implementing multigrid methods on GPUs.

1.2.1 GPGPU Overview

When performing GPGPU computations, GPUs can be seen as highly parallel streaming SIMD processors. Array data is kept in one- or two-dimensional textures with 1 to 4 components of INT8, FP16 or FP32 type. The program parts that would traditionally make up the inner loop of a SIMD program are executed by the GPU as pixel shaders – these are (usually small) code fragments specifying calculations that are carried out for each destination pixel to determine its value.

Figure 1.6 illustrates the mapping of a very simple CPU program working on a few 1D arrays to some GPU. The details of texture creation and rendering were omitted, as they are long-winded and differ between APIs.

The most important advantages compared to a CPU implementation are as follows:

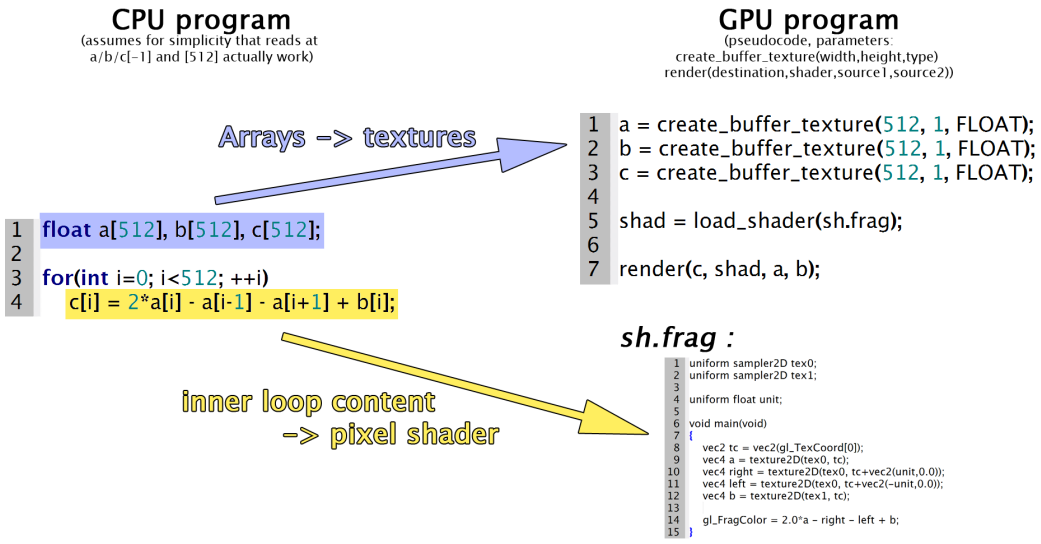

Figure 1.6: Mapping a simple CPU program to GPGPU.

- Very high degree of parallel execution, enabling high speedups.
- High bandwidth, both on-chip and external.
- For some algorithms, a cache structure optimized for 2D-locality is advantageous.
- CPU is mostly free to perform other tasks while GPU is working.

However, there are also some serious limitations and drawbacks:

- I/O exclusively via textures, which only allow a restricted amount of formats and accesses.
- As of yet, no double precision data types.
- Very slow branching, especially if the taken branch alternates repeatedly.
- Limited support for Integers, particularly for bit-wise operations.
- Large overhead costs, therefore GPGPU only makes sense for big problem sizes.
- Fixed amount of on-board memory, transfers to and from main memory are costly.

- Most mathematical operations beyond basic arithmetics are only available as either reduced-precision approximations or slow software implementations.
- Complicated programming model for non-graphics programmers.
- Until recently, no direct support for scalar operations. Only 4-component vector operations were supported (due to the heritage of RGBA color and XYZW geometry processing) natively, to achieve a speedup for scalar-only computations some packing method had to be used [18]. APIs as of yet still don't fully support single-component computations.

Obviously, the main draw of GPGPU development is performance. How much there is to be gained in this area will be investigated in the next section.

1.2.2 Theoretical Performance Data

To understand the interest in GPGPU we shall examine a modern high-end GPU – NVIDIA's G80 – and compare its theoretical performance with that of a similarly priced CPU (as of June 2007), Intel's Core 2 Quad 6600. Note that all the following estimates only hold true under the unrealistic assumption of perfect utilization of all computational and bandwidth resources, and that they use the hardware vendor standard of counting 1 multiply-add (MAD) as 2 FLOPs.

NVIDIA G80 data:

- 16 Multiprocessors, each capable of 8 FP32 MADs per cycle
  → 256 FP operations per cycle
- Shader ALUs clocked at 1350MHz → 345 GFLOPs
- 6x64bit wide external memory bus, clocked at 900MHz DDR
  → 86.4 GB/s memory bandwidth

Core 2 Quad 6600 data:

- 4 cores, each capable of 4 FP32 MADs per cycle via SSE
  → 32 FP operations per cycle
- Clocked at 2400MHz → 76 GFLOPs
- 2x64bit wide external memory bus, clocked at 400MHz DDR
  → 12.8 GB/s memory bandwidth

While these numbers are not very useful on their own, they still show that the GPU has the potential to perform more than 4.5 times as many floating point operations per second, and – perhaps even more important – can transfer nearly 7 times as fast to and from external memory. Figure 1.7 illustrates the difference, and thus the main reason for the practical relevance of GPGPU techniques, graphically.

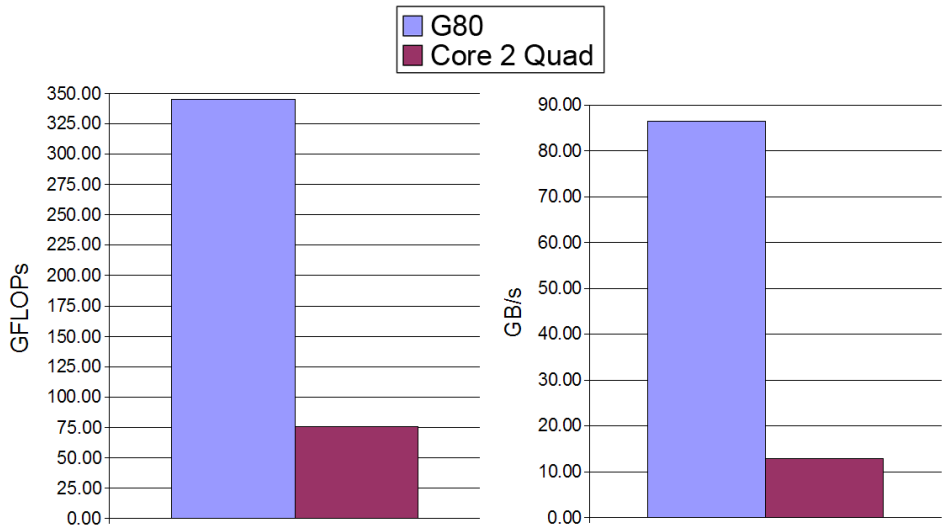

Figure 1.7: Theoretical performance comparison of high-end GPU and CPU

1.2.3 The Multigrid Method and GPUs

From the data and information gathered above, one can surmise that a GPGPU implementation is advisable when a problem is highly parallel in nature and expressible as a number of streaming SIMD operations on – preferably large – blocks of data. It should also require as little branching as possible while making extensive use of floating point operations and memory bandwidth.

The basic multigrid method as described in section 1.1.3, with the components chosen as stated, fits these requirements admirably. Each of the main operations, Jacobi-based smoothing, residual calculation, restriction and interpolation, can be expressed as an independent sequence of data accesses and floating point calculations at each position. The only requirement that is not fulfilled perfectly is that each operation should always be executed on a large block of data. On the coarsest grid level, there is only a single node left, and even the first few levels immediately above are not large compared

to the batch sizes[1] of modern GPUs. Thus some inefficiencies of the GPGPU implementation at those grid levels are to be expected.

1.3 Previous Work

As explained above, multigrid methods seem well suited to a GPU-based implementation. As such, it is not particularly surprising that there have been quite a few previous efforts in the the field.

Bolz et al. in 2002 [1] mapped both conjugate gradient and multigrid solvers to GPUs and tested a reference implementation on the GeForce FX architecture. The most interesting aspect of their work is splitting the problem domain into 4 subdomains to fully use all computational resources on the GPU and work around the 4-component-vector-only problem described in section 1.2.1. They use a fragment program to synchronize boundary conditions between the quadrants. Unfortunately they do not provide data on the overhead costs incurred by this process.

Goodnight et al. in 2003 [7] implement a GPU-based multigrid solver for boundary value problems. They show three practical applications and an in-depth analysis of optimization opportunities. However, a large part of their efforts are directed at reducing pBuffer switching overhead, a task that has thankfully been made obsolete by advances in OpenGL [11] since then.

Four years are a fairly long time in graphics processing, and some limitations these implementations had to work around were lifted by new developments, making the workarounds unnecessary, and, in some cases, detrimental to performance or flexibility. Also, up until now no effort at a comprehensive comparison of multigrid performance on different GPUs has been undertaken. This work aims to provide an implementation focused on contemporary and future architectures, and perform comprehensive performance analysis on multiple GPUs.

[1] An explanation of this concept and its effects on performance can be found in section 2.2.2.

Chapter 2

Implementation

In this chapter, our implementation of the multigrid solver for the model problem described in section 1.1.1 will be presented in detail. After starting with a general description of how the individual components of the algorithm map to GPUs, the technical choices made and libraries used will be described in the second section.

Auxiliary requirements, that is, implementation parts that are not central to the multigrid method, but required to test or use the implementation, are the topic of the following section. Finally, the actual implementation of the central operations of the GPGPU elliptic PDE solver will be examined. The intricacies of interacting with OpenGL will be largely ignored, except when they are relevant to either the design or performance of the implementation.

In section 2.5 some thoughts on a possible CUDA [5] version are gathered. While CUDA only became available when this thesis was well underway, it – and similar low-level APIs like ATI's CTM [10] – offer some unique advantages for the implementation of numerical methods on GPUs.

2.1 Multigrid Method to GPU Mapping

In section 1.1.3 we identified the components of a multigrid solver and made some choices in terms of their implementation. Now, an overview of how those components can be mapped to GPUs will be given.

Grid and Coarsening Type

A regular 2D grid with standard coarsening was selected. On GPUs, this structure is easily represented by a pyramid of FP32 textures. However, due to current restrictions in the OpenGL API that are leftovers from previous 3D hardware, it is not possible to use single-component FP32 textures in

framebuffer objects (FBOs) [11, 12]. This causes a gross inefficiency that is not easily reduced.

Bolz et al. [1] used a four-way split of their problem domain to overcome this limitation. However, this causes additional complexities when resolving boundary conditions. As the additional overhead caused by such methods on modern hardware would be significant, and as the limitation is no longer one caused by hardware design [4], but rather by API inadequacy, we decided to simply use multi-component FBOs. This ensures that no additional overhead is introduced that will be useless once API support for single-component FBOs becomes available. Until such support materializes, one possiblity would be using the "wasted" processing power to solve four similar problems at the same time. This could be useful in applications like video processing.

Jacobi-based Relaxation

ω-Jacobi, our smoothing method of choice, is fully parallelizable and can thus be implemented in a single pass of a pixel shading kernel. It also has good locality in it's memory accesses on a regular 2D grid enabling the texture cache to hide the costs of most repeated fragment reads. The same holds true for **residual calculation**, which is very similar computationally to ω-JAC smoothing.

Restriction and interpolation

Both seem like problems perfectly suited to GPUs at first glance – in fact, bilinear filtering has been supported in GPU hardware for about a decade now. However, when floating-point, and especially FP32 textures are used, there is still no uniform support for any other sampling method than nearest neighbor. This means these methods have to be implemented via pixel shader kernels.

2.2 API and Library Considerations

Until recently, and when the implementation discussed in this chapter was started, the only API choices for GPGPU programs were DirectX and OpenGL, both originally intended for graphics only. The related high level shading languages are HLSL for DirectX, GLSlang for OpenGL and Cg [3] as a third choice.

Out of these three possibilities only the OpenGL/GLSL combination is both multiplatform capable and completely vendor-agnostic. In the past, OpenGL had a significant disadvantage compared to DirectX for GPGPU operations on smaller datasets in the high pBuffer [13] switching costs – this

is one of the main performance hurdles reported by Goodnight et al. [7]. However, this disadvantage has been greatly reduced by the introduction of the framebuffer object extension, as shown by Green [8]. For these reasons, OpenGL with GLSL fragment programs was selected for this implementation.

Performing non-graphics tasks directly with any of these graphics-oriented interfaces is quite cumbersome, so an existing wrapper library aimed at easing GPGPU tasks was used. This library will be described in short in the next section.

2.2.1 The GPGPU Framework

The C++ OpenGL/GLSL wrapper originally developed for PDE-based image processing in [18] provides the following fundamentals for GPGPU operations:

- An easy to use texture/rendertarget class that abstracts from OpenGL specifics, especially during initialization.
- Support for loading, compiling and using GLSL fragment programs, with correct reporting of compiler and linker errors.
- A stream oriented model for intuitively executing the steps of a GPGPU algorithm.
- Loading and management of required OpenGL extensions.
- A number of standard shaders useful as building blocks of a larger algorithm.

The basic object model of the library is shown in figure 2.1. The library is documented in detail and some usage examples are provided in [18].

Since that library was developed, some advances towards better GPGPU support have been made in OpenGL, and some new insight on performance improvements was gained. Changes have been made to accommodate these. They will be summarized in the following section.

2.2.2 Library Changes and Improvements

The most relevant change to OpenGL since 2004 was the introduction of framebuffer objects, which both enhance performance by no longer causing a complete OpenGL context switch and provide additional functionality. In response, the `GLRenderTexture` class was completely reimplemented using FBOs instead of pBuffers.

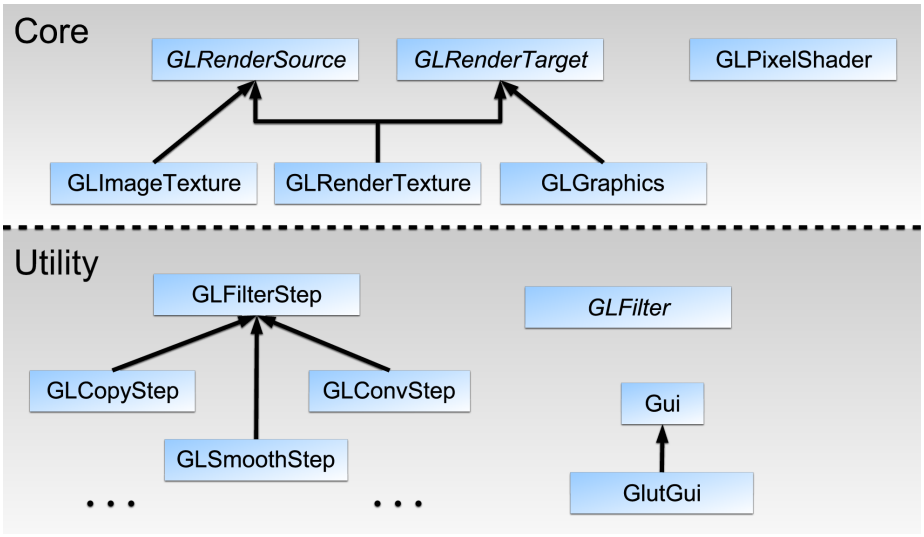

Figure 2.1: GPGPU library object model.

In practical application, this change yielded the results shown in table 2.1. While the gains are nearly non-existent at larger problem sizes of 512^2 and beyond, they are quite significant at sizes were the program was constrained by context switching overhead. This is of particular importance for multigrid methods – regardless of the initial problem size, the algorithm will always be applied at grids of all sizes, down to a single grid cell.

	16×16	***64×64***	***512×512***
Before	4481	3623	87
After	8165	5050	88
Improvement	55%	40%	1%

Table 2.1: FBO performance advantage over pBuffers. Numbers are iterations per second.

Another improvement over the original framework, also aimed mainly at increasing performance at smaller problem sizes, pertains to the actual rendering operation carried out to apply a shading kernel. The concept is illustrated in figure 2.2.

GPUs always compute a rectangular area – a *batch* – of pixels together. These batches can be up to 4×4 pixels in size depending on the architecture. When performing the rendering step intuitively, using a quad formed of two triangles, performance will suffer along the dividing line. As the batch size is constant the effect of this performance loss can be expected to be proportionally greater at smaller framebuffer sizes. If, instead, a single triangle is used

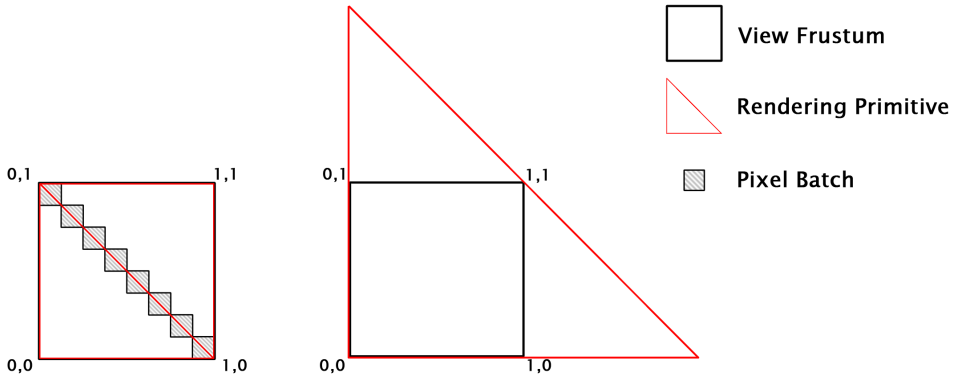

Figure 2.2: Quad versus single triangle rendering.

as the rendering primitive batching should work uniformly over the whole area. The parts of the primitive outside the view frustum get culled automatically and do not have any measurable impact on performance. Table 2.2 shows the effects of this change.

	16×16	***64×64***	***512×512***
Original	4481	3623	87
FBO	8165	5050	88
FBO & Tri	9718	5484	89
Improvement	19%	9%	1%

Table 2.2: Single triangle rendering performance advantage over quads. Numbers are iterations per second.

While the improvement is not as significant as the one caused by the switch to FBOs, it is still worthwhile, especially at very small sizes. There is another interesting fact to note looking at the original values vis-à-vis those after the improvement. Going from 64^2 to 512^2 is an increase in computational effort by a factor of 64. In the unimproved framework, the actual factor measured is just 42, suggesting that performance is suppressed by overhead factors. After both enhancements, the measured factor is 62, very close to the theoretical maximum.

Obviously, the scaling from 64^2 downwards is still very much limited by overhead and aspects other than sheer GPU performance. This is a pattern that plays a significant role for the coarsest levels of the V-cycle, and that will be explored in more depth later.

2.3 Auxiliary Requirements

Implementing the multigrid calculations directly would at this point be possible based on the improved framework described above. However, to judge the accuracy of the computations, or even just determine whether the correct result is being computed, a few more components are required. Namely, a mechanism for writing data to, and reading it back from, the GPU's memory, and a way to visualize results. Both of these were implemented, and their realization will be described in this section.

2.3.1 Data Input and Output

To facilitate writing arbitrary data to and from GPU memory two methods were added to the `GLRenderTexture` class.

readData(data) Reads the whole amount of data from the FBO to the user-supplied location. While this requires the caller to keep track of the amount of memory needed, it was implemented this way to ensure proper deletion of the allocated space.

writeData(data) Writes the supplied chunk of information to the FBO represented by this object.

In the current implementation these methods use the conventional OpenGL data transfer operations. It is assumed that transfer operations appear infrequently enough that any performance improvements would be insignificant. One enhancement would be using OpenGL Pixel Buffer Objects (PBOs) as described by Göddeke [6]. In any case, if at all possible, initial data, boundary conditions and inhomogeneities should be computed via pixel shaders rather than supplied manually, especially for big problem sizes.

2.3.2 Visualization

For quickly judging whether a result is at all plausible and for debugging visualizations of numerical results are very useful. In keeping with the framework's traditions as introduced in section 2.2.1, a new subclass of `GLFilterStep` was added, `GLVisualizationStep`. It is designed to be expandable to different visualization types, but for now only heightfield visualization is implemented. It is used via one of the following methods.

heightfield(*source, *target, lowbound = 0.0f, highbound = 1.0f) Visualizes the numerical values provided by the source on the target,

coloring values between lowbound and highbound with a smooth gradient.

debugHeightfield(*source, msg, lowbound, highbound) The same as above, but renders directly to screen and displays a supplied message. Also halts program execution until a key is pressed. Useful for debugging the individual steps of a more complex GPGPU algorithm.

The heightfield visualization process is implemented via a fragment program that samples the numerical value provided in the source texture, scales it according to the bounds supplied and uses it to select a color value from a lookup texture. It is thus quite efficient and can also be used to animate the progress of a solver. An example of the output that is produced by this process is given in figure 2.3.

Figure 2.3: Heightfield visualizer example.

2.4 Main Algorithm Implementation

With all the preparative work now described, it is time to delineate the implementation of the multigrid solver itself. In this section, first the individual components of the algorithm will be examined, and finally it will be shown how they fit together to form a complete multigrid solver.

2.4.1 Restriction

Some of the simplest operations to implement are the various restriction (fine-to-coarse transfer) methods. To facilitate these operations a helper class `MGRestrictStep` was added. It provides the following static methods:

inject(*source, *target) Performs injection from source to target. For this and the following two methods `target` is required to have a size in the range $\left[\left\lfloor\frac{s}{2}\right\rfloor, \left\lceil\frac{s}{2}\right\rceil\right]$ in both dimensions (with s being the width of `source`), otherwise the scaling will not work correctly on all pixels.

half_weight(*source, *target) Performs reduction via half-weighting.

full_weight(*source, *target) Performs the full-weighting downscaling operation.

GLRenderTexture* inject(*source) This inject operation creates the downscaled buffer instead of requiring the user to supply it.

GLRenderTexture* half_weight(*source) As above, but for the half-weighting operation.

GLRenderTexture* full_weight(*source) As above, but this time for full-weighting.

These operations are very similar and not hard to implement, so only the GLSL code for the full-weighting operation will be given in listing 2.1.

Listing 2.1: GLSL code implementing full-weighting.

```
1  uniform sampler2D tex0;
2  uniform float unit;
3
4  void main(void)
5  {
6      float u = unit*0.5;
7      vec2 tc = vec2(gl_TexCoord[0]);
8      vec4 top = texture2D(tex0, tc+vec2(0.0,-u));
9      vec4 bottom = texture2D(tex0, tc+vec2(0.0,u));
10     vec4 left = texture2D(tex0, tc+vec2(-u,0.0));
11     vec4 right = texture2D(tex0, tc+vec2(u,0.0));
12     vec4 topleft = texture2D(tex0, tc+vec2(-u,-u));
13     vec4 topright = texture2D(tex0, tc+vec2(u,-u));
14     vec4 bottomleft = texture2D(tex0, tc+vec2(-u,u));
15     vec4 bottomright = texture2D(tex0, tc+vec2(u,u));
16     vec4 center = texture2D(tex0, tc);
17
18     gl_FragColor =
```

```
19          (4.0*center + 2.0 * (top + left + right + bottom) +
20            topleft + topright + bottomleft + bottomright) / 16.0;
21  }
```

2.4.2 Interpolation

Bilinear interpolation is slightly more tricky to implement. Figure 2.4 shows the composition of pixels on the finer level. As illustrated, one out of every four pixels can be taken directly from the lower level, two are created by adding two source values each with weights $\frac{1}{2}$ and one by adding 4 with weights $\frac{1}{4}$.

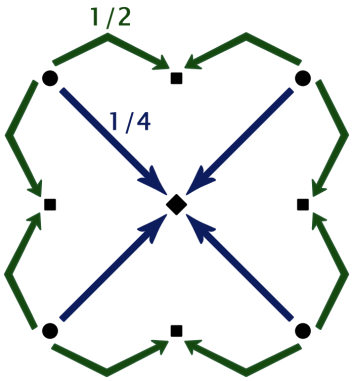

Figure 2.4: Interpolation pixel weights.

On CPUs, it is usually advantageous to use a 1D decomposition of the process, first filling every second line by horizontal interpolation and then interpolating vertically. This method reduces the number of repeated memory reads and operations required.

On GPUs, which always work most efficiently when performing the same calculations on all pixels of the target area, such a decomposition would be wasteful. However, directly implementing scaling as shown in 2.4 is also not a good method, as performing completely different computations depending on the position of each pixel does not fit the GPU model well.

Figure 2.5 shows a better method, and the one that has been used in this implementation. Each node is handled uniformly, by sampling (using the nearest-neighbour method) four times at a small distance around its position and calculating the mean of these four values. Obviously, this results in the same weights as the naive version. Implementing the process in this way may seem very inefficient, however nearly all duplicate reads of different

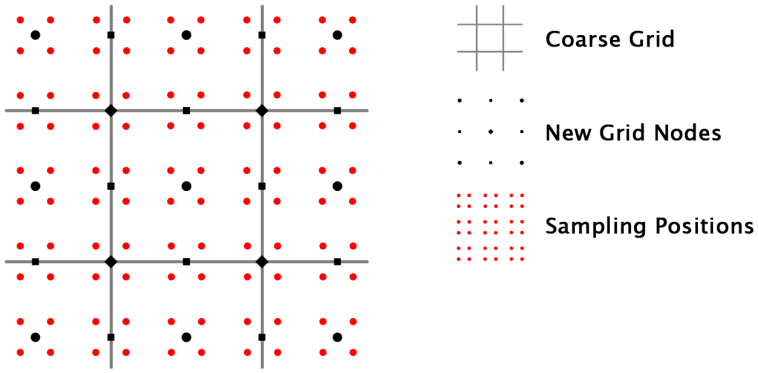

Figure 2.5: GPU bilinear interpolation implementation.

nodes will be cached, and the small additional computations are masked by memory latency on most hardware.

In benchmarks, the GPU-optimized method is nearly 3 times faster than the intuitive, branching implementation on latest G80 hardware. The difference can be expected to be even greater on older GPUs.

For usage in the multigrid algorithm, the interpolation fragment program has been combined with the addition of the result to the existing estimate. This not only reduces the overhead caused by each rendering step, but also allows us to use only three buffers at each grid level. The kernel for performing both interpolation and addition is shown in listing 2.2.

Listing 2.2: GLSL code implementing interpolation and addition.

```
1  // Textures:
2  // tex0 - the texture to interpolate
3  // tex1 - the additive texture
4
5  uniform sampler2D tex0;
6  uniform sampler2D tex1;
7  uniform float unit;
8
9  void main(void)
10 {
11     vec2 co = vec2(gl_TexCoord[0]);
12     float u = unit*0.25;
13     vec4 a = texture2D(tex0, co+vec2(-u,-u));
14     vec4 b = texture2D(tex0, co+vec2(u,-u));
15     vec4 d = texture2D(tex0, co+vec2(u,u));
16     vec4 c = texture2D(tex0, co+vec2(-u,u));
17
18     gl_FragColor = (a+b+c+d)*0.25 + texture2D(tex1, co);
19 }
```

2.4.3 Smoothing

Inarguably the most important step of the multigrid algorithm is smoothing, as it is the one actually causing the method to converge to the desired solution. In section 1.1.3 the choice of ω-Jacobi as the solver used in for relaxation was made. For the model problem a single step of Jacobi approximation is then defined as

$$\begin{aligned} z(x,y) &= \frac{1}{4}[h^2 f(x,y) + \\ &\quad u_n(x-h,y) + u_n(x+h,y) + u_h(x,y-h) + u_h(x,y+h)] \\ u_{n+1}(x,y) &= u_n(x,y) + \omega[z(x,y) - u_n(x,y)] \end{aligned}$$

with $h = 1/n$. Translating this equation to shader code is fairly straightforward. The result is listed in 2.3. Note that h is passed to the fragment program in already squared form, as performing that multiplication for each pixel would be superfluous.

Listing 2.3: Fragment shader to perform one step of omega-jacobi relaxation for poisson equations.

```
// Textures:
// tex0 - the current estimate
// tex1 - the right side of the poisson equation

uniform sampler2D tex0;
uniform sampler2D tex1;

uniform float unit;
uniform float hsquare;
uniform float omega;

void main(void)
{
    vec2 co = vec2(gl_TexCoord[0]);
    vec4 top = texture2D(tex0, co+vec2(0.0,-unit));
    vec4 bottom = texture2D(tex0, co+vec2(0.0,unit));
    vec4 left = texture2D(tex0, co+vec2(-unit,0.0));
    vec4 right = texture2D(tex0, co+vec2(unit,0.0));
    vec4 center = texture2D(tex0, co);
    vec4 f = texture2D(tex1, co);

    vec4 z = 0.25 * (hsquare*f + left + right + top + bottom);
    gl_FragColor = center + omega*(z-center);
}
```

2.4.4 Residual Calculation

Calculating the residual or defect is central to the coarse grid correction concept. The defect equation for our model problem is given by

$$d_h = f_h - \Delta_h u_h$$

which expands to

$$\begin{aligned} d(x,y) &= f(x,y) - 1/h^2[4u_h(x,y) - u_h(x-h,y) - u_h(x+h,y) \\ &\quad -u_h(x,y-h) - u_h(x,y+h)] \end{aligned}$$

when fixing h and applying the five-point discretization to Δ. Again, transferring this equation to a fragment program is straightforward, the result is shown in listing 2.4. Note that the computational effort required for this step is very similar to that of the Jacobi implementation above.

Listing 2.4: GLSL code for residual calculation.

```
// Textures:
// tex0 - the current estimate
// tex1 - the right side of the poisson equation

uniform sampler2D tex0;
uniform sampler2D tex1;

uniform float unit;
uniform float hsquare;

void main(void)
{
    vec2 co = vec2(gl_TexCoord[0]);
    vec4 top = texture2D(tex0, co+vec2(0.0,-unit));
    vec4 bottom = texture2D(tex0, co+vec2(0.0,unit));
    vec4 left = texture2D(tex0, co+vec2(-unit,0.0));
    vec4 right = texture2D(tex0, co+vec2(unit,0.0));
    vec4 center = texture2D(tex0, co);
    vec4 f = texture2D(tex1, co);

    gl_FragColor = f - (4.0f * center
        - top - bottom - right - left) / hsquare;
}
```

2.4.5 Boundary Conditions

One reason for the Jacobi smoothing implementation shown above being quite simple is that it completely disregards boundary conditions. This is

possible because GPUs allow a variety of ways for indirectly dealing with them. It is also advantageous in terms of performance, because as SIMD architectures GPUs are most efficient when they can treat each node identically.

Homogeneous Dirichlet boundary conditions are implemented using the OpenGL `GL_TEXTURE_BORDER_COLOR` property. It can be employed to set the borders to the desired value. Then the texture wrapping behavior can be changed to `GL_CLAMP_TO_BORDER`, in effect causing all reads outside of the designated area to return the boundary value. To use these properties, two methods were added to the `GLRenderTexture` class:

setBorderColor(color) Changes the border color of the texture associated with this buffer to the supplied floating point value.

setClampToBorder() Sets both the horizontal and vertical clamping behavior to clamp to border.

Using this method, the grid size at level n must be set to $2^n - 1$. The border can be omitted, so starting from the coarsest level, grids of size 1^2, 3^2, 7^2, etc. will be treated.

Inhomogeneous Dirichlet boundary conditions can be simulated in a wider variety of ways.

- Causing the fragment program to conditionally omit pixels on the boundary, thus ensuring they stay constant. This requires undesirable branching in the pixel shader.
- Recreating the boundaries after each pass of the smoothing operator. Also not ideal, as it introduces additional drawing operations with their own overhead.
- Drawing only to the area of the target surface that actually should be changed. This allows keeping the shader identical across all fragments and does not require additional operations.

The final possibility is obviously preferable. It could intuitively be realized by just rendering a smaller quad. However, that would make the triangle rendering optimization described in section 2.2.2, which provides significant performance gains for small grids, impossible to realize.

A better way to implement this method was found in the OpenGL scissor rectangle. This antique OpenGL property allows the user to define a rectangular area outside of which all drawing will be suspended. It is enabled

by `GL_SCISSOR_TEST` and the area is defined via the `glScissor` function. To facilitate its use, only a small change in `GLFilterStep` was required, and the addition of one more method to `GLRenderTexture`:

setBorderPixel(n) Defines n pixels around the edges as an unchangeable border.

Testing has shown that enabling the scissor rectangle with a one-pixel border causes no perceptible performance difference. However, it is important to note that slightly larger grids have to be used compared to the homogeneous case. In particular, the grid side length at level n must be set to $2^n + 1$ to accommodate the additional border pixels, making the coarsest grid 3^2 in size.

Other types of boundary conditions have not been implemented in this work. However, homogeneous Neumann conditions should be fairly easy to implement on GPUs by adjusting the texture wrapping behavior. More complex variations require individual treatment of the edge cases – Goodnight et al. [7] have demonstrated some.

2.4.6 The Complete Multigrid Solver

All individual components of significance to the multigrid cycle have now been described. What remains is taking these parts and using them to form a correct numerical solver. Recalling the multigrid method described in 1.1.2, the process shown in figure 2.6 can be derived.

The first fact to note from the illustration is that, at each level, three buffers are required. This is caused by OpenGL disallowing – with good reason – rendering to a buffer that is also bound as an input texture. If interpolation and addition had not been combined, four buffers would have been required. However, due to the pre- and postsmoothing processes and the need to retain f, the right-hand side of the equation, during those, it is not possible to reduce the number below three regardless of optimizations.

The solver is implemented in the class `MGPoissonSolver`. It contains the following public methods:

MGPoissonSolver(*rightside, bordervalue, h, pre, post, omega)
The constructor of a solver for Poisson equations with homogeneous Dirichlet boundaries. It requires the right hand side of the equation, the boundary value and discretization width at the finest level – the latter determines the size of the calculation area. The other parameters are optional and determine the amount of pre- and

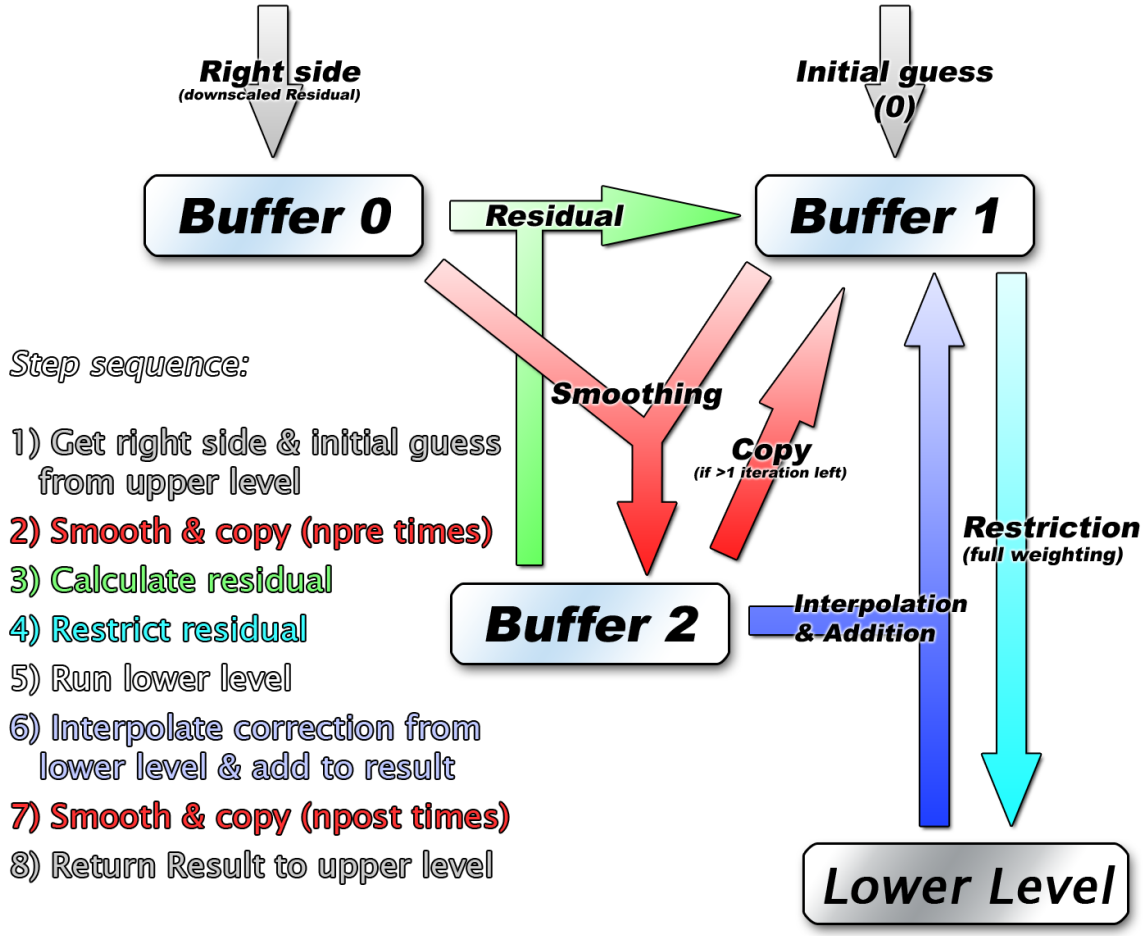

Figure 2.6: GPGPU Multigrid cycle.

postsmoothing steps and the relaxation parameter of the Jacobi smoother, respectively.

The constructor verifies that the parameters are correct, allocates and populates all required buffers and loads and compiles the fragment programs corresponding to the required operations. After it has finished the solver is ready to run.

~MGPoissonSolver() The destructor deletes all the allocated buffers and fragment programs, freeing GPU memory.

vCycle(level) Runs the process shown in figure 2.6 on the designated grid level. This method contains the main part of the algorithm. It sets up the correct shaders and buffers for all operations required and executes them in the correct order. Smaller grids are solved recursively unless the coarsest level has been reached.

run(iterations) Repeatedly calls `vCycle` at the finest level, thus executing a number of iterations of multigrid approximation. In most cases, the user will only need the constructor, this method and the following one.

GLRenderTexture* getResult() Returns the current result of the multigrid approximation. At the beginning, before any call to `run`, this equals the initial guess, which is chosen to be zero in the current implementation.

benchmark() Not part of the actual computational process, this method times the execution of each individual operation as well as the full V-cycle and reports the results. Most results shown in chapter 3 were obtained this way.

One element that would be useful in practice but has been omitted here because it is not relevant to *computational* performance analysis would be a run method that continues to iterate until the "error" has fallen below a certain threshold. This would involve accumulating the difference between the current and previous approximations every n iterations. On GPUs, this process itself would have to be iterative, as only a limited number of texture accesses can be carried out in one pixel shader.

2.5 Alternatives: CUDA

While a complete reimplementation based on CUDA, which was only publically released some time after programming on this thesis had already started,

would go beyond the scope of this work, it still merits some interest to look at the possibilities afforded by such an API. Most of this section applies to ATI's CTM as well, though it has been based on the CUDA technical documentation [5].

CUDA offers a C-like language with some extensions that allow the user to schedule programs for execution on GPU multiprocessors. The central advantages of this more direct approach compared to traditional GPGPU development with DirectX or OpenGL are as follows:

- More direct control over the destination of writing operations. This enables important stream processing operations like scatter. In practice, for multigrid, it would mean that – for example – interpolation could be efficiently implemented like on CPUs.
- Unrestricted access to all memory levels, including on-chip cache.
- A unified programming environment for both host (CPU) and client (GPU) code that requires no knowledge of graphics APIs.
- Elimination of graphics-related overheads.

However, at this point in time there are also a few drawbacks:

- Global memory accesses need to follow specific patterns to enable coalescing, otherwise severe performance reductions can be expected.
- Some hardware features accessible by graphics APIs can not be used via CUDA as of yet.
- No automatic caching of most memory read types.
- Unlike OpenGL and GLSL, not cross-vendor compatible.

It can be assumed that, for a Jacobi-based standard-coarsening method like the one examined in this work most enhancements provided by CUDA would be caused by the elimination of overheads. For other algorithms like GS-RB based solvers or problems with more complex boundary conditions CUDA can provide very significant improvements – even going so far as making previously unpractical algorithms, for example those that rely on scatter operations, feasible on GPUs.

Chapter 3

Performance Evaluation

As determined in Chapter 1, the prime motivator for implementing numerical methods on GPUs is performance. In this chapter, the performance of the GPGPU multigrid implementation described in chapter 2 will be examined. At first a quick explanation of how the measurements were obtained will be given, then the overall performance of the full algorithm and its individual components will be presented in detail for a single architecture.

After these basic values are established, benchmark data obtained from systems with a variety of GPUs will be compared, and some of the differences analyzed. Another comparative section will concern itself with the performance of the GPU implementation vis-à-vis a conventional CPU version of the same algorithm. Finally, venues for additional optimization uncovered by the gathered data will be explored.

3.1 Notes on Benchmarking

To facilitate gathering a wide range of results from various platforms, a standalone noninteractive benchmarking application was developed and released[1]. The test case used is an instance of the model problem with homogeneous Dirichlet boundary conditions, the V-cycles are configured to perform two steps of pre- and a single one of postsmoothing.

The application first runs a test to verify that correct results are obtained – this is required to exclude benchmarking data from platforms that do not perform the correct calculations, but do also not report an error, which is the case for some hardware/operating system/driver combinations. It then carries out some "warm-up" iterations that make sure that all required shaders

[1]The benchmark was made available in the GPGPU subsection of the Beyond3D hardware discussion forum, located at http://forum.beyond3d.com/.

are loaded and reduce driver and cache initialization impacts during the first real benchmarking iterations.

Finally, the following measurements are taken: for each of $2^{n+1} - 1$, n ranging from 1 to 10, the time to execute one step of ω-JAC, residual calculation, full weighting, interpolation & addition and a complete V-cycle is recorded in a log file. Here, finding a good balance between accuracy and runtime of the program was important, so the individual step benchmarks are performed $(11-n)\cdot 1000$ times, while the full V-cycle is tested for $(11-n)\cdot 100$ iterations. Before and after these loops, the time is taken using the Windows system API function `GetSystemTimeAsFileTime`. The final result is computed by dividing the time obtained by the number of iterations performed. This whole process is repeated three times for each data point.

3.2 Component Performance

Knowing the performance of the individual components of the algorithm allows us to judge more effectively which components may be optimized to significantly alter the overall runtime. Unless otherwise noted, all measurements in this section were obtained on the primary development system, comprised of an Athlon64 X2 CPU and NVIDIA GeForce 8800 GTS GPU.

Table 3.1 shows the values achieved by each component at different levels of discretization. They are given in time (in μs) per iteration of the operator in question. For V-cycle, this means a complete cycle through the given discretization, all coarser levels, and back up. There are a few relationships between these numbers that merit detailed examination, so they will each be discussed in an individual section.

Component	***63***	***127***	***255***	***511***	***1023***
ω-JAC	4.6	15.6	62.5	250.0	1045.3
Residual	67.2	67.2	92.2	335.9	1351.5
Full weighting	70.3	70.3	75.0	226.6	865.7
Half weighting	71.9	70.4	70.3	176.6	576.5
Injection	71.9	71.8	71.9	78.1	257.8
Interpolation & Add	67.2	65.7	65.6	225.0	857.9
VCycle	3937	4593	5360	6484	13078

Table 3.1: Performance of multigrid components. Numbers are μs per iteration.

3.2.1 Scaling with Problem Size

One very important factor in judging the efficiency of an implementation is its scaling behavior with regards to problem size. When giving an overview of the massively parallel GPU architecture in chapter 1 it was already noted how scaling problems should be expected at smaller problem sizes. Now that the numbers are available, it is time to examine whether this speculation holds.

Figure 3.1 shows the scaling behavior of each of the components measured. Values were normalized to each respective maximum at 1023^2. As the workload increases quadratically along the measurement sizes, the chart uses a logarithmic scale.

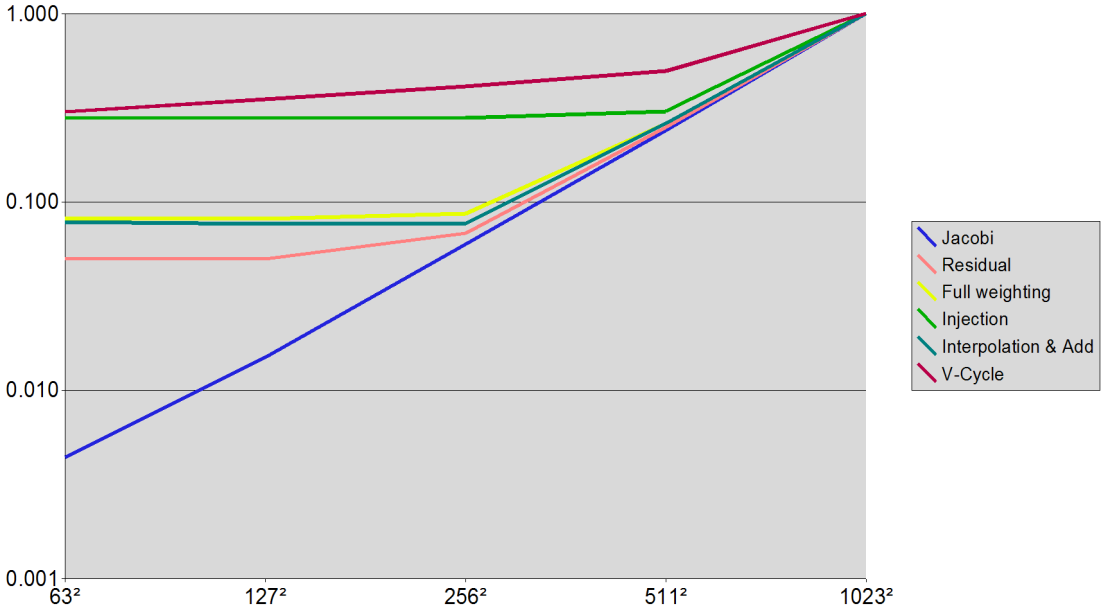

Figure 3.1: Scaling behavior of multigrid components.

From this representation the following observations can be made:

- All single components scale nearly perfectly from 1023^2 to 511^2. Most, except for injection – which is not used in practice – also scale adequately to 255^2.
- This, however, does not imply that the complete V-cycle also scales well at those sizes. The fact that it does not is easily explained by it requiring calculations at *all* granularities.
- Jacobi-based smoothing scales almost perfectly.

- On the other hand, residual calculation, a very similar process, all but stops scaling below 255^2. This warrants further investigation.
- Both transfer operations, restriction and interpolation, are completely limited by outside factors at 255^2 and below, as evidenced by their iterations taking nearly the same amount of time at that size as they do at 63^2 – where only 1/16 as much work has to be performed!

The positive aspect is shown by the Jacobi numbers: it **is** indeed possible to provide near-perfect scaling down to 63^2 at least. Now the essential question is why the other components do not exhibit the same behavior. But before that, the relations between the times measured for each component should be illuminated.

3.2.2 Expected Workload versus Measured Performance

To identify bottlenecks it is useful to compare the benchmark results of the individual methods with their expected relative performance based on workload. This is best done at 1023^2, where external limitations are marginalized. Figure 3.2 shows a visual comparison of the values achieved by the various operations at that size.

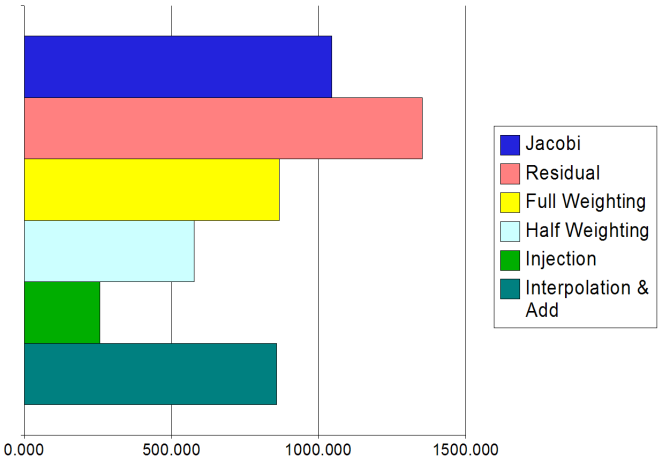

Figure 3.2: Comparison of component performance at 1023^2 nodes.

The most directly comparable results are Jacobi smoothing with residual calculation, and the three restriction methods amongst each other.

- The smoothing and residual calculation kernels each sample a five-point stencil from one texture, and a single point from another. They then

perform a comparable amount of additions and multiplications on the data. The only significant difference that could explain the variation in performance is a division by h^2 in the residual calculation shader. This will be examined in section 3.2.4.

- Full weighting, half weighting and injection are nice cases for performance analysis: they each perform nearly the same operation, but on a different amount of values. They all take a number of samples and then perform a weighted interpolation of those. Full weighting takes nine, half weighting five and injection one. Though injection is not nine times as fast as full weighting, the measurements fall in line with these expectations.

- The interpolation and addition step is slightly harder to judge, as there is no directly comparable operation. However, Jacobi smoothing comes reasonable close – both take some samples from one texture and a single one from another, and then perform arithmetic operations on them. Interpolation takes one less sample, but more importantly samples a smaller texture and should thus have a cache advantage. Therefore its performance level at around 22% faster than Jacobi iterations is acceptable, if a bit slower than expected.

On a whole, the relative results between components are more expected and easily explained than the individual scaling results shown previously.

3.2.3 V-cycles as Sums of Components

In our multigrid implementation, the workload for each grid element is constant regardless of grid level. Combined with the grid size reduction to $\frac{1}{4}$ caused by standard coarsening, this observation leads to the conclusion that at most $\frac{1}{3}$ of the time spent for one full V-cycle starting at level n should be spent at levels below n. More specifically, the portion of work at lower levels is given by the formula

$$\frac{\sum_{a=1}^{n-1}(2^a-1)^2}{(2^n-1)^2}$$

which converges to $\frac{1}{3}$ for $n \to \infty$, and reaches 0.33 for $n = 9$.

Using this approximation, we can determine how much inefficiencies at smaller grids influence the total runtime. One complete V-cycle in our example runs the following operations at the finest level: three Jacobi iterations, one residual calculation, and one each of full weighting and interpolation, the two transfer operations. With these values and the results gathered in table

3.1 the efficiency ratings shown in table 3.2 were derived. The "Component Sum" is determined by taking the single-step results of the operations listed above at the specified level and summing them up. "2/3 V-cycle time" are two thirds of the V-cycle at that level, as per the explanation in the previous paragraph. Figure 3.3 illustrates these results.

	63	***127***	***255***	***511***	***1023***
Component Sum	218.500	250.000	420.300	1537.500	6211.000
2/3 V-cycle time	2624.667	3062.000	3573.333	4322.667	8718.667
Efficiency	8.32%	8.16%	11.76%	35.57%	71.24%

Table 3.2: Efficiency of GPU multigrid implementation compared to theoretical optimum.

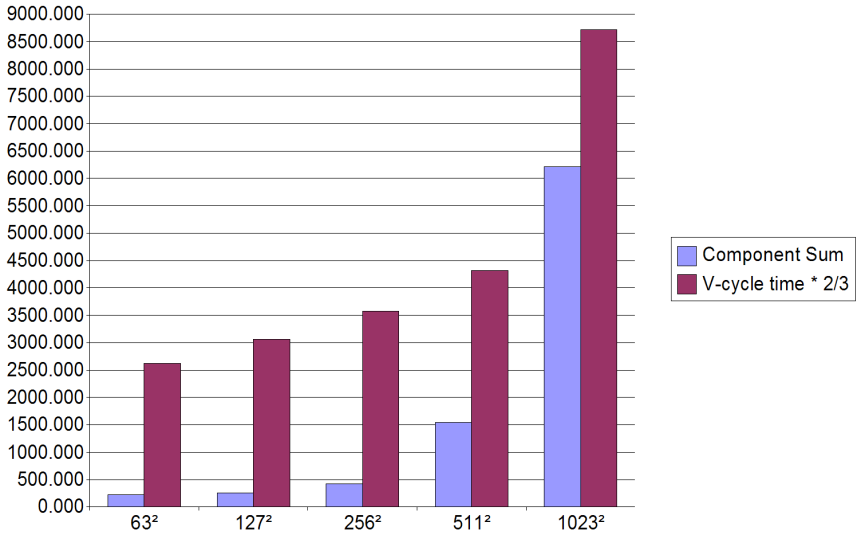

Figure 3.3: Ideal theoretical versus measured V-cycle performance.

While not entirely unexpected after the scaling behavior observed in section 3.2.1, these results are still disappointing. Though a high-performance implementation is usually required only for large problem sets, one could imagine an application where solving a large number of smaller equations fast is required. In that case, an efficiency of around 10% greatly detracts from the utility of a GPGPU implementation.

3.2.4 Optimizations Based on Component Benchmarks

In section 3.2.1 it was shown that, while Jacobi smoothing scales well to coarse grids, all other components exhibit low efficiency at grid sizes below 256^2. Shortly after, in the following section, we found that – while most operators behave as expected in terms of relative performance – residual calculation is slower than anticipated. In this section, an attempt will be made at reducing these problems.

As noted earlier, the only major difference in the shader programs for smoothing and residual calculation is that the latter contains a division operation. However, eliminating that devision by replacing the parameter `hsquare` with `rhsquare`, its reciprocal, did not change measured performance. Restructuring the code to be even more similar to the Jacobi kernel did also not alter the runtime behavior in a meaningful way.

Not satisfied with this result, a next step was to make the benchmarking conditions as equal as possible. The original measurements were taken using the same buffers as those used in the actual computation. This resulted Jacobi smoothing and residual calculation running in different directions each, that is, one's source was the other's target and the other way around. Equalizing this aspect resulted in Jacobi smoothing taking *longer*, while residual computation got faster. In the end, both were within 2 microseconds per iteration at each grid size – as expected from such similar operations in the first place.

How or why this happens is not completely clear. As *both* operation's times are affected by changing the buffer order of just one of them, one reason could be a bizarre cache effect. However, all such should be eliminated by the benchmark process, which preforms 10000 iterations of each operator. At this point, further investigation of this matter was postponed for two reasons: firstly, it would require intricate knowledge of the hardware and OpenGL driver behavior, and secondly, improving the scaling behavior of all operations except Jacobi smoothing was deemed more important.

Improving performance at coarse grid levels is an exercise in reducing overhead. The fundamental observation in this case was that, while Jacobi scores remained largely unaffected by the C++ program's compilation options[2], the time required to run those operations that did not scale well changed when switching compiler optimizations on and off. In other

[2]Specifically, the differences were observed using Visual C++ 2005 and switching between debug builds (no optimizations, debug symbols included) and release builds with full optimization.

words, their execution was CPU limited – and not in the driver, but in the program!

Based on this knowledge, the following changes were made:

- Operations that were used via helper objects like the transfer methods were re-integrated into the main method of the algorithm or performed via static member functions. While not as clean from a program design standpoint, this proved quite effective.
- The central `VCycle` method was completely rewritten, with each line examined as to its purpose and necessity.
- Two static methods were added to `GLFilterStep`: `direct(source, shader, target)` and `direct2s(s1, s2, shader, target)`. These allow the direct use of shaders from one or 2 buffers to another and forgo many of the requirements of the general method, like exhaustive error checking and variable argument lists.

The complete listing of the new results after these changes is gathered in table 3.3. The changes were very effective at improving performance at coarser grid levels, as shown by table 3.4. Note that the unused restriction methods were no longer benchmarked, and that an additional coarsest level of 31^2 grid points was added to the test run.

Component	***31***	***63***	***127***	***255***	***511***	***1023***
ω-JAC	3.0	3.1	14.1	60.9	243.7	967.2
Residual	3.1	4.7	17.2	78.1	314.1	1259.4
Full weighting	3.1	3.2	14.1	57.9	240.6	964.1
Interpolation & Add	3.2	3.1	12.5	54.6	220.3	881.2
VCycle	2203	2672	3203	3813	4782	14022

Table 3.3: Performance of multigrid components after overhead reduction. Numbers are μs per iteration.

	63	***127***	***255***	***511***	***1023***
Before Optimizations	3985	4627	5537	6601	14178
After Optimizations	2672	3203	3813	4782	14022
Improvement	49,14%	44,46%	45,21%	38,04%	1,11%

Table 3.4: Improvements to overall V-cycle times due to overhead reduction.

Charting the scaling behavior on a logarithmic scale like in section 3.2.1 shows a far more agreeable picture than before. As depicted in figure 3.4, all

operators now scale near optimally down to 63^2, the coarsest level measured previously. From there on, workload does no longer seem to matter and the limitations lie elsewhere again. What could be done to further increase performance and decrease the impact of these remaining inefficiencies is discussed in section 3.6. These limitations at the very lowest levels, together with some fixed switching costs that are hard to remove, render the complete V-cycle – while certainly faster than before – not nearly as enhanced as the sum of its parts.

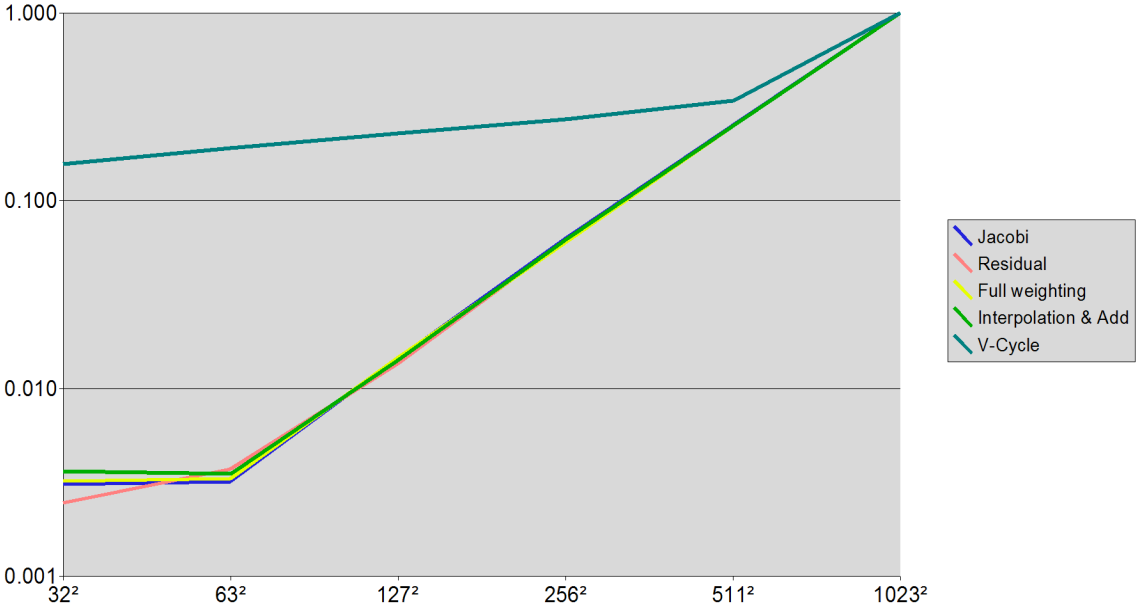

Figure 3.4: Scaling behavior of multigrid components.

3.3 Comparison Among GPUs

The main testing system used throughout this work contains an NVIDIA G80 GPU. While that is a good baseline for optimization-related and comparative performance analysis, one of the greatest advantages of using OpenGL and GLSL for the implementation is that it is capable of running on a wide variety of graphics solutions. In this section, results gathered from different GPUs will be compared.

Before starting to present numbers, recall the testing methodology described in section 3.1. It was used to obtain all the results forthcoming. When more than one set of results from comparable platforms was available, the median results are used in the following analysis.

In table 3.5 all the V-cycle results are summarized. Obviously wrong[3] results are excluded and marked as "Wrong". Results marked as "Failed" on the other hand signify that the driver produced some error or that the system crashed. Also, the very small sizes, 31^2 and below, should be mostly disregarded here, as they are more influenced by CPU speed and driver overhead than GPU performance and have little practical relevancy. Note that only G80-derived GPUs could complete the benchmark at the finest tested grids of 2047^2 nodes. This is most likely due to a driver or GPU addressing limitation and not memory capacity problems, as even 8800 cards with only 320 MB of on-board memory were able to complete this test, while other boards equipped with 512 MB or more failed to do so.

Based on these numbers a multitude of observations can be made. Some particular aspects will be discussed in the following subsections. When it is more practical to compare the GPUs at only one size, 511^2 or 1023^2 will be used. At those dimensions, most systems are already limited by GPU performance rather than driver and CPU overhead, but they could still be completed by most tested cards.

3.3.1 Vendor-specific GPU Progression

In figure 3.5 the performance of various NVIDIA GPUs is illustrated. Results obtained running the benchmark on 8000-series GPUs in Windows Vista are excluded, as they seem incomparable to XP results – most likely due to driver problems.

Looking at the remaining results and at comparable GPUs, the performance improvement from the 6000- to the 7000-series is slightly above factor two, while going from 7000 to 8000 shows a nearly fivefold increase. This result is impressive but not unexpected: many of the architectural changes made to G80, as detailed in [4], are very well suited to GPGPU processing.

The situation on the corresponding ATI chart shown in figure 3.6 is more complex. While the scores for the 1000-series cards follow expectations[4], the 2900 XT results fall far short. The most likely reason for this behavior are immature drivers: the 2900 XT series was released very recently, so the drivers may still be missing optimizations for rare use cases, like those presented by this multigrid benchmark.

[3] Some ATI drivers – the 8.37 series specifically – reported results of less than $100\mu s$ at 1023^2 and beyond, while not performing any work at all. Also, some cards reported faulty numbers at 2047^2.

[4] Based on the relative amount of computational resources and bandwidth available. One comprehensive resource for such information are the *Beyond3D GPU tables* available at `http://www.beyond3d.com/resources/`.

GPU (clocks) & OS	***3***	***7***	***15***	***31***	***63***
NV 6800 GT XP	797.0	1354.4	1875.0	2522.9	3125.0
NV 7600 GT XP	609.4	1024.3	1484.4	1875.0	2395.8
NV 7900 GTX Vista64	2819.0	4681.1	6608.8	8515.7	10383.3
NV 8600 GTS (720/1050) Vista	6443.0	10850.0	15093.8	19545.7	24336.7
NV 8800 GTS 640 (525/830) XP	1532.0	2483.3	3437.5	4375.7	5363.3
NV 8800 GTX Vista	6391.0	10660.0	15605.0	19732.9	24350.0
NV 8800 GTX XP	391.0	660.0	937.5	1205.7	1485.0
NV 8800 GTX (670/1050) XP	344.0	590.0	801.3	1048.6	1301.7
ATI 1600 Mobile XP	640.6	1059.1	1464.9	1942.0	2708.4
ATI 1600 XT XP	562.0	937.8	1347.5	1718.6	2318.3
ATI 1900 GT XP	421.9	694.4	996.1	1294.6	1666.7
ATI 1900 XT Vista	334.0	543.6	753.1	965.3	1178.3
ATI 1950 XT XP	656.0	1076.7	1523.8	1941.4	2448.3
ATI 1950 XTX CF XP	375.0	607.8	858.8	1115.7	1406.7
ATI 2900 XT XP (Cat 7.5)	328.1	555.6	761.7	959.8	1171.9
ATI 2900 XT Vista	446.0	725.6	1020.0	1290.0	1598.3
ATI 2900 XT Vista64 (8.38)	245.0	401.1	552.5	714.3	861.7
ATI 2900 XT Vista64 (8.39)	827.0	1560.0	2301.3	3075.7	3821.7
GPU (clocks) & OS	***127***	***255***	***511***	***1023***	***2047***
NV 6800 GT XP	3938.0	8400.0	35520.0	165470.0	Failed
NV 7600 GT XP	3031.3	5820.3	23177.1	114453.0	Failed
NV 7900 GTX Vista64	12372.0	15917.5	26433.3	76145.0	Failed
NV 8600 GTS (720/1050) Vista	29952.0	40170.0	70200.0	191490.0	Failed
NV 8800 GTS 640 (525/830) XP	6312.0	7617.5	10106.7	24765.0	154220.0
NV 8800 GTX Vista	28718.0	34882.5	44686.7	82970.0	236410.0
NV 8800 GTX XP	1782.0	2227.5	3856.7	15855.0	94850.0
NV 8800 GTX (670/1050) XP	1562.0	1952.5	3540.0	14530.0	86100.0
ATI 1600 Mobile XP	5281.4	19336.4	84012.6	Failed	Failed
ATI 1600 XT XP	3844.0	13672.5	40676.7	135470.0	Failed
ATI 1900 GT XP	2218.8	5117.2	10729.2	Failed	Failed
ATI 1900 XT Vista	2210.8	9594.1	38747.5	154385.0	Wrong
ATI 1950 XT XP	3220.0	4805.0	11250.0	30155.0	Failed
ATI 1950 XTX CF XP	2000.0	4647.5	18803.3	98205.0	Failed
ATI 2900 XT XP (Cat 7.5)	1937.5	5546.9	31041.7	128047.0	Wrong
ATI 2900 XT Vista	1894.0	4380.0	22086.7	89600.0	Failed
ATI 2900 XT Vista64 (8.38)	1056.0	4525.0	22576.7	89760.0	Wrong
ATI 2900 XT Vista64 (8.39)	4556.0	8852.5	32863.3	129405.0	Wrong

Table 3.5: Mean V-cycle times for various GPUs.

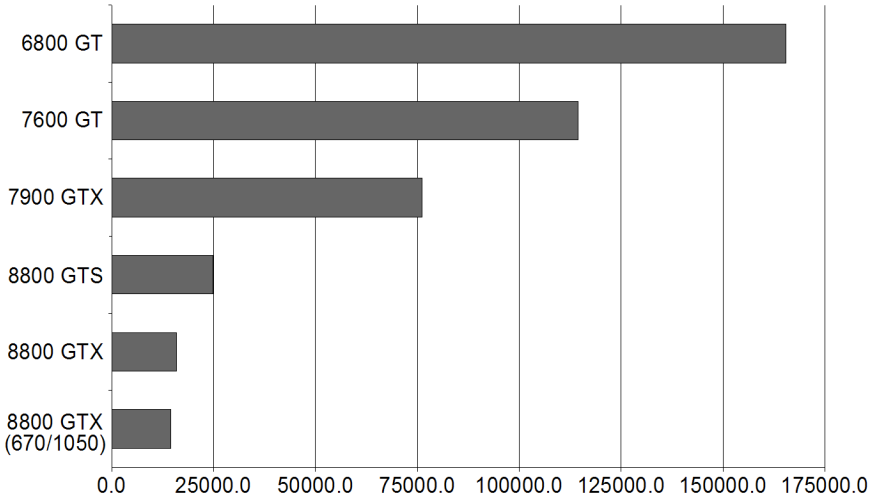

Figure 3.5: Scaling over generations of NV GPUs. Numbers are μs per V-cycle at 1023^2.

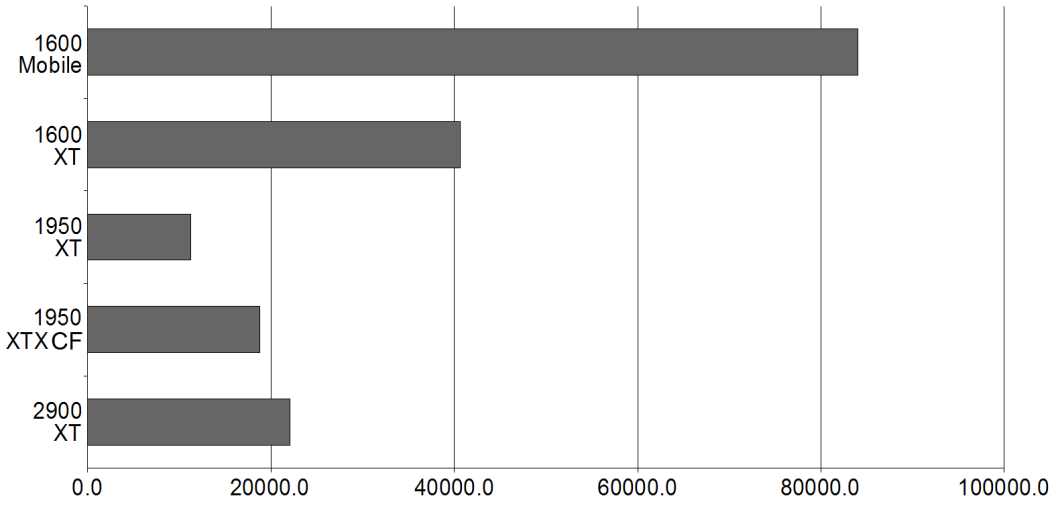

Figure 3.6: Progression of ATI GPUs. Numbers are μs per V-cycle at 511^2.

Another slightly disappointing data point from the graph is that CrossFire – an ATI technology to use two graphics cards in conjunction for rendering – not only fails to increase performance, but apparently reduces it. This is not too surprising, because the techniques used to split work between two GPUs are probably not designed to work with fairly small FP32 non-framebuffer rendertargets.

3.3.2 Operating System and Driver Influence

One interesting, repeatable pattern discovered during testing was that operating system and driver versions often had a profound impact on the performance of otherwise identical hardware. While some correlations could be expected, their magnitude is quite surprising. Figure 3.7 shows some of the largest discrepancies.

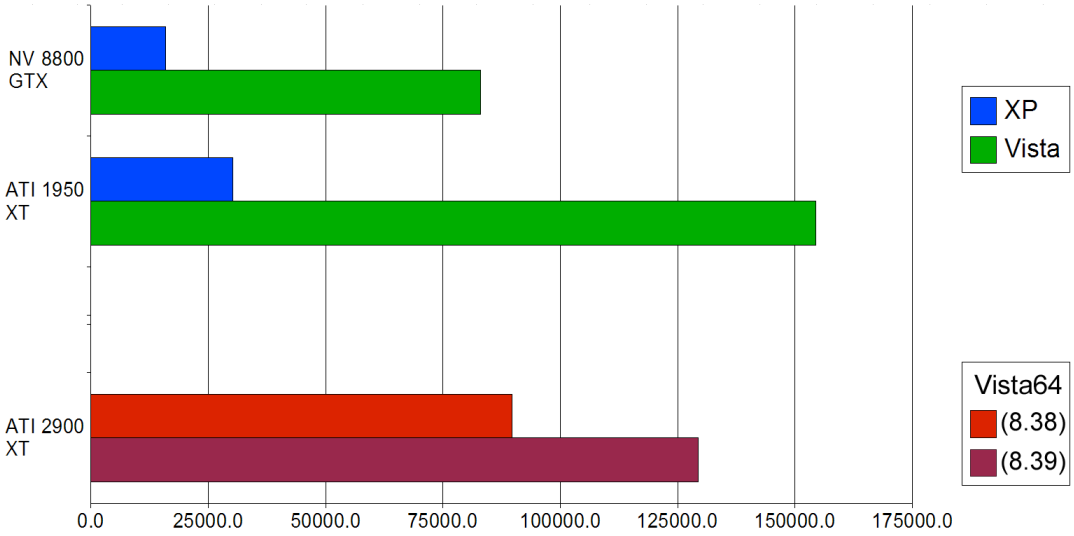

Figure 3.7: Driver and OS dependency of results. Numbers are μs per V-cycle at 511^2.

The results can be summarized as follows:

- On nearly all test systems, results for the same GPU were better on Windows XP than Windows Vista.
- The difference was particularly pronounced for NVIDIA 8-series cards and ATI cards *before* the HD 2900 XT.

- The performance of the HD 2900 XT varied widely between even minor driver revisions. As speculated in section 3.3.1, one likely reason for this are immature drivers for such a recently released architecture.

The Vista results are quite bad across the board – this can be attributed to vendors having difficulty adapting their drivers to the new model. For NVIDIA 8800 GTX cards the difference even reaches factor 8. The only conclusion to draw from this is that, at the moment, Windows XP is a far more viable system for OpenGL-based GPGPU efforts than Vista.

3.3.3 Cross-Vendor Comparison

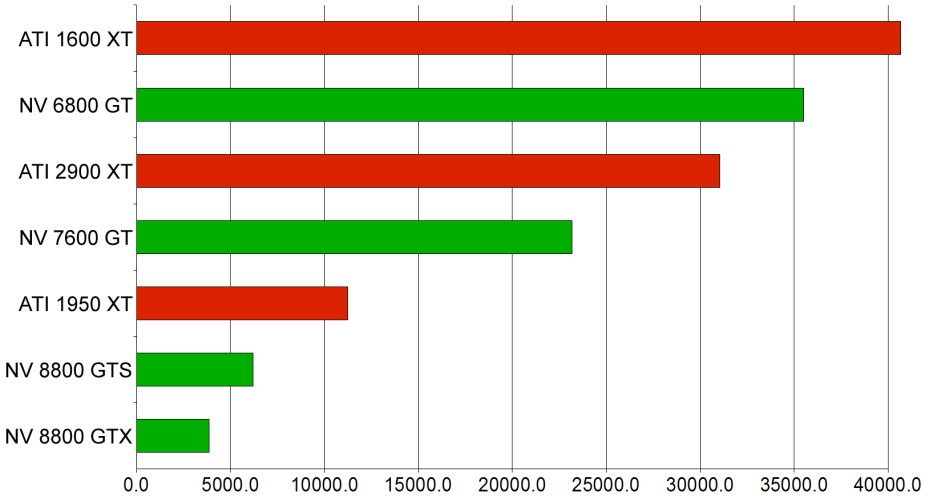

Figure 3.8: Results of cards by both vendors, sorted by speed. Numbers are μs per V-cycle at 511^2.

After examining each individual vendor's performance evolution over time and the additional factors introduced by OS and driver choice it is now time to compare the results of GPUs by both ATI and NVIDIA directly. Because of the findings presented in the previous section, only results obtained running in Windows XP are used for this comparison. Figure 3.8 illustrates the most salient of those.

While the competition was fierce in the previous generation of GPUs, currently the G80-based cards are out of reach for any other solutions. This may still change with future ATI drivers for the 2900 XT, but a complete turnaround seems very unlikely. For now the only recommendation for OpenGL-based multigrid processing that can be made based on these results are NVIDIA 8800-series cards.

3.4 CPU ↔ GPU comparison

GPGPU implementations are in nearly all cases more complex to realize than CPU programs. In chapter 1 it was determined that the main reason to still implement them is an expectation of higher performance for massively parallel streaming processes. In this section, it will be examined whether the results follow this speculation.

Before presenting the numbers, there are some important facts to note to put this comparison into perspective:

- The test system is an Athlon64 X2 4400+ with a GeForce 8800 GTS.
- The CPU program is a standard C implementation with full compiler optimizations, compiled with Visual C++ 2005. No SIMD instructions or threading are used. On the given CPU, those could be expected to provide a performance boost of about factor 4, if there are no memory bandwidth limitations before that.
- As explained in section 2.1, as of yet, the GPU version uses RGBA buffers – in practice doing nearly 4 times as much work as required.

After these facts have been established, here are the results. Table 3.6 shows a comparison of the results of both implementations at a problem size of 511^2 nodes at the finest grid.

Component	***CPU***	***GPU***
ω-JAC	8438.0	243.7
Residual	7656.0	314.1
Full weighting	1906.0	240.6
Interpolation & Add	6391.0	220.3
VCycle	51642	4782

Table 3.6: Comparison of CPU and GPU solver at 511^2.

Obviously the GPU implementation is much faster in this specific case. However, there are a few interesting facts to note beside that result:

- The CPU full weighting implementation uses the decomposition optimization impossible on GPUs, giving it a distinct performance advantage in the comparison vis-à-vis the other components.
- While the GPU is on average about 25 times as fast at running the components of the algorithm at the full size grid, in real application –

running a full V-cycle – that advantage is reduced to factor 10. This can most likely be attributed to the GPU inefficiencies at smaller grid sizes identified earlier.

- In section 3.2.3 it was observed that on the GPU, summing up the component times at the highest grid level does not even come close to reaching the expected $\frac{2}{3}$ of the full V-cycle time. On the CPU, that sum is 41267 while the expected value is 36681. Not only does the CPU version not have inefficiencies at coarse grids, it actually gets more efficient. This is explained by cache effects – smaller grids are more likely to fit into the L2 cache.

These points lead to the suspicion that the GPU implementation's advantage will quickly dwindle at smaller problem sizes. Indeed, as table 3.7 illustrates, the GPU version only starts to have the advantage at 255^2 and beyond. At smaller sizes, the CPU is significantly faster. The scale of the difference, both at the small and large problem sizes is shown graphically in figure 3.9. This is not as bad as it may appear – usually, a high performance implementation is only needed for sizable problems.

Component	***CPU***	***GPU***
3^2	3.0	750.0
7^2	7.0	1188.0
15^2	25.0	1703.0
31^2	56.3	2187.0
63^2	204.3	2672.0
127^2	1181.0	3203.0
255^2	7425.0	3813.0
511^2	51889.4	4782.0
1023^2	370160.0	14022.0

Table 3.7: CPU and GPU performance at different problem sizes.

The very small sizes below 32^2 were not included for their practical relevancy on their own, but rather because they are still required as part of the solvers at finer grids. This motivates an optimization that will be discussed in the following section.

Taking into account the results shown in table 3.7, it can be concluded that – for the specific system tested – GPU implementations only really start to make sense at problem sizes of 512^2 and beyond. However, at these fine grid levels the difference is very significant, ranging from factor 10 to 25 for a full V-cycle. It is hard to imagine such a difference being overcome even by

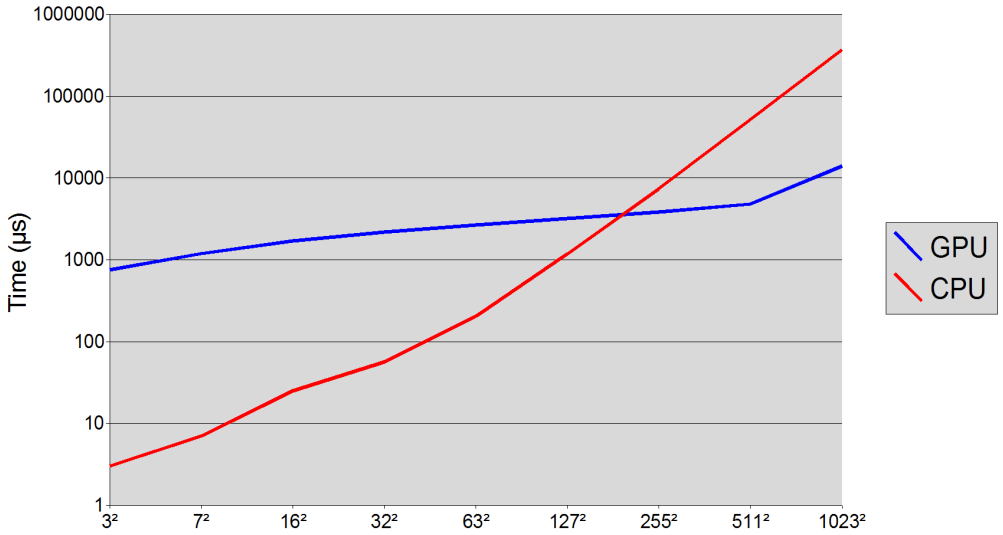

Figure 3.9: Logarithmic illustration of CPU and GPU solver scaling behavior.

the most optimized and multi-core capable CPU program. Of course these specific results are only valid for the tested system. For other CPU/GPU combinations the exact point where the GPU gains the advantage may well vary, but the overall trends would be similar.

3.5 CPU/GPU Combined Solving

The idea of using both CPU and GPU in combination to run a multigrid algorithm has been explored before [9], however, in that case a parallel solver was considered. The results presented in the previous section, in particular figure 3.9, motivate a different approach: instead of solving different problems or subdomains of the same problem on GPU and CPU, it should prove more advantageous to solve finer grid levels on the GPU and coarser ones on the CPU.

While this will introduce an additional speed penalty for transfers and synchronizations, the former should be quite small at coarse enough grids, where, for example, only 32^2 grid points need to be transferred – 16MB of data. At the same time, as per table 3.7, there are still significant gains to be made by moving those very small levels to the CPU. Figure 3.10 illustrates the concept.

The idea is to run the algorithm as usual until some specific level – the *switching point* – is reached. Then, use the data transfer methods described in section 2.3.1 to transfer the current state of the approximation to the

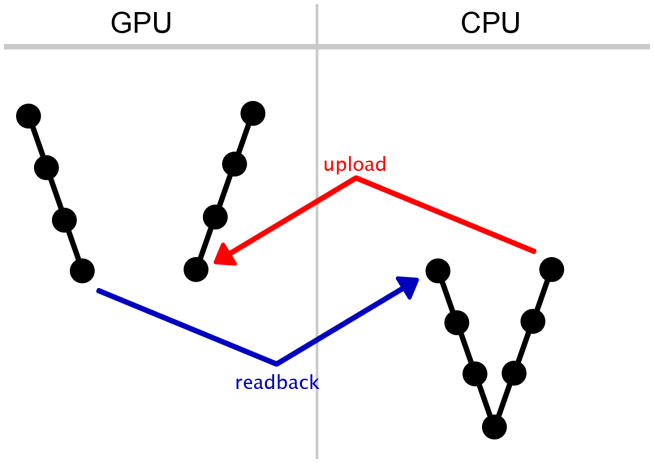

Figure 3.10: GPU and CPU combined V-cycle.

CPU. The coarser grid levels down to 1x1 and back up to the switching point are then calculated by the CPU-based solver. Finally, the result of that operation is written back to the GPU.

Now, there are two central questions about this theoretical approach. Firstly, will the performance gained by reducing the impact of GPU overhead at low levels be enough to result in a net gain when factoring in the added transfer costs? And secondly, how should the switching point be selected? This latter question is actually a simple optimization problem. Figure 3.11 depicts an idealized view of the situation. Obviously, optimal switching would occur at the point where the CPU starts to outperform the GPU.

Both of these questions can be answered by benchmarks. To that end, an experimental mixed solver was created and tested. Table 3.8 shows the results gathered from this. The table lists the time required for one V-cycle of a 511^2 problem when switching to the CPU implementation at various grid sizes. Figure 3.12 illustrates these results, and includes a comparison to the GPU-only implementation.

One positive aspect of these results can be identified immediately: at most switching points, the combined implementation is faster than the pure GPU solver. The general shape of the curve is also following expectations. If the CPU is used too soon at large grid levels, the performance advantage is small or even negative, and the transfer costs are big. On the other hand, if the switching is performed too late the full potential of the method can not be realized.

In this particular case, the ideal switching point is 15^2. Using it, the combined GPU/CPU implementation achieves a speedup of about 30%. While

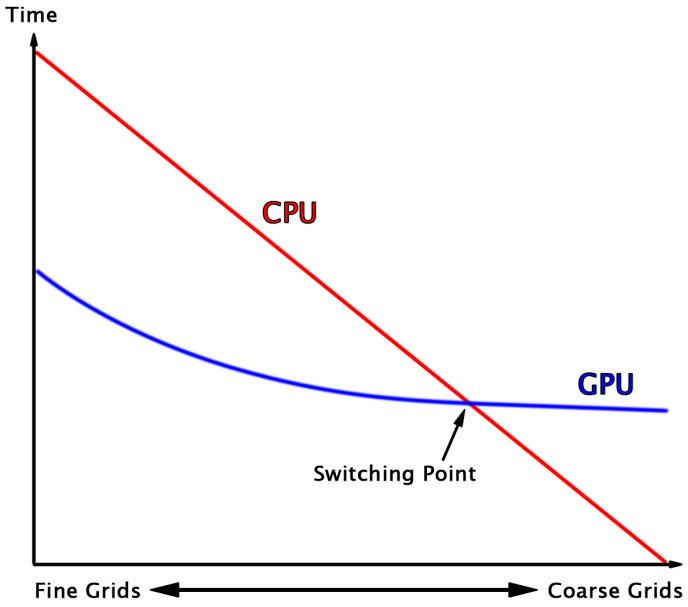

Figure 3.11: Selecting a good switching point for the combined implementation.

Switching point	*Time*
255^2	14188.0
127^2	5125.0
63^2	4343.0
31^2	4187.0
15^2	3703.0
7^2	3937.0
3^2	4172.0

Table 3.8: Combined CPU/GPU solver performance (in μs per cycle) at 511^2, for various switching points.

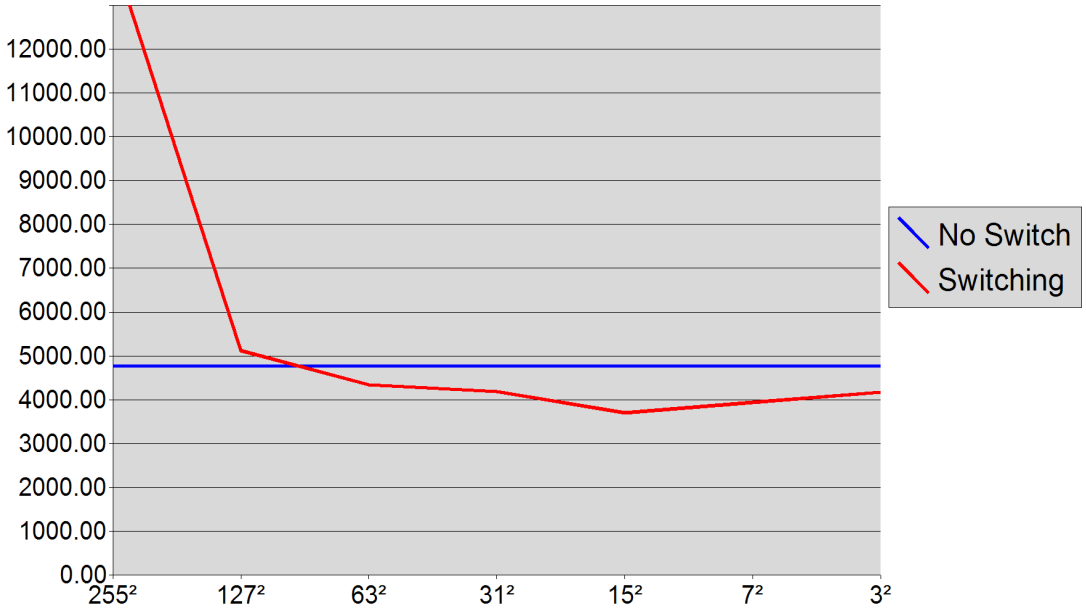

Figure 3.12: Combined CPU/GPU solver performance compared to GPU-only solver.

this result is not earth-shattering it is still significant enough a gain to demonstrate that combined CPU and GPU implementations of algorithms, where each processing unit performs the tasks most suited to it, are a useful technique. With the future adoption of APIs such as CUDA their importance may be reduced, but there will always be parts in many algorithms that are inherently unsuited to massively parallel processing.

3.6 Possibilities for Further Optimization

So far, a number of optimizations were performed, most of them aiming specifically at improving performance at coarse grid levels.

- The first improvement discussed in section 2.2.2 was switching from pBuffers to framebuffer objects. This resulted in a speedup of up up to 55%.
- Shortly after, the switch from quad-based to triangle-based rendering was performed, causing another 20% gain at small grid sizes.
- After still measuring bad scaling behavior for some operations in section 3.1, many overhead-reducing changes were implemented, improving performance by around 40% on average for full V-cycles.

- Finally, another improvement by 30% was achieved by implementing a combined GPU/CPU solver.

While this is a sizable number of improvements, there are probably still some significant gains to be made by further optimizations. The most important being the following:

- Switching to single-component rendertargets once OpenGL and GLSL provide support for them. This change should be very simple, and has to potential to improve performance by up to a factor of 4 on modern GPUs. However, in practice, the advantage will most likely only come close to that theoretical maximum at problem sizes of 1023^2 and greater. An alternative for some applications would be solving 4 similar problems at the same time using the current implementation.

- Looking back at the illustration of the GPU multigrid process in figure 2.6, there is a small inefficiency: when performing more than one step of pre- or postsmoothing, the result has to be copied back to the original buffer. As a configuration of two pre- and a single postsmoothing step was used throughout this work, this was deemed not very significant. However, if more smoothing steps are to be used it would make sense – and be perfectly possible – to eliminate any copying by performing a bit more housekeeping and adapting the calculation process to the number of steps required.

Chapter 4

Future Research

Like many GPGPU efforts, the work presented here is only a beginning. There are two main avenues for future developments: one, making use of the advances in GPGPU-related technologies, like new hardware features or APIs, and two, implementing related and extended numerical algorithms. Both of these approaches will be discussed in this chapter.

4.1 GPGPU Advances

As outlined in section 1.2, the art and science of using GPUs for non-graphics purposes has both advanced and changed greatly over the past 5 years. These changes are far from over, and we expect many future advances to benefit the field of numeric processing in general, and multigrid methods in particular. What follows is a summary of some of the relevant changes that are likely to happen over the next few years.

- Better support for unlinked single-component floating point computation from both vendors' hardware and in the APIs. This will enable up to fourfold performance increases without any significant changes to the computational process and boundary condition handling.
- Double-precision floating point arithmetic will be introduced, though probably with a significant performance penalty. Still, this will open up a wide field of applications that depend on higher than 32 bits of accuracy to GPGPU solutions.
- Increased significance, usage and performance of APIs like NVIDIA CUDA and ATI CTM will enable more flexible memory access and a unified programming model.

- Further improvements to branching performance will allow a wider variety of algorithms to be processed on GPUs, and enable new optimization options for existing algorithms.

- Continued increases in parallelization and performance. Due to their massively parallel architecture, it is comparatively easier to increase GPU performance compared to CPUs. Recently, NVIDIA announced[1] that the G92 architecture (the high-end successor to the current G80) will provide 1 Teraflop of computing power.

- Larger amounts of on-chip memory for caching or direct use by GPGPU APIs.

Looking at this list it would be hardly surprising for a CUDA implementation of a multigrid solver running on G92 to be able to solve systems of sizes of 2047^2 and above in real-time – about a year from now.

4.2 Related and Extended Algorithms

Besides perusing the advanced features and performance of future graphics platforms, a second way to build upon the work done in this thesis would be to implement other forms of multigrid or related PDE solvers on GPUs.

- One rather trivial to implement but potent improvement would be using *Full Multigrid* instead of the simple V-cycles of the current implementation. This method computes the initial estimate for the solver by using successive V-cycles at increasingly fine levels, and interpolating the result. It thus – in most cases – requires fewer iterations to reach the desired degree of accuracy. However, the actual numerical workload is very similar, so the method was not employed for this study.

- With better branching support and new APIs, implementing smoothers other than ω-JAC may prove advantageous. For example, GS-RB has a significant theoretical performance advantage.

- Adding support for a wider variety of boundary conditions would complicate the solver, but also make it applicable to more real-world problems.

- Solving three-dimensional systems of equations using a multigrid method would be another interesting expansion. While the limits on

[1] http://www.theinquirer.net/default.aspx?article=39829

data output in pixel shaders made working with 3D datasets cumbersome, CUDA partly alleviates such issues. For large datasets, the limited amount of on-board memory may pose a problem though.

Obviously, there is a wealth of research topics still available in the field of implementing high-performance multigrid-derived solvers on GPUs.

Bibliography

[1] J. Bolz, I. Farmer, E. Grinspun, and P. Schroeder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22:917–924, July 2003.

[2] W.L. Briggs, V.E. Henson, and S.F. McCormick. A multigrid tutorial. Society for Industrial and Applied Mathematics, 2000.

[3] NVIDIA Corporation. *Cg Toolkit User's Manual*, 1.2 edition, 2004.

[4] NVIDIA Corporation. *NVIDIA GeForce 8800 GPU Architecture Overview*, 2006.

[5] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture – Programming Guide*, 0.8.2 edition, 2006–2007.

[6] D. Goeddeke. Gpgpu::fast transfer tutorial. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial3.html, 2006.

[7] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. Siggraph 2003, In Proceedings of SIGGRAPH 102-111.

[8] S. Green. The opengl framebuffer object extension. NVIDIA Corporation, GDC 2005.

[9] S. Howard. Parallel multigrid solving using programmable graphics hardware. www.mit.edu/~showard/6.338/final.pdf, 2004.

[10] Advanced Micro Devices Inc. *ATI CTM Guide – Technical Reference Manual*, 1.1 edition, 2006.

[11] J. Juliano et al. *EXT framebuffer object Extension Specification*. OpenGL ARB SuperBuffers Working Group, 109 edition, 2003–2005.

[12] J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL Shading Language*. 3D Labs, 59 edition, 2004.

[13] D. Kirkland, B. Podder, and S. Urquhart. *WGL ARB pbuffer Extension Specification*. The Architectural Review Board, 1.1 edition, 1999–2002.

[14] E. Lindholm, M. Kilgard, and H. Moreton. A user-programmable vertex engine. Siggraph 2001, In Proceedings of SIGGRAPH 149-158.

[15] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[16] U. Ruede. The multigrid workbench. http://www.mgnet.org/mgnet/tutorials/xwb/xwb.html, 1995.

[17] P. Thoman. Hardware accelerated image processing – theory, 2004.

[18] P. Thoman. Hardware accelerated image processing – practise, 2005.

[19] U. Trottenberg, C.W. Oosterlee, and A. Schueller. *Multigrid*. Academic Press, 2001.