NTNU
Norwegian University of
Science and Technology

Matrix-Free Conjugate Gradient Methods for Finite Element Simulations on GPUs

Runar Heggelien Refsnæs

Master of Science in Physics and Mathematics
Submission date: June 2010
Supervisor: Trond Kvamsdal, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

Develop and implement matrix-free conjugate gradient methods applicable for solving Poisson problems by the finite element method on GPUs.

Assignment given: 01. February 2010
Supervisor: Trond Kvamsdal, MATH

Abstract

A block-structured approach for solving 2-dimensional finite element approximations of the Poisson equation on graphics processing units(GPUs) is developed. Linear triangular elements are used, and a matrix-free version of the conjugate gradient method is utilized for solving test problems with over 30 million elements. A speedup of 24 is achieved on a NVIDIA Tesla C1060 GPU when compared to a serial CPU version of the same solution approach, and a comparison is made with previous GPU implementations of the same problem.

Preface

This master thesis is written as part of the Master of Science study program in Applied Mathematics at the Department of Mathematical Sciences at NTNU, and is the result of work done in the spring of 2010 at the HPC laboratory of the Department of Computer and Information Science.

I would like to thank my supervisor Trond Kvamsdal for his guidance in our weekly meetings and for his valuable help in completing this project. Anne C. Elster, the head of the HPC-lab, has been kind enough to have me as an honorary member of the lab, allowing me to use their excellent equipment and participate as an equal member in their activities, including traveling to the Super Computing convention in Portland, Oregon november last year.

A special thanks goes to the master students Aleksander Gjermundsen, Ahmed Aqrawi, Holger Ludvigsen and Øystein Krog at the HPC-lab, who have helped me immensly with whatever technical difficulties I have encountered, and for providing good company and many laughs this last year. Finally I would like to thank my friend and fellow math-student, Gagandeep Singh, for the collaberation last fall on the project leading up to this master.

Trondheim, June 25, 2010

Runar Heggelien Refsnæs

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the technical report "The landscape of Parallel Computing Research: A View from Berkeley" [1] from 2006, a Berkeley research group predicts that processor architecture must shift from multicore technology to *manycore*, utilizing 1000s of cores per chip instead of the single digit numbers we see today.

To allow applications to draw from the full potential of the added computational capabilities, this shift will demand new ways of thinking and programming for parallel processors. The gap between applications and hardware must be filled with new programming models and tools to evaluate the success of applications. In the report, the problem is illustrated with figure 1.1, inspired by the view of the Golden Gate Bridge from the University of California at Berkeley.

The article from Berkeley was published in december 2006, and only months later NVIDA released the first iteration of a new general purpuse SDK for development on NVIDIA GPUs, the Compute Unified Device Architecture, or CUDA.

CUDA tries to answer many of the challenges proposed in [1], such as a programming model independent of the number of processors, minimizing remote accesses, and balancing the opacity of the underlying architecture, while keeping visible the key elements vital for performance. GPUs have several of the hardware properties that are mentioned in [1], and figure 1.2 illustrates the floating-point performance that can be gained compared to traditional processor design.

This thesis seeks to bridge the gap between the two towers in figure 1.1, represented by Finite Element Analysis as the application in the left tower, and

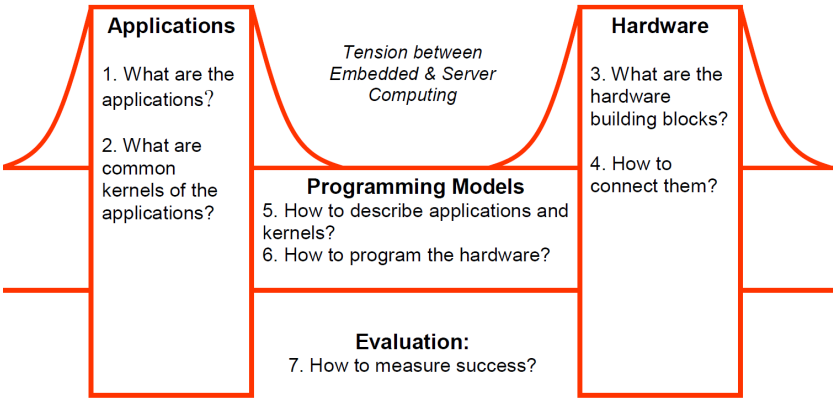


Figure 1.1: A view from Berkeley: seven critical questions for 21st Century parallel computing. The picture is taken from [1].

NVIDIA GPUs as the specific hardware in the tower on the right. Our *bridge* is a highly structured matrix-free approach to the conjugate gradient method.

1.2.

1.1 Report Outline

The rest of the report is structured as follows:

Chapter 2 presents general purpose programming on GPUs, with special attention to NVIDIA GPUs and the CUDA architecture and development tools.

Chapter 3 gives the mathematical basis for the finite element formulation, and the test problems used in the thesis.

Chapter 4 describes the conjugate gradient method used for solving the discretized finite element system, and explains the matrix-free version we have utilized.

Chapter 5 provides details on the block-structured approach to domain decomposition, and introduces our 2-dimensional take on the spline patch topology from the ICADA framework provided by SINTEF.

Chapter 6 presents the details of our implementation, and explains the design choices and optimizations done to make the program suitable for the ar-

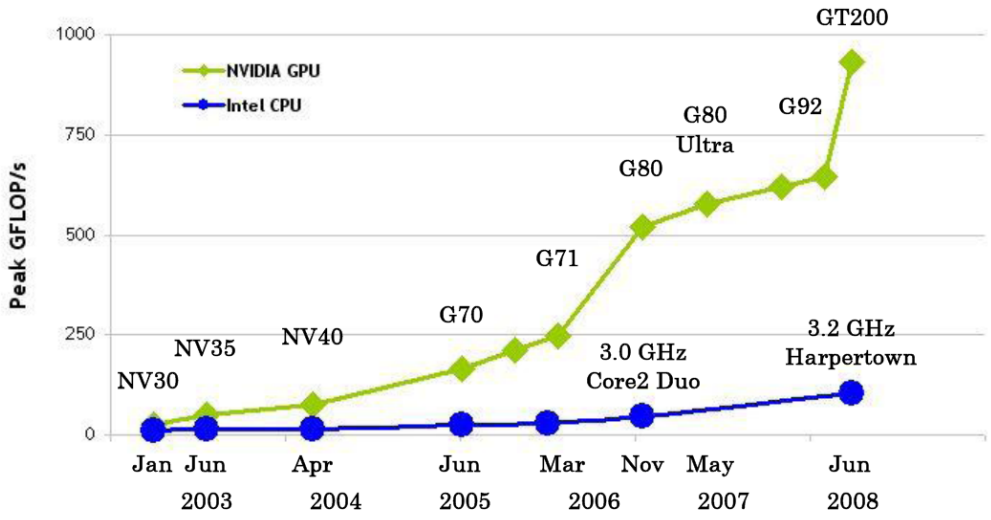


Figure 1.2: Floating-Point Performance for the CPU and GPU.

chitecture.

Chapter 7 contains the speedup and runtime results of our implementation, as well as a comparison with previous efforts and a discussion of the results.

Chapter 8 concludes the project and proposes further work.

Chapter 2

General Purpose Computations on GPUs

Graphical Processing Units, GPUs, were initially developed as coprocessors for offloading compute-intensive 2D and 3D operations involved in graphics. A highly parallel architecture allowed them to do these operations much faster than what was possible on CPUs. Early GPUs were fixed hardware designed to perform very specific operations, but advancements in graphical effects such as 3D shading demanded more programmable arhcitectures. Today almost every personal computer has a GPU, and a new market has opened up for handheld devices such as mobile phones. The two biggest companies in the industry are, ATI, who is owned by AMD, and NVIDIA.

The term General-Purpose computation on Graphics Processing Units or *GPGPU*, was coined in 2002 [5], by Mark Harris who now works for NVIDIA. The concept is to use the new programmable GPUs for computational tasks not necessarily linked to graphics. Initial attempts at running non-graphics related software on GPUs were largely proof-of-concepts, and often relied on clever use of the hardware and a substantial knowledge of the inner workings of the GPU. Interest in GPGPU has been on the rise the last few years, and in an attempt to appeal to a broader audience of scientific communities, NVIDIA launched its CUDA architecture in 2007. In 2008 an open framwork for multi-core programming called OpenCL was also released by the Khronos Group, and both ATI and NVIDIA are now supporting this platform.

2.1 Heterogeneous Programming

The consept of heterogeneous programming is to use different types of computational units in a system. A computer with a GPU is an example of a heterogeneous system, where the CPU acts as the host processor, and the GPU is a coprocessor.

The CPU is desgined as a general purpose processor capable and designed for handling advanced flow control and data caching. The GPU on the other hand is developed for handling code that is computational intensive, requires a high memory bandwidth, and can run in parallel. This is typical for the graphical operations performed in games, with high-resolution textures and 3D-models with a high polygon count.

In a typical program the CPU will handle the overall program execution and serial code, while the GPU performs tasks with a high degree of parallelization. The bottleneck in heterogenous computing such as GPU programming, is very often the transfer of data between the two devices. For the CPU and GPU, all communication must go through the PCI-Express interface. This means that as a programmer you want to transfer as much data as possible in as few transfers as you can, in order to reduce overhead costs. You also want to design your program such that it does as many computations as possible on the data each device has locally before transfering more data.

2.2 GPUs and Scientific Computations

The appeal of using GPUs in scientific computing lies in taking advantage of a hardware architecture that provides a high level of parallelization and computational power for a relatively small amount of money. This means that a scientist in practice can have what is effectively a supercomputer on his own desk. Using GPUs for scientific applications like numerical analysis does however often pose different requirements to the hardware than the graphics programs the GPUs traditionally have been developed for.

2.2.1 Double Precision

One of the most important issues is double precision. Double precision is important both for exact results as well as rapid convergence in numerical solvers.

On modern CPUs this basically comes for free, but on GPUs single precision floating point units outnumber those for double precision. This means that the speedup one usually get for programs requiring less precision, is much more limited when requiring double precision. The trend in hardware development is a shift in this ratio towards a more balanced split.

2.2.2 Accuracy and Reliability

Video games have always been the driving force for GPU technology. Physical simulations and visual effects in games are basically smoke and mirrors, and accuracy of the phenomenas simulated is only important up to the point that it looks good. In addition, if some calculations fail, resulting in say a miscolored pixel, it is typically overwritten within a fraction of a second.

Scientific applications on the other hand generally have much higher demands for accuracy and reliability, and miscalculations in a numerical simulation can have disastrous consequences for the final solution. NVIDIA has taken this seriously, and error correction technology in the form of **ECC** memory is part of the new generation of scientifically oriented GPUs from NVIDIA, such as the Tesla C2050 used in some of our tests.

2.2.3 Development Tools

Another very important aspect of programming on the GPU is the need for decent development and debugging tools. Both CUDA and OpenCL are frameworks developed for making development of more general applications easier, and the availability of good frameworks and reliable compilers for these is a necessity for the GPU's success in the scientific community. While OpenCL has gained some popularity over the past year, CUDA has the advantage of having been out for more years, and since it is developed by NVIDIA it naturally has the advantage of working well on NVIDIA GPUs.

An obvious selling point for OpenCL is that it is cross platform, and thus may eventually become the de facto standard. There are however several similarities between the two programming models, and OpenCL was made with CUDA in mind. The choice of platform for this thesis is CUDA, due to the fact that the currently available OpenCL compilers are known to have problems, and NVIDIA have GPUs out on the market that are specially made for GPGPU computing such as the Tesla GPUs used for our performance tests.

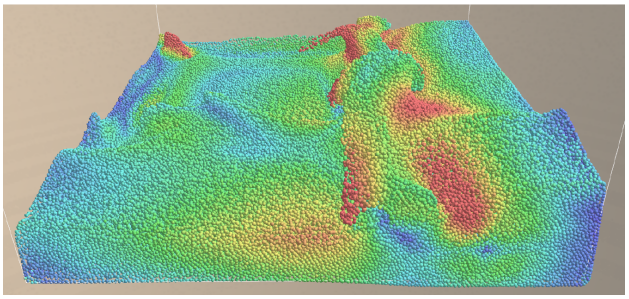


Figure 2.1: Smoothed-particle hydrodynamics(SPH) performed on the GPU. Picture taken from [11].

2.3 FEM on GPU and Similar Work

Research done on numerical simulations on the GPU range over a wide area of methods and applications, including sparse matrix solvers [2], fluid simulations using smoothed-particle hydrodynamics [11] and finite difference computations [13]. Some work using the finite element method focus on specific applications such as cloth simulations [18], and biomechanical simulations [21] achieving approximate speedups of 9 and 16 respectively.

Work has also been done on the assembly of the stiffness matrix resulting from the finite element discretization, and an article from Cecka, Lew and Darve at Stanford [3] presents various strategies for assembly using the GPU. Some of this work shares similarites to ours, since the matrix-free sparse matrix-vector multiplication in our conjugate gradient implementation in essence is an assembly procedure. We have not been able to find any work utilizing a block structured approach similar to ours, and believe that our strategy of combining this structure with a matrix-free implementation has not previously been explored.

2.4 The GPU Architecture

We will here present some of the main features of modern graphics cards, or GPUs, with a special emphasis on NVIDIA cards that incorporate the CUDA architecture.

Modern NVIDIA GPUs are seperated into several multiprocessors, each divided into processors that share a memory space, but with individual local memory.

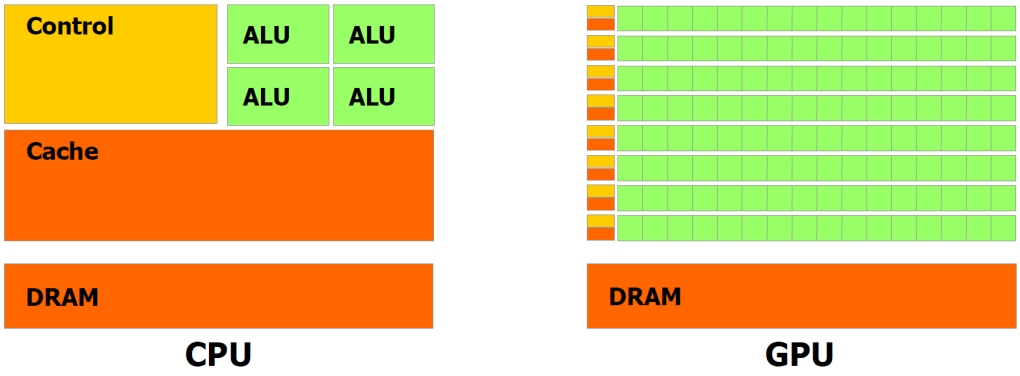


Figure 2.2: The difference in priorities of transistor usage on the CPU and GPU. Picture taken from [15].

For the NVIDIA Tesla C1060, the number of streaming multiprocessors is 30, with 8 cores in each. The new Tesla C2050 has 14 multiprocessors with 32 cores in each. In addition to the memory space shared between processors in a multiprocessor, all processors have access to an on-board global memory.

2.4.1 Processing power

Figure 2.2 shows the difference in priorities of transistor usage on the GPU compared to the CPU. The GPU has a very specialized architecture compared to the CPU, with many more transistors dedicated to floating point calculations, at the expense of features like flow control and branch predicition. The clock frequencies of GPUs are generally lower than for CPUs, and for our test system the CPU runs at 2.83 GHz, while the Tesla C1060 and Tesla C2050 GPUs run at 1.3GHz and 1.15GHz respectively.

2.4.2 Memory

The GPU has a lot of specialized memory units. Figure 2.3 shows how these memory units are located with respect to the processor. Memory transfers within the graphics card have a high bandwith. On the CPU, memory management is handled automatically, while on the GPU many of the choices on how the memory should be utilized is left to the programmer in order to achieve good performance. This often calls for a different approach to structuring of data, as will be seen in the implementation chapter.

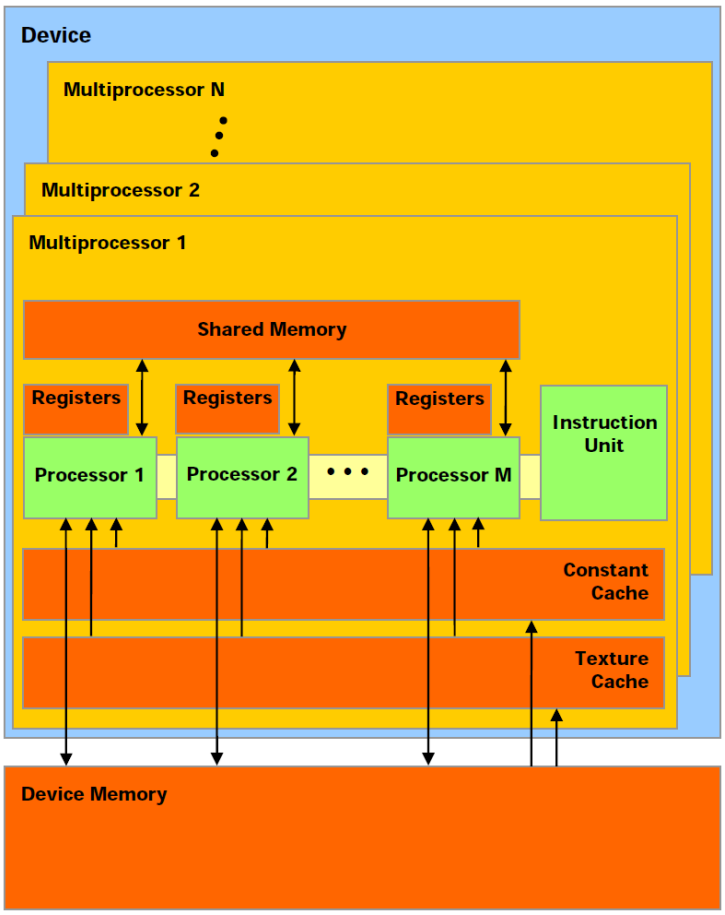


Figure 2.3: The different memory types on a NVIDIA GPU with CUDA support. Picture taken from [15].

2.4.3 Data transfers

The memory spaces on the CPU host system and the GPU device are completely separate, and in order to do work on the GPU, data sets must be transfered over the PCI-express bus. Transfers over the PCI-e bus is a bottleneck, with high latency and limited bandwith, and extensive use should hence be avoided. Even with tasks more suited for execution on the CPU, it is still sometimes better to make a GPU implementation with the same functionality, to avoid transfering data between the host and device systems.

In games, vertex and texture data for a scene is typically preloaded to the GPU when the scene is loaded. A similar approach should also be taken for more general numerical applications like our own. Coordinate data for all nodes is

only loaded once, and stays on the GPU through all stages of the solution process, including visualization.

2.5 CUDA

2.5.1 CUDA Programming Model

The first CUDA SDK was released in february 2007, and the current stable release is version 3.0. As a developer you can choose between two interfaces to write CUDA programs. The one that is the simplest to pick up for people accustomed to C programming, is C for CUDA, which offers a set of extensions to the C language. These include extensions for launching kernel functions in C on the GPU, and synchronizing program flow.

The other interface is the CUDA driver API, which provides a more low-level approach with functions to load modules of binary or assembly code. This gives the programmer more control and the program better portability, but is also harder to use. In our GPU implementations we have used C for CUDA.

Another difference between the two interfaces is that C for CUDA comes with a device emulation mode initiated by using the -deviceemu option in the compilation stage. The device emulation mode allows for better debugging by allowing break points and code that can not run on the device, like printf statements. Device emulation mode can also be used on systems without a CUDA enabled graphics card.

The compiler driver supplied with the CUDA SDK is called nvcc, and it can handle source files with a mix of both device code and host code. Device code is compiled with proprietary NVIDIA compilers, and host code by invoking the C/C++ compiler available on the host system. We have done our programming on Linux platforms, so the GNU compiler, gcc has been used.

2.5.2 Main Language and Syntax Structure

In CUDA, the *thread* is the smallest unit of parallelization. Threads are light weight processes. All threads used in executing a particular kernel, run the same version of the code. Branching can be initiated based on a thread ID. Threads are scheduled in *warps* of 32, and branching will not hurt performance

significantly as long as all threads in a warp follow the same branch. However, if threads in a warp diverge, the execution will be serialized and performance will drop.

Communication between threads must happen indirectly by writing and reading to the shared memory within the block. Unlike on CPUs, the GPU will in general not ensure that reading and writing is deterministic and atomic. Atomic operations that guarantees that an operation such as addition is performed without interuption from other threads, are supported, but only for integers.

In order to coordinate memory accesses, calls to a special synchronization function called *synchtreads()* must be made. This makes sure that all threads are at the same point in the execution of the kernel function. A typical example of use is when threads collaborate on reading a chunk of data from the global memory into the shared memory.

Threads are further structured into *blocks*. On NVIDIA devices of compute capability 1.3 such as the Tesla C1060, thread blocks can hold up to 512 threads. On the new 2.0 devices the limit is 1024. Cooperation between thread blocks is very limited, since it is a requirement that they can be executed in any order independent of each other.

This independency has the advantage that blocks can be scheduled on different cores, and will scale well to GPUs with a different number of cores. Synchronization between blocks can only be ensured at the completion of a kernel function. Programs dependent on interchanging of data between blocks must hence work by splitting the program into several kernel executions.

Threads can be structured in one, two or three dimensions, such that each thread has a thread index with x,y,z coordinates. This is simply an abstraction and is there for the convenience of the programmer.

A kernel program can run on several thread blocks, and these are structured in *grids*. Blocks in a grid are structured in one or two dimensions, again this is simply there for ease of implementation. The same can be done on a one-dimensional as on a two-dimensional structure.

All threads in a block must run on the same multiprocessor, but since blocks are independent, a grid can be split and scheduled over several cores, with far more blocks then there are processors. The number of blocks chosen is therefore rather a result of data size and what makes sense for the program at hand.

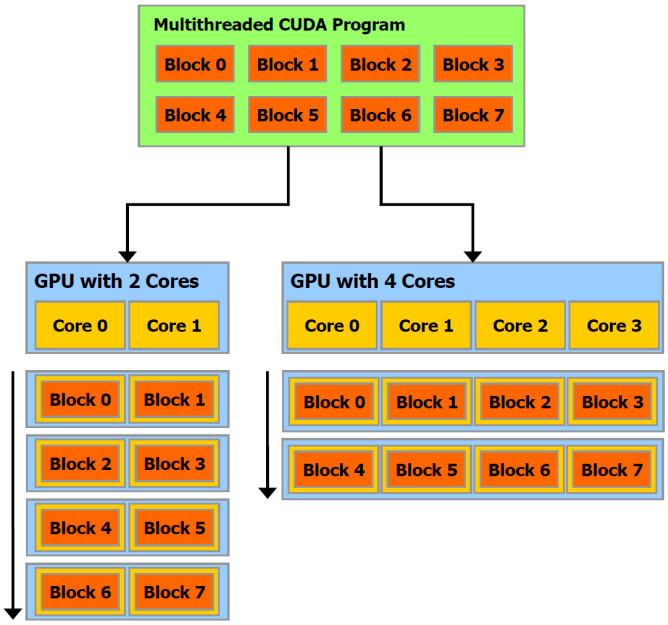


Figure 2.4: The partitioning of multithreaded programs into independent blocks means that programs will automatically scale to GPUs with a different amount of cores. Picture taken from [15].

Figure 2.4 illustrates this point.

The point of the thread, block and grid structures, is to provide a good model for GPU implementations, where a program is split into coarse independent part divided into blocks, but with finer parts that can cooperate between eachother within a block.

2.5.3 Memory Hierarchy

The memory structures that are part of the CUDA architecture are designed to work well with the parallelization structure mention above. Threads in a block will in most programs have to exchange data, and the shared memory allows for this.

A thread also has access to its own private local memory, as well as the big global memory or DRAM. The shared memory is on-chip and therefore much faster then the device memory. Typically a kernel will read in the data it needs

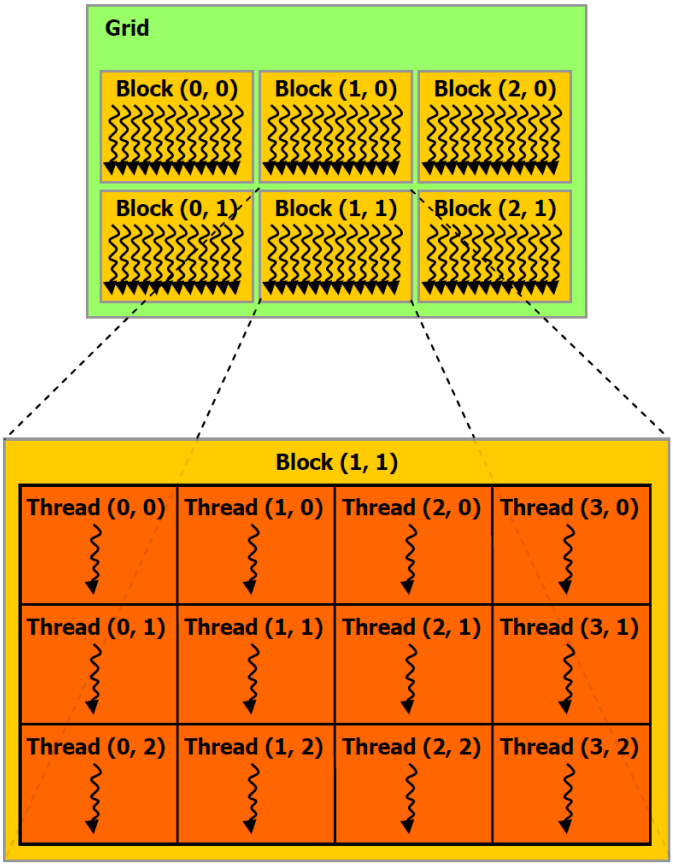


Figure 2.5: Grids are divided into blocks that again are divided into different threads. Picture taken from [15].

from the global memory, and use the shared memory as a working memory.

Shared memory will have the same lifetime as the the block it is attached to, while the global memory is persistent throughout execution of the program, and hence can be used for communication between blocks.

In addition to the private, shared and global memory spaces, threads have read access to the texture memory and the constant memory. The constant memory is cached and so it is extremely fast. It is however very limited in space, only 16kb, and is hence of limited use.

The host and the device will maintain different memory spaces, and pointers from one, can not be dereferenced in the other. Before execution of a kernel function, the necessary memory must be allocated on the GPU, and data must

Table 2.1: Glossary of terms used in GPU programming using CUDA.

Term	Description
Kernel	A function running in parallel on the GPU.
Thread	A single process.
Block/Thread block	A collection of threads.
Grid	A collection of thread blocks.
Streaming multiprocessor/SM	A collection of cores.
Host	The CPU system.
Device	The GPU system.
Shared memory	Memory shared by threads.
Global memory	Big memory space available for all.
Local memory	The individual memory for a single thread.
Constant memory	Small memory space available for all.

be transferred from the host system. This transfer goes over the PCI-Express interface.

The host has read and write access to the global and constant memory and must be used to allocate shared memory. The threads on the device can read and write to their own registers and local memory, shared memory within the block, and global memory. Threads only have read access to the constant memory.

Chapter 3

Mathematical Problem Description and The Finite Element Method

3.1 The Poisson Equation

In this thesis we will focus on the 2-dimensional Poisson equation. The purpose of our implementation is prototyping and proof-of-concept for block-structured FEM solvers on GPUs, and the Poisson equation is a natural choice given its familiarity and well understood properties. Among the uses for the Poisson equation are calculations of electrical potentials and of time independent heat distributions. The equation is given in (3.1). Here Ω is the domain, Γ_e is the *essential*, Dirichlet boundary conditions reflected in the solution space X, and Γ_n is the *natural*, Neumann boundary conditions. $\frac{\partial u}{\partial n} = \nabla \cdot n$, is the directional derivative in the outward normal direction.

$$\begin{gathered} -\Delta u = -\nabla^2 u = -\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u(x,y) = f(x,y) \text{ in } \Omega. \\ u = g \text{ on } \Gamma_e \\ \frac{\partial u}{\partial n} = h \text{ on } \Gamma_n \end{gathered} \tag{3.1}$$

3.1.1 Test cases

We have chosen two test problems for verification and performance tests in this thesis. They are presented below.

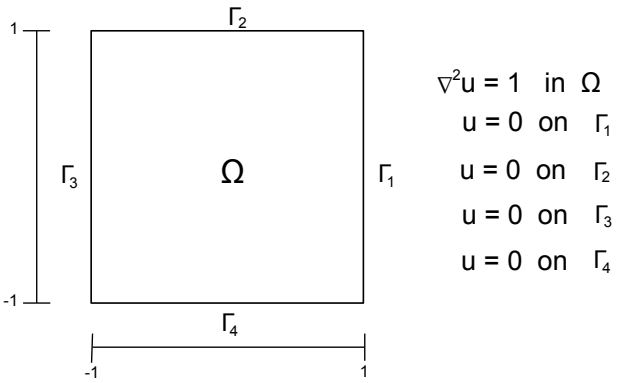


Figure 3.1: The constant source, zero boundary problem: Solution domain and boundary conditions.

The constant source, zero boundary problem

The test case we have called the *constant source, zero boundary problem*, is used for both verification and speedup tests. It has a constant source function $f = 1$ over the entire solution domain.

The previous implementations of finite element solvers developed in fall 2009 [22], appendix A, and january 2010, appendix B, both used this test case for performance measures, and we do the same in this thesis for the purpose of comparison. Figure 3.1 show the domain and boundary conditions. This is a very simple test case, and it is easy to visually verify due to the characteristic bell shape of its solution.

The analytic solution of the problem is given in equation (3.2).

$$u(x,y) = \frac{(1-x^2)}{2} - \frac{16}{\pi^3} \sum_{k=1,3,5,..}^{\infty} \left\{ \frac{\sin(k\pi(1+x)/2)}{k^3 \sinh(k\pi)} \times (\sinh(k\pi(1+y)/2) + \sinh(k\pi(1-y)/2)) \right\} \tag{3.2}$$

The non-constant load and boundary problem

The second test case is named the *non-constant load and boundary problem*, and has a load function $f = \sinh(\frac{\pi y}{2})\sin(\frac{\pi x}{2})/\sinh(\frac{\pi}{2})$, that varies over the domain. We also have a mix of boundary conditions, with homegenous dirichlet on two of the edges, $u = \sin(\frac{\pi x}{2})$ on a third, and finally a neumann condition on the fourth.

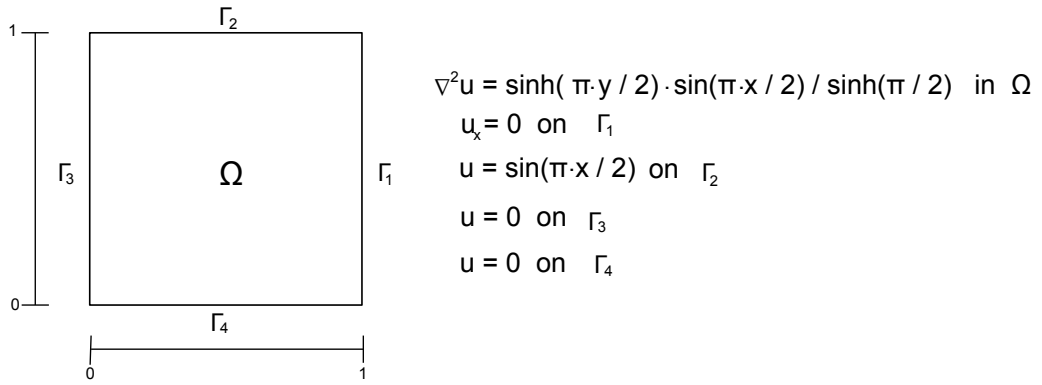


Figure 3.2: The non-constant load and boundary problem: Solution domain and boundary conditions.

We have chosen this test problem for verification of our handling of different boundary conditions and of the quadrature needed for evaluation. Figure 3.2 shows the domain and placement of the different boundaries, and the analytic solution is found in equation (3.3).

$$u(x,y) = \sinh\left(\frac{\pi y}{2}\right)\sin\left(\frac{\pi x}{2}\right) / \sinh\left(\frac{\pi}{2}\right) \tag{3.3}$$

3.2 The Finite Element Method

The Finite Element Method (FEM) was developed for solving complex problems in science and engineering. The initial development of the method is often accredited to Alexander Hrennikoff and Richard Courant in the early 1940s. The method offers a flexible way of solving equations over complex domains, and today several variations of the method using different types of elements and formulations are well established in the mathematical community.

The finite element method is based on the weak, variational formulation of boundary and initial value problems [16]. Multiplying (3.1) with a test-function $v \in X$ on both sides of the equation, and performing integration-by-parts using the boundary conditions, gives the weak formulation:

Find $u \in X^e$ such that

$$a(u,v) = l(v),\ \forall v \in X\ ; \tag{3.4}$$

where

$$X^e = \{v \in H^1(\Omega) \mid v|_{\Gamma_e} = g\}$$

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, d\Omega = \int_\Omega f v \, d\Omega + \int_{\Gamma_n} (\nabla u \cdot n) v \, ds = l(v)$$

$$H^1(\Omega) = \left\{ v | \int_\Omega v^2, \int_\Omega {v_x}^2, \int_\Omega {v_y}^2 < +\infty \right\}$$

$$X = H_0^1(\Omega) = \left\{ v \in H^1 | v|_{\Gamma_e} = 0 \right\}$$

The final term on the right-hand side, $\int_{\Gamma_n} (\nabla u \cdot n) v \, ds$ will simply vanish when there are no Neumann condition, or if they are zero.

3.2.1 Discretization

In order to solve our system numerically, the solution space must be discretized. We have chosen triangular elements in our treatment of the method. This gives the domain:

$$\overline{\Omega} = \bigcup_{T_h \in \tau_h} \overline{T}_h$$

where T_h is a triangle in the triangulation τ_h.

With piecewise linear basis functions, we define the finite-dimensional approximations to X and X^e to be respectively

$$X_h = \{v \in X | \; v|_{T_h} \in \mathbb{P}_1(T_h), \, \forall T_h \in \tau_h\}$$

and

$$X_h^e = \{v \in X^e | \; v|_{T_h} \in \mathbb{P}_1(T_h), \, \forall T_h \in \tau_h\}$$

That is all functions v in X_h and X_h^e that are linear polynomials over each element in our triangulation. We choose a nodal basis, that is a basis such that

$$v(\mathbf{x}_j) = \sum_{i=1}^{n} v_i \phi_i(\mathbf{x}_j) = \sum_{i=1}^{n} v_i \delta_{ij}.$$

where $\mathbf{x}_j$ are the nodes. The space X_h can then be written as

$$X_h = \text{span}\{\phi_1, ..., \phi_n\} :$$

$$\phi_i \in X_h, \, \phi_i(x_j) = \delta_{ij}, \, 1 \leq i, j \leq n$$

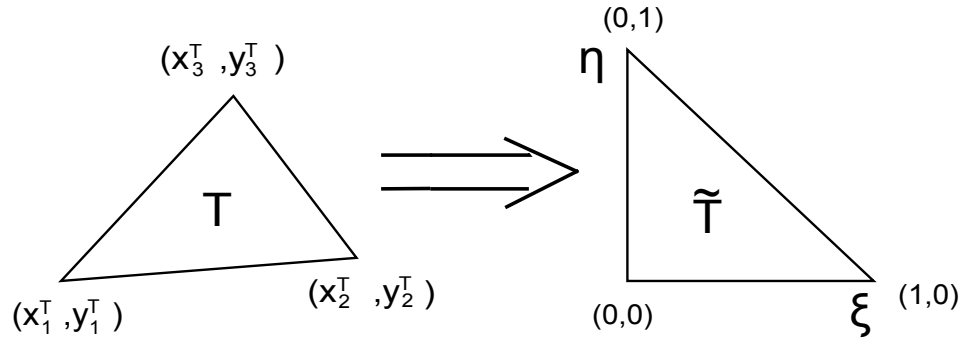


Figure 3.3: Mapping between physical and reference element

By expressing u and v in terms of the basis functions we can get the final discrete formulation. Find $u_h \in X_h^e$ such that

$$a(u_h, v) = l(v) \; \forall \, v \in X_h \tag{3.5}$$

This leads to the set of algebraic equations in (3.6)

$$\underline{A}_h \underline{u}_h = \underline{F}_h$$

$$A_{hij} = a(\phi_i, \phi_j) = \int_\Omega \frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y}\frac{\partial \phi_j}{\partial y}\, d\Omega \tag{3.6}$$

$$F_{hi} = l(\phi_i) \; , \; 1 \leq i \leq n.$$

$\underline{u}_h$ is the vector of nodal values of u_h.

3.2.2 Assembly

Let $\phi_i|_T = N_i$. We want to compute the integral over the element T_h^k

$$\int_{T_h^k} \frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x} + \frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial y}\, dA \tag{3.7}$$

In terms of programming, it is easier to calculate the integral (3.7) by use of substitution. We define a linear(and affine) mapping between T_h^k and a reference element $\widetilde{T_h^k}$ as shown in figure 3.3. Basis functions for $\widetilde{T_h^k}$ corresponding to $N_1(x,y)$, $N_2(x,y)$ and $N_3(x,y)$ for T_h^k are given by

$$\widehat{N}_1(\xi,\eta) = 1 - \xi - \eta$$
$$\widehat{N}_2(\xi,\eta) = \xi$$
$$\widehat{N}_3(\xi,\eta) = \eta$$

The relationship between the coordinates (x, y) og (ξ, η) is given by

$$x = x_1^T \widehat{N}_1 + x_2^T \widehat{N}_2 + x_3^T \widehat{N}_3$$
$$y = y_1^T \widehat{N}_1 + y_2^T \widehat{N}_2 + y_3^T \widehat{N}_3$$

We have

$$\frac{\partial N_i}{\partial x} = \frac{\partial \widehat{N}_i}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \widehat{N}_i}{\partial \eta}\frac{\partial \eta}{\partial x}; \quad \frac{\partial N_i}{\partial y} = \frac{\partial \widehat{N}_i}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial \widehat{N}_i}{\partial \eta}\frac{\partial \eta}{\partial y}$$

We need to find $\frac{\partial \xi}{\partial x}$, $\frac{\partial \xi}{\partial y}$, $\frac{\partial \eta}{\partial x}$ og $\frac{\partial \eta}{\partial y}$. Since $x = x(\xi, \eta)$ og $y = y(\xi, \eta)$, we get

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} d\xi \\ d\eta \end{bmatrix} \tag{3.8}$$

The determinant of the Jacobian matrix (3.8) is given by

$$|\underline{J}| = \begin{vmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{vmatrix} = \begin{vmatrix} x_2^T - x_1^T & x_3^T - x_1^T \\ y_2^T - y_1^T & y_3^T - y_1^T \end{vmatrix} \tag{3.9}$$

The inverse of (3.8) is given by

$$\begin{bmatrix} d\xi \\ d\eta \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} = \frac{1}{|\underline{J}|} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial x}{\partial \eta} \\ -\frac{\partial y}{\partial \xi} & \frac{\partial x}{\partial \xi} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix}$$

This gives us

$$\frac{\partial \xi}{\partial x} = \frac{1}{|\underline{J}|}\frac{\partial y}{\partial \eta} \quad \frac{\partial \xi}{\partial y} = -\frac{1}{|\underline{J}|}\frac{\partial x}{\partial \eta}$$
$$\frac{\partial \eta}{\partial x} = -\frac{1}{|\underline{J}|}\frac{\partial y}{\partial \xi} \quad \frac{\partial \eta}{\partial y} = \frac{1}{|\underline{J}|}\frac{\partial x}{\partial \xi}$$

It follows that

$$\begin{aligned} \frac{\partial N_1}{\partial x} &= \frac{1}{|\underline{J}|}(y_2^T - y_3^T) & \frac{\partial N_1}{\partial y} &= \frac{1}{|\underline{J}|}(x_3^T - x_2^T) \\ \frac{\partial N_2}{\partial x} &= \frac{1}{|\underline{J}|}(y_3^T - y_1^T) & \frac{\partial N_2}{\partial y} &= -\frac{1}{|\underline{J}|}(x_3^T - x_1^T) \\ \frac{\partial N_3}{\partial x} &= -\frac{1}{|\underline{J}|}(y_2^T - y_1^T) & \frac{\partial N_3}{\partial y} &= \frac{1}{|\underline{J}|}(x_2^T - x_1^T) \end{aligned} \tag{3.10}$$

All the terms in (3.10) are constant, and the area of T_h^k is denoted by A_T. The element matrix for an element k is then given by

$$\underline{A}^k = A_T \begin{bmatrix} \partial_x N_1^2 + \partial_y N_1^2 & \partial_x N_1 \partial_x N_2 + \partial_y N_1 \partial_y N_2 & \partial_x N_1 \partial_x N_3 + \partial_y N_1 \partial_y N_3 \\ \partial_x N_2 \partial_x N_1 + \partial_y N_2 \partial_y N_1 & \partial_x N_2^2 + \partial_y N_2^2 & \partial_x N_2 \partial_x N_3 + \partial_y N_2 \partial_y N_3 \\ \partial_x N_3 \partial_x N_1 + \partial_y N_3 \partial_y N_1 & \partial_x N_3 \partial_x N_2 + \partial_y N_3 \partial_y N_2 & \partial_x N_3^2 + \partial_y N_3^2 \end{bmatrix}$$

The different local contributions in the element matrices $\underline{A}^k$, can be added into the global stiffness matrix $\underline{A}$. The final linear system can be solved using a regular direct or iterative solution process.

In the next chapter, chapter 4 we will define am iterative conjugate gradient approach for solving the system in (3.6), using only the element matrices $\underline{A}^k$ locally, never actually forming $\underline{A}$.

A similar treatment, using the reference element, is done for the right-hand side of the equation (3.4), $l(v)$. For the *constant source, zero boundary problem*, the right-hand side is simply reduced to $\int_\Omega v \, d\Omega$, and evaluation is very simple. Since we allow for inhomogenous load and boundary function in our implementation, the evaluation of the integral for more general source functions and boundary conditions must be handled through use of quadrature.

3.3 Quadrature

An analytic solution is usually not available when calculating integrals over elements for the stiffness matrix, the load vector and boundary conditions given in the previous section. We then have to use a numerical approximation method.

We have chosen to evaluate integrals over the reference element given in figure 3.3, using Gauss-Legendre quadrature. Details on this can be found in [12]. The approximation to the integral of a function f over the reference element in two dimensions can be expressed through a double summation over special weighted sampling points. M_1 and M_2 in the formula below is the number of integration points along each axis.

$$\int_0^1 f(\xi, \eta) \, d\xi \, d\eta \approx$$

$$\sum_{i=1}^{M_1} \sum_{j=1}^{M_2} W_i W_j f(\xi_i, \eta_j)$$

Table 3.1: Weights and sampling points for 3-point Gauss-Legendre quadrature over a triangular domain.

ξ	η	Weight
0.1666667	0.1666667	0.3333333
0.6666667	0.1666667	0.3333333
0.1666667	0.6666667	0.3333333

The weights W and integration points (ξ, η) for 3-point Gauss-Legendre quadrature over the triangular reference element is given in table 3.1.

3.4 Error Analysis

If we define the discretization error as $e = u_h - u$, we can find a general error result for the finite element formulation given earlier.

$$1.\ a(v,e) = 0\ ,\ \forall v \in X_h$$

$$2.\ a(e,e) \leq a(U_h - u, U_h - u)\ ,\ \forall U_h \in X_h^e$$

The first part tells us that u_h is the projection of u onto X_h with respect to $a(\cdot,\cdot)$, and the error e is orthogonal to every v in X_h. The second equation is the *best approximation property*, and it tells us that there is no better approximation u_h to u in X_h^e. The error is minimized in $a(\cdot,\cdot)$. A proof for these properties can be found in [8].

The bilinear symmetric form $a(\cdot,\cdot)$ is also called the *energy norm*, and we define it as

$$||e||_\varepsilon = a(e,e)^{\frac{1}{2}} = \left(\int_\Omega \nabla e \cdot \nabla e \, d\Omega \right)^{\frac{1}{2}} \tag{3.11}$$

In the the results chapter, chapter 7, we shall use this norm for verification of the implementation. We will then use the exact solution, u, for the *constant source, zero boundary* problem given in equation (3.2).

Chapter 4

The Conjugate Gradient Method

4.1 The Conjugate Gradient Method

4.1.1 Theory and Background

The discrete system of equations from the finite element method must be solved by a numerical solver. The Conjugate Gradient method was originally proposed by Magnus R. Hestenes and Eduard Stiefel in 1952 [7] as a method for solving systems of linear equations. The method requires a symmetric positive-definite(SPD) system of equations, that is a system of the form $\underline{\underline{A}}\underline{x} = \underline{b}$ where $\underline{\underline{A}}$ is such that $\underline{x}'\underline{\underline{A}}\underline{x} > 0 \ \forall \underline{x} \in \Omega$. The stiffness matrix from the finite element method applied to the Poisson equation satisfies this requirement.

The method draws it name from the fact that the succesive search directions $\underline{p}_k$, are *conjugate* with respect to the SPD A. That is

$$\underline{p}_i{}^T \underline{\underline{A}}\underline{p}_j = 0 \ , \forall i \neq j.$$

A more complete presentation of the mathematical foundation for the conjugate gradient method can be found in [19] and [24].

4.2 Algorithm

The standard algorithm as given in [19] is stated in algorithm 1. $\underline{r_0}$ and $\underline{p_0}$ are respectively the initial residual vector and the initial search direction. When the initial guess $\underline{x_0}$ is $\underline{0}$, $\underline{r_0}$ and $\underline{p_0}$ simply becomes $\underline{b}$ which for the finite element method is the right hand side of (3.6), $\underline{F}_h$. α_k is the steplength, and β_k is used to determine the next search direction.

Algorithm 1 The Conjugate Gradient Method

$\underline{r}_0 = \underline{b} - \underline{A}\underline{x}_0$, $\underline{p}_0 = \underline{r}_0$
For $k = 0, 1, 2, ..$until convergence

$$\alpha_k = \frac{\underline{r}_k{}^T \underline{r}_k}{\underline{p}_k{}^T \underline{A}\underline{p}_k}$$
$$\underline{x}_{k+1} = \underline{x}_k + \alpha_k \underline{p}_k$$
$$\underline{r}_{k+1} = \underline{r}_k - \alpha_k \underline{A}\underline{p}_k$$
$$\beta_k = \frac{\underline{r}_{k+1}{}^T \underline{r}_{k+1}}{\underline{r}_{k+1}^T \underline{r}_k}$$
$$\underline{p}_{k+1} = \underline{r}_{k+1} + \beta_k \underline{p}_k$$

End

4.3 Convergence and Error

Theoretically the Conjugate Gradient method will converge and reach the exact solution in at most N steps for N degrees of freedom. Each step k projects the exact solution into the k-dimensional solution space spanned by the A-conjugate basis vectors.

In practice, each step will not be solved exactly due to round-off error in the computations. The method will however in general converge to an acceptable error-tolerance in far less than N iterations. This rapid convergence is one of the greatest strengths of the method.

It can be shown [19] that the convergence in the A-norm is given as in equation (4.1), where $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ is the condition number of $\underline{A}$, x_* is the exact solution, and x_k is the approximate solution after k steps.

$$\|x_* - x_k\|_A \leq 2 \left[\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right]^k \|x_* - x_0\|_A \tag{4.1}$$

4.4 Matrix-Free Versions

Our implementation of the finite element method never forms the actual stiffnes matrix $\underline{A}$, and hence the matrix-vector product $\underline{A}\underline{p}_k$ in the algorithm is performed implicitly by doing the calculations per-element as can be seen in algorithm 2. θ is the local-to-global mapping of indices. The terms $A_{\alpha\beta}$ are the entries in the element matrix $\underline{A}^k$ as given in section 3.2.2. This direct "on-the-fly" computation of the element matrices is proposed in [14].

Algorithm 2 Direct evaluation of $\underline{y} = \underline{A}\underline{p}_k$

$\underline{y} = 0$
For $k = 1,2,..K$(elements)
    For $\alpha = 1,2,3$
        $i = \theta(k,\alpha)$
        For $\beta = 1,2,3$
            $j = \theta(k,\beta)$
            $y_i = y_i + A^k_{\alpha\beta} p_j$
        End
    End
End

In the GPU-implementation, the outer for-loop in the algorithm is replaced with a parallel distribution where calculations for each element is done on seperate threads.

Chapter 5

Structured Block Representation

Our first effort at constructing an implementation of a finite element equation solver, fall 2009 [22], appendix A, was done on completely unstructured triangular meshes. These were then divided into partitions favouring load balancing. This approach relied heavily on preprocessing of the mesh data in order to fit it into data structures suitable for the GPU programming model. Numerous support structures were needed to handle boundaries between partitions and on the closure of the domain. The overall complexity made the application rely to much on a particular implementation of the pre-processing, hence making it implausible for a general programming approach to finite element implementations on GPUs.

As a precursor to the final implementation presented in this thesis, a simpler prototype utilizing a topological and geometrically structured grid was developed in january 2010, and showed considerable promise in terms of speedup and general ease of implementation [17], appendix B. Parts of the implementation quite heavily exploited the rigid and well-defined relationship of the various blocks. In particular the problem of reduction of values on the borders between blocks, was done in a manner not really suited for a general block-structured approach.

In order to solve problems on more complex geometries than the ones utilized in [17], the topological block structure is kept but we allow for blocks arranged in an arbitrary manner, with chartesian geometry varying for each individual block.

Researchers at SINTEF Applied Mathematics have developed a system for han-

dling isogeometric representations in CAD and FEA, called ICADA [20]. We have decided to utilize a cutomized version of their specification of spline patch topology for our 2D finite element domain implementation.

5.1 Technical Specifications

The fundamental idea behind the block structured topology is to reduce the number of variables and indices needed to describe the domain, and instead introduce only a few that all others can be deduced from. For instance, knowing the index of the first node on an edge and a corresponding increment value, allows us to find the indices of all other nodes along that edge. The representation developed in the ICADA project is for 3-dimensional geometries, and all the nodes of a block can be found from the values listed in table 5.1. An example of the numbering convention can be seen in figure 5.1. The full specifications can be found in appendix C.

Certain assumptions are made in the documentation. They only allow for *completely matching* blocks. This means that two blocks are connected topologically only if all nodes on the surface between them are shared and identical for the blocks. In other words completely matching blocks share an entire common surface, otherwise there is a *crack* between them. It is also required that the grid generation process produces unique global indices for all nodes or *control points* in the model, and that the internal node number generation can be done locally within each block independently and in arbitrary order, for instance in a parallel context. This last point is of particular importance for our project, given the nature of our hardware. As explained in chapter 2, CUDA blocks are required to execute independently.

The representation described in [20], appendix C, only provides information of the topology. The necessary coordinate data needed for the solution or simulation step must be provided seperately.

Table 5.1: Variables used in the specification of spline patch topologies.

Variable	Description
IBLOCK	Spline patch index
IBNODi	Global node number of vertex i
ICNODi	Global node number of second point along edge i
INCRi	Increments in global numbering alond the edge (± 1)
ISNODi	Global node number for first interior point on face i
INCIi	Increments in global numbering in local I-direction on the face (± 1)
INCJi	Increments in global numbering in local J-direction on the face (± 1)
IINOD1	Global node number of the first interior point of the patch

Table 5.2: Variables used in the description of a 2-dimensional block structure.

Variable	Description
IBLOCK	Block ID
IBNOD1,...,4	Global node numbers for vertices
ICNOD1,...,4	Global node numbers for second points along edges
INCR1,...,4	Increments along positive direction of the edges
ISNOD	Global node number for first interior point in block
INCI	Increment in local I-direction of interior nodes
INCJ	Increment in local J-direction of interior nodes
NINOD	Number of total nodes in horisontal direction
NJNOD	Number of total nodes in vertical direction
NEL	Number of elements

5.2 Adaptation to the project

The 2-dimensional domains we want to look at in our implementation and tests use a slightly modified version of the block-definitions suggested in the ICADA spline patch model. Table 5.2 shows all variables included for each block.

Some additional assumptions are made of the grid in order to make the GPU implementation more efficient. The total number of elements in each block is assumed to always be 512. This coincides with the maximum number of threads allowed in a CUDA thread block on architectures of compute capability 1.x, and a more detailed reasoning for this choice can be found in chapter 6. Indices for all nodes in the global domain are assumed to follow the ordering given in equation 5.1. Again the reasoning is due to implementation details fleshed out in chapter 6.

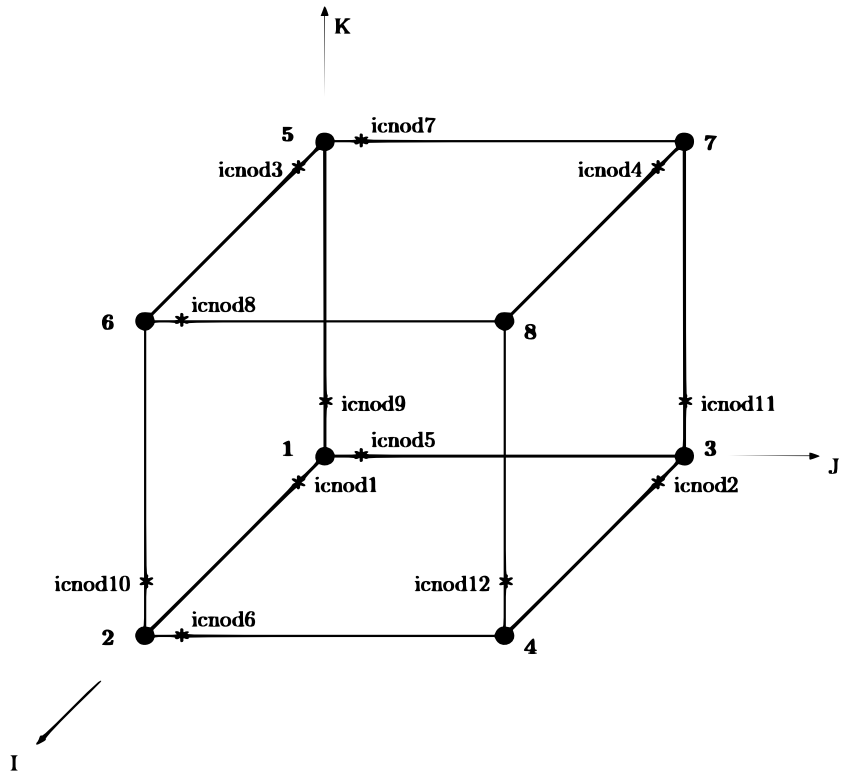


Figure 5.1: Numbering convention for the spline surfaces in ICADA.

$$0 \leq i < j < k < N_{tot}. \tag{5.1}$$

such that $\overline{x}_i \in I = \{\text{Corner Nodes}\}, \overline{x}_j \in E = \{\text{Edge Nodes}\}, \overline{x}_k \in I = \{\text{Internal Nodes}\}$

In accordance with equation 5.1, we start by assigning a unique index to the vertices of all blocks in the domain, starting on 0 with increments of 1. Then all edges are numbered, counting first along the positive I-direction, then the J-direction, as indicated in figure 5.2. Finally all interior points are numbered such that all interior point of a block are succesive.

The numbering scheme outlined above is useful for describing the global properties of the blocks and the domain, and hence for coordinating contributions on nodes shared by several blocks. But the solution over each block is solved locally and independently and allows for a local numbering more convenient for an efficient and simple implementation. A natural ordering of nodes is such that we number from left to right along the positive I-direction indicated in figure 5.2, and then jump to the next horizontal line, such that index = $i + j *$

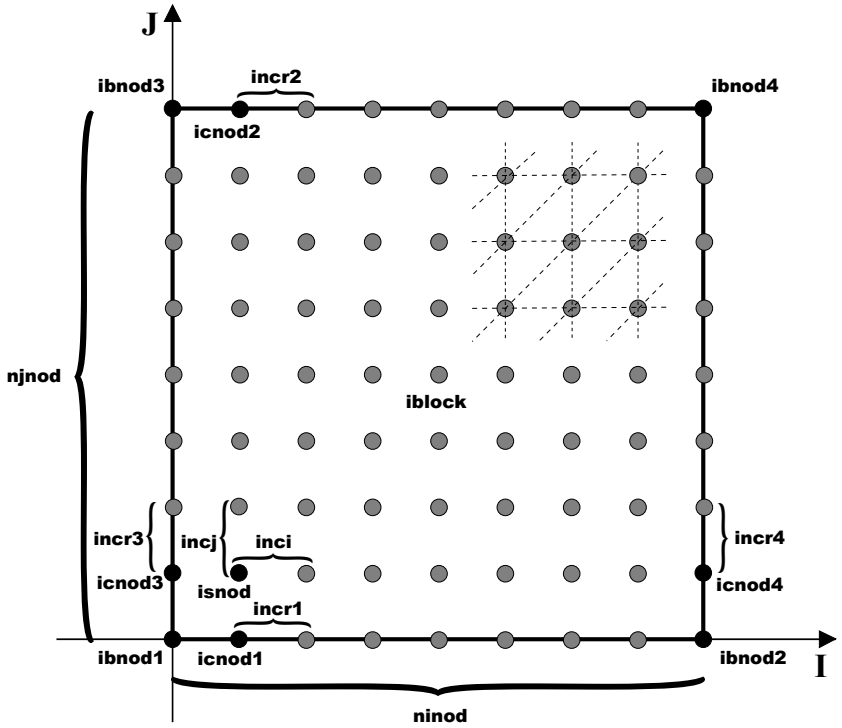


Figure 5.2: Numbering convention for the 2-dimensional block structure.

NINOD, where NINOD is the width of the block in number of nodes.

We use triangular elements in our implementation, and the numbering of elements follows the same natural ordering as the local nodes. The combination of the strict ordering of nodes and elements allow us to derive all local indices for an element, based on its index in the CUDA thread block as can be seen in algorithm 4 in chapter 6. The variables in 5.2 provides a mapping from the local nodes following a natural ordering, to the global indices used for reduction on block boundaries, and to represent the vectors in the conjugate gradient iteration. See figure 6.4 in chapter 6 for an illustration.

Chapter 6

Implementation

In this chapter we outline the implementation of the finite element solver, and present some of the challenges involved. All programming is done in C++ with the CUDA SDK 3.0 for the GPU specific parts. For the visualization we have used OpenGL. Both the GPU and the CPU versions are done in single precision.

6.1 Program Flow

The entire solution process consists of several parts. Figure 6.1 shows the workflow of the program. First, the grid data and coordinate data must be generated. This data is then fed into the solution driver that sets up the data in the appropriate data structures for solving the CG-iteration. In the solver, the load vector F is calculated on the host system and the data is then transferred to the GPU. In the CG-iteration process, the matrix-free "matrix-vector product", is performed on the GPU. Then the remaining linear algebra tasks of the CG step is performed using the CUBLAS library provided by NVIDIA. The succeding sections of this chapter follow this order. Some additional CUDA kernels used for enforcing Dirichlet boundary conditions and masking steps on the vectors are not included in the descriptions presented here, as they are insignificant to performance and do not contain any core concepts not presented elsewhere.

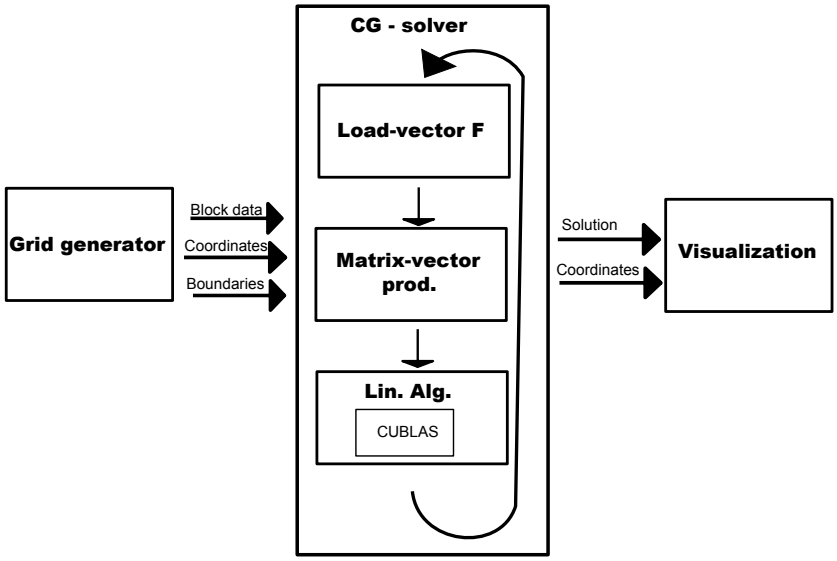


Figure 6.1: Program flow.

6.2 Grid Generation

In order to test the implementation of the finite element solver, we needed test cases to do this on. A simple application was made for generating grids with boundary conditions.

The program generates square grids, with blocks containing 512 elements each, and the user can specify the refinement and corner coordinates for the grid.

For specification of boundary conditions, a subroutine in the grid generation process draws the blocks, and let the user select edges to apply a specific boundary condition code. This code can be arbitrary, but must be linked to an interpretation of its meaning in the solver. The convention we used, reserved the numbers 1-99 for dirichlet conditions, and 100-> for neumann. A screenshot of this selection process can be seen in figure 6.2. Once the user is done assigning boundary codes, the codes are written to a binary file read by the solver. Function definitions for non-constant boundaries are specified in a header-file that goes with the solver.

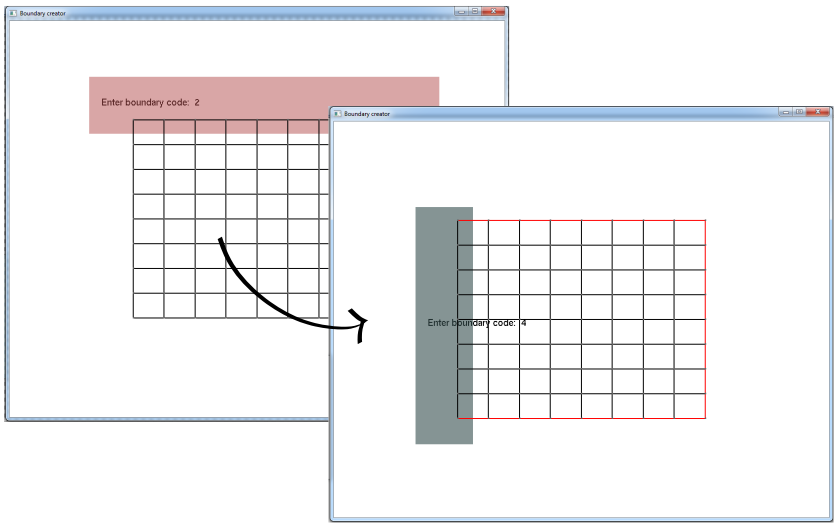


Figure 6.2: Screenshots from the application for selecting boundary conditions for the edges

6.3 Load Function

As can be seen from equation (3.4), the right-hand side of the equation handles both the loads over the domain, as well as the boundary conditions. This makes the calculation of the F-vector the most laborous and complex part of the program. Fortunately, this only needs to be done one time in the whole solution process, as opposed to the matrix-vector multiplication that must be calculated in every iteration of the CG-method.

The CUDA programming manual suggests that operations best suited for a serial implementation, should be considered to run on the host system. The downside of such an approach is that data must be transfered between the host and GPU memory. Again, the load function is only evaluated once, so the memory transfer is also only performed once. Because of these considerations we have decided to implement the function to run on the CPU.

A pseudo-algorithm of the work done by the load-vector function can be found in algorithm 3. Of particular interest is the calculation of each elements indices. We use the natural ordering described in chapter 5, and the algorithm for calculating the indices for the 3 nodes of an element is shown in algorithm 4. The advantage of this method is that it exploits the natural ordering of our structured block, and hence only need one variable, the element index, and one

Algorithm 3 Pseudo algorithm for the load vector F.

```
for block = 0 to grid size - 1 do
  for element = 0 to block size - 1 do
    [calculate the elements indices] [Algorithm 4]
    [read coordinate data for nodes in the element]
    [calculate load contributions]
    [calculate boundary contributions]
  end for
  for node = 0 to Nodes in block - 1 do
    [reduce contributions from elements] [Figure 6.3]
    [local index -> global index]
    [write to global vector]
  end for
end for
```

constant, the width of the block, for calculating all indices.

The operators "%", and "/" used in algorithm 4 are the modulo and integer division operators. It is important to note that integer operations are very slow on the GPU, and division and modulo operations should be replaced with bit-operators where possible. Integer division i/n, where n is a power of 2 can be replaced with the bitwise right shift assignment $i >> \log2(n)$, and the modulo $i\%n$ with a bitwise AND assignment $i\&(n-1)$. Since the integer operations in our algorithm are done on powers of 2, and are literal constants, the compiler will replace them with the appropriate bit operators.

The calculations from local to global indices in the final stages of algorithm 3, are done using the block-variables in table 5.2, chapter 5, and the storage order shown in figure 6.4.

6.4 Matrix-Vector Product

The matrix-free matrix-vector multiplication between the stiffness matrix $\underline{A}$ and the search direction $\underline{p}$ as described in the final section of chapter 4 is performed on the GPU. A pseudo-algorithm of the process is found in algorithm 5.

For the load function running sequentially on the CPU, there is no problem of race conditions between the various blocks. On the GPU, when threads are try-

Algorithm 4 Calculation of local indices for nodes in an element.

```
if(elementId%2==0){
        indices[0] = elementId/2 + elementId/BW;
        indices[1] = (elementId +BW)/2 + (elementId+BW)/BW +1;
        indices[2] = (elementId +BW)/2 + (elementId+BW)/BW;
}else{
        indices[0] = elementId/2 + elementId/BW;
        indices[1] = elementId/2 + elementId/BW +1;
        indices[2] = (elementId +BW)/2 + (elementId+BW)/BW +1;
}
```

Algorithm 5 Pseudo algorithm for the matrix-free matrix-vector multiplication.

[*read data from global to shared memory*]
[*calculate the elements indices*][Algorithm 4]
[*read coordinate data for nodes in the element from shared memory*]
[*calculate contributions*]
[*reduce contributions from elements*][Figure 6.3]
[*local index -> global index*]
[*write to global vector*]

ing to write to the same space in memory at the same time, no guarantees are given about the correctness of the result. From figure 6.3 we see that each node can get contributions from six different elements, which calls for a way of handling the race conditions that would arise between threads both internally in a block, but also between blocks.

The implementation made in january [17] used a method where each node was represented with 6 slots in the global vector arrays. This wasted a lot of space, and eventually limited the test problems we could fit in the GPU memory. In this final implementation, we have instead taken advantage of the local natural ordering within blocks, and the indices for corners and edges provided by the block-structured variable list.

Locally in a block, we let each element k(running on one thread each) store in shared memory the calculated local element matrix $\underline{A_k}$ multiplied with the corresponding values from $\underline{p}$. We then synchronize threads in the block with *__syncthreads()* and let each node gather its contributions and store the final value in the result array $\underline{y}$.

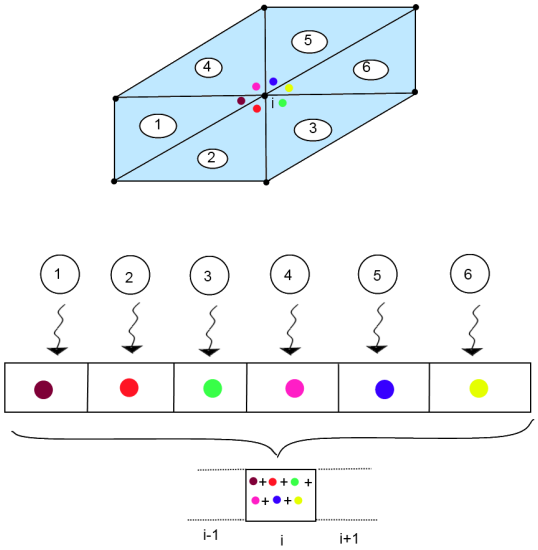


Figure 6.3: The contributions from all 6 elements to a node are computed by independent threads, and must be collected and summed up at the end.

Nodes that are on the boundaries of the block can get contributions from 2 different blocks for edge nodes, and up to 4 blocks for corner nodes. This is the reason for the numbering scheme in figure 6.4, with the total amount of array entries being $4 \times n_C + 2 \times n_E + n_I$ for the global vectors used in the conjugate gradient iteration, where n_C is the total number of unique corner nodes, n_E the number of unique edge nodes, and n_I the total number of unique internal nodes.

The global indices for the edge and corner nodes can be found from the IBNOD, ICNOD and INCR arrays in table 5.2 in chapter 5. In addition, local corner and edge nodes need to know which of the 4 or 2 places, respectively, they should store their contributions in. Figure 6.5 shows how this is done for the edge nodes.

A node being on the *left* or *bottom* edge of the block will store its number on the *first* slot, and contributions on the *right* and *top* edges are stored in the *second* slot. A similar scheme is used for corner nodes as seen in figure 6.6. In other words, the storage corresponds to the index number in the IBNOD and ICNOD tables, as can be seen from figure 5.2 in chapter 5. Inner nodes are unique for

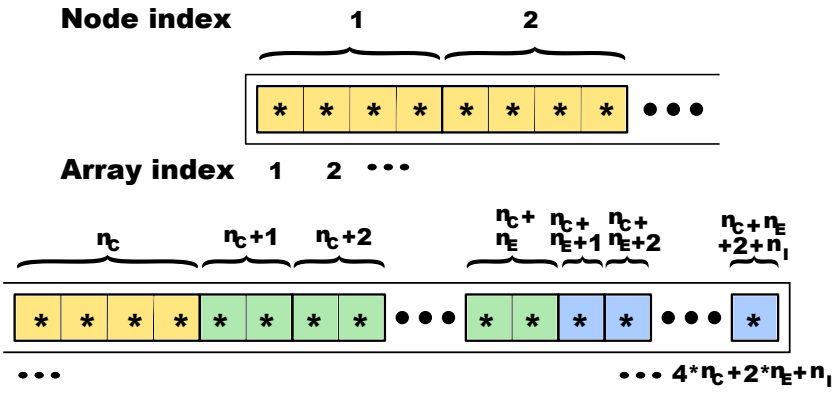


Figure 6.4: Storage order in arrays for corner, edge and inner nodes. n_C is the total number of corner nodes, n_E is the total number of edge nodes and n_I is the total number of inner nodes.

each block and can be stored directly in the array entry corresponding to the global index.

6.5 Linear Algebra

The calculation of the matrix-free matrix-vector product demands a custom made kernel to gain the benefits of parallelization. The innerproduct of vectors needed wheen finding the step length α, and β to determine the next search direction, as well as the SAXPY operations needed to update the solution $\underline{u}$, residual $\underline{r}$ and search direction $\underline{p}$ can be done using the BLAS implementation for CUDA, called *CUBLAS*. The benefit of using the CUBLAS library is that it is already heavily optimized, and implementing our own versions of the functions would be a lot of work unlikely to provide any performance improvements.

One slight disadvantage of using the CUBLAS library is that the result of the dot-product must be sent to the host system. This introduce a small latancy, but the performance of the function is so good that it by far outweighs this small addition in runtime. The initialization of the library also introduce an added time of about 0.33 seconds on our system, and for smaller test cases, this will dominate the runtime. The initialization function *cublasInit()* serves to allocate hardware resources and attach CUBLAS to the GPU used by the host thread. However this initialization function only needs to be run once for the entire application and is also negligable for more realisticly sized test problems. The first version of the CUBLAS library only allowed for single precission calculations, but the new

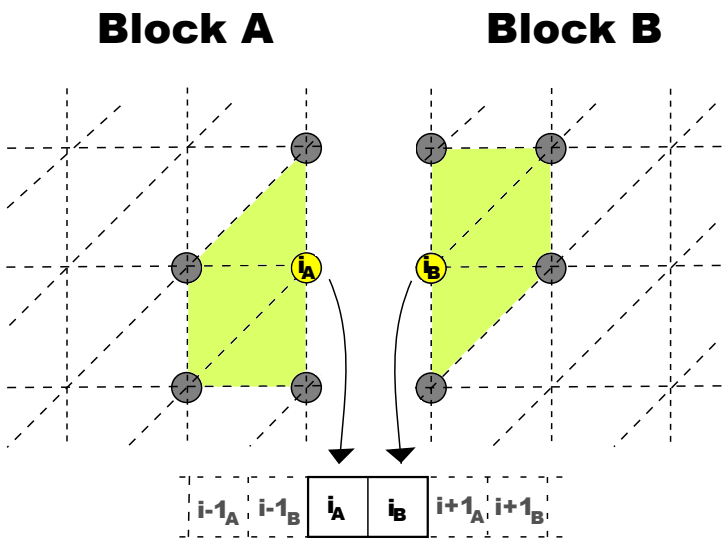


*Figure 6.5: Definition of indices from various block for reduction to edge-nodes. The index **i** is for a specific edge-node, and contributions are stored in the global solution vector based on the left-right orientation of the blocks **A** and **B** with respect to **i**.*

version provides double precision for GPUs that can support it.

The way our data is stored in the various vectors means that running the innerproduct on the entire vectors would result in the wrong value, since corner and edge nodes shared by blocks will be repeated in the data set. As can be seen from figure 6.4, the last part of the vector contains the values at the internal nodes of each block. These are all unique, and hence the regular dot-product function *cublasSdot*() can be run on the vector, utilizing pointer arithmetic such that the pointer passed to the function is the pointer to the vector plus an offset equal to total number of corner and edge nodes.

For the corner and edge parts of the vector, the dot-product is still performed with CUBLAS, but with an offset between each node. This slows down the function a bit, and it runs at a lower occupancy, but the time spent in this function is marginal, so no attempts were made to implement a more optimized version.

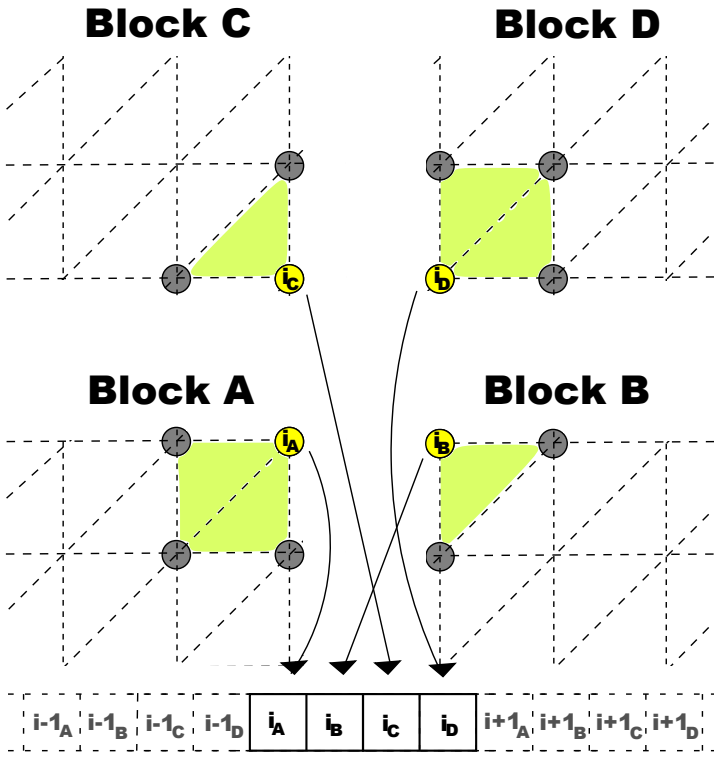


*Figure 6.6: Definition of indices from various block for reduction to edge-nodes. The index **i** is for a specific corner-node, and contributions are stored in the global solution vector based on the orientation of the blocks **A**, **B**, **C** and **D** with respect to **i**.*

6.6 Visualization

In order to get the necessary figures for presenting the results of the solver, we decided to build a custom visualization tool in OpenGL. This was also very useful for tracking implementation errors during development. An alternative to creating our own visualizer could have been to export the solution data to MATLAB, and use the built-in visualization capabilites there. The benefit of our tool is that we can get the results immediatly instead of having to go through another application. The flexibility provided with having full control over coloring and rendering parameters was also useful.

An important aspect of using OpenGL is the interoperability with CUDA. This means that solution data do not have to be transfered back from the GPUs device memory back to the host memory. In our implementation the solver is only

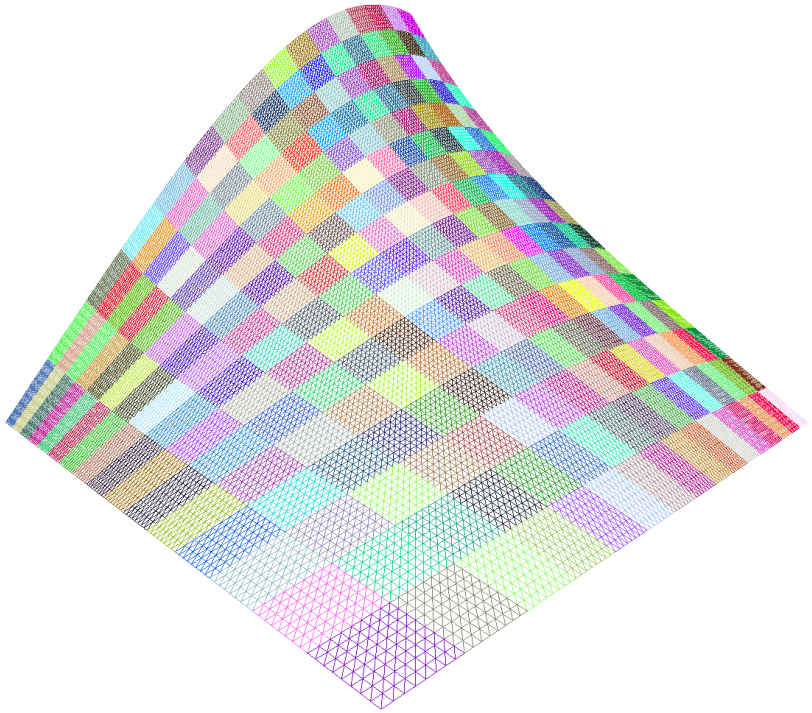


Figure 6.7: Screenshot of a Poisson solution in the custom built visualization tool, demonstrating the wireframe mode

run once, such that data would only have to be transferred once anyway. But in an application running in realtime with changing data, our choice of visualization would probably prove valuable. Finally, the visualization done in OpenGL can handle much larger meshes without slowing down when performing realtime scaling and rotation of the figure. For large sets of data this is very useful.

The visualization is handled by passing the mesh-data from the partitioning and the solution vector from the numerical solver into a routine that constructs structures of vertices, indices and color. This data is then passed into an OpenGL Vertex Buffer Object for fast rendering, and surface normals are also constructed for lighting. The final object is then rendered in an application where the user can rotate and zoom as well as apply some transformations in highlighting and coloring.

Figure 6.7 shows a screenshot from the visualization.

6.7 Performance Optimization

Several of the design choices made in the construction of the CUDA kernels were done to make the code apply better to the architecture. When programming for the GPU, one has to take into consideration both the strength and the limitations of the hardware as compared to a CPU.

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. Each multiprocessor on the device has a set of N registers available for use by CUDA thread programs. Maximizing the occupancy can help to cover latency during global memory loads that are followed by a *syncthreads()* call. The occupancy is determined by the amount of shared memory and registers used by each thread block.

Because of this, programmers need to choose the size of thread blocks with care in order to maximize occupancy. The occupancy calculator can assist in choosing the right thread block size based on shared memory and register requirements. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor.

The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously. If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N, the launch will fail. N is 16 384 on the Tesla C1060 and 32 768 on the the Tesla C2050.

Some tricks were performed to push down the number of registers. On some places readability of the code was sacrificed for performance, reusing variables and recasting variables instead of declearing new of a different data type.

For our implementation we would like the block size to be as high as possible. The reason for this is the ratio of internal nodes vs. boundary nodes. For instance, with a block size of 512 the number of internal nodes will be a approximately 4 times higher then the number of boundary nodes. With a block size of 128, the ratio is under 2. Computation of boundary nodes is slower than for internal nodes, and hence we would like the ratio to be high. The memory requirement for an internal node is half that of an edge node, and a quarter of the memory needed for a corner node, due to the storage scheme outlined in figure

6.4. So a smaller blocksize would also negatively effect the need for storage and memory transfers.

The read and write operations between global and shared memory in the CUDA kernels, are performed in a coalesced manner. Threads in a block are structured in warps of 32, and when one thread performs a read/write operation, all other in the same warp must follow. By performing the reads/writes such that all threads work on numbers placed after each other in memory, we can fully take advantage of the memory bandwith.

With the new 2.0 compute capability hardware in the Tesla C2050, part of the shared memory can be utilized as an L1 cache. If too many registers are used in a kernel, variables will spill into the local memory. On older devices this would significantly hurt performance, but with the new devices, the registers instead spill to the L1 cache which is much faster. Reading from global memory should also see som speedup, but since we already have focused on optimal memory reading through coalescing, not much performance is gained from this.

Chapter 7

Results and Discussion

7.1 Introduction

In the following we present various runtime and speedup results for the implementation in chapter 6, as well as comparisons with previous implementations. We then discuss the implications of these results and evalute how they fit with the goals of the thesis.

7.2 Performance

7.2.1 Performance Criterias and Considerations.

Convergence

As a convergence criteria for the conjugate gradient method we have used the expression in 7.1.

$$\textbf{while}((||\underline{r}_k||_2 > \epsilon||\underline{r}_0||_2)\textbf{and}(k < k_{\max})) \tag{7.1}$$

as suggested in [4] where $\underline{r}_0$ and $\underline{r}_k$ are the initial and k'th residual vectors respectively, $k_{\max}$ is a fixed maximum of iterations appropriate for the problem size, and ϵ is the tolerance, in our tests set to $\epsilon = 10^{-8}$.

All the following results were obtained by running the test problems described in chapter 3 for various complexity, or number of degrees of freedom, *NDOF*,

for both the CPU and the GPU-implementations, until the the criteria given in (7.1) was satisfied.

Timing

Timing is done using the CPU time before and after the conjugate gradient solution process. A call to *cudaThreadSynchronize()* is done to ensure that the final GPU kernel executions have all completed before the final clock time is acquired. This is necessary since GPU kernels run asynchronously to the CPU. All runtimes presented are computed as the mean of 10 different runs. Runtime per iteration is simply the runtime of a complete solution process divided by the number of iterations.

Preprocessing such as grid generation and reading of grid and boundary data from file is not included in the timing. Runtime of these events is vanishingly small compared to the solution process, and no emphasis has been put on optimizing these parts of the program.

Time spent on postprocessing, in particular visualization, is also not included. The intital steps of the visualization runs marginally faster for the GPU version of the program, since some of the data is already in the GPU memory space. For our test problems with the time independent heat equation, the memory transfer is only performed once, when the final solution is found, and hence has no real significance for our experiments. For a realtime application of a time dependent solution, the difference could be greater.

Speedup

Speedup is a measure of how much faster a parallel program is compared to a serial/sequential version of the program. It is defined as $S_p = \frac{T_1}{T_p}$, where T_1 is the runtime of the serial version, and T_p is the time for the parallel version running on p processors.

One separates between *relative* and *absolute* speedup. Relative speedup is when T_1 is acquired by running the parallel version of the program with the number of processors $p = 1$. If however T_1 is from a program especially tailored for the sequential architecture, we are talking about absolute speedup.

Doing speedup measurements on a GPU is somewhat different from a regular supercomputer. One would usually vary the number of processors and see how the speedup scales, but the GPU has a fixed number of cores, and does not allow the user to specify and use a subset of the cores. Hence our measurements will be an investigation into how the speedup scales depending on problem size.

We consider the speedup achieved to be an absolute speedup, since the CPU version had to be built from the ground up for the system. Relative speedup is not really an option, since the parallel code for the GPU can not be made to run on the CPU without modifications. We have tried to follow good practices when implementing the serial version, and believe the speedup results achieved are representative of the problem invetigated.

Verification

The numerical solver was controlled for correctness by testing the problems described in figure 3.1 and 3.2, that is the *constant source, zero boundary*, and *non-constant load and boundary* problems.

The relative error is defined as

$$\text{Relative error} = \frac{||u - u_h||_\varepsilon}{||u||_\varepsilon} \tag{7.2}$$

where $||\cdot||_\varepsilon$ is the energy norm defined in equation (3.11), chapter 3. Figure 7.1 shows a log-log plot of the relative error vs. the number of degrees of freedom for the *constant source, zero boundary* problem. The energy norm is computed using the Gauss-Legendre quadrature explained in chapter 3. The linearity with a slope of 0.5 is the expected convergence rate for this problem.

Figure 7.2 shows a visual comparison between the exact solution and the approximate solution for the *non-constant load and boundary* problem.

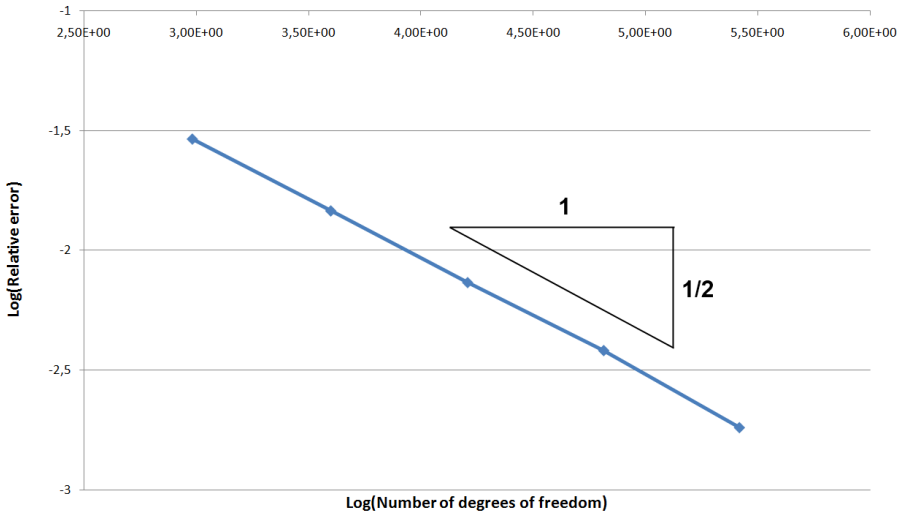


Figure 7.1: The constant source, zero boundary problem: Log-log plot of relative error vs. number of degrees of freedom.

Table 7.1: Hardware Specifications for the Test System.

CPU	Intel Core 2 Quad Q9550 (2.83 GHz)
Memory	2 GB
OS	Linux - Ubuntu 10.04
Compiler	GCC 4.4.1
GPU 1	NVIDIA Tesla C1060
GPU 2	NVIDIA Tesla C2050

7.2.2 Test Results

The final implementation utilizing the complete block-structured approach was tested on two different GPUs, using the same host system. The hardware specifications for the host system can be found in table 7.1

Most of the time spent testing and optimizing the results was done on GPU1 in 7.1, the Tesla C1060. The reason for this is that the other GPU with the new improved hardware architecture just became available at the end of the project. Some tests were still done to see if the program scaled well to future architectures with more cores, and if the other new hardware inventions made a positive impact on performance.

The time results for the serial version of the program developed in chapter 6

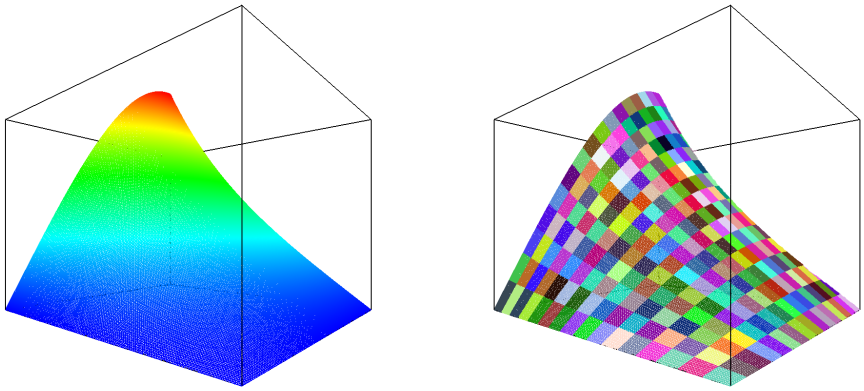


Figure 7.2: The non-constant load and boundary problem: (left) Exact solution, (right) Approximate solution with 32 768 elements.

Table 7.2: The constant source, zero boundary problem: Timing results for the serial version running on the intel CPU.

NDOF	Elements	Total runtime (sec.)	time per iteration (sec.)
961	2 048	0.0100	0.0001
3 969	8 192	0.0600	0.0004
16 129	32 768	0.4800	0.0017
65 025	131 072	4.4200	0.0071
261 121	524 288	38.9300	0.0296
1 046 529	2 097 152	298.9900	0.1136
4 190 209	8 388 608	2 469.6299	0.4627
16 769 025	33 554 432	21 856.8310	1.8172

is shown in table 7.2.

Table 7.5 and 7.6 shows the runtime and speedup results for the parallel version running on GPU 1 and GPU 2 from table 7.1, the Tesla C1060 and the Tesla C2050. Some of the most important specifications of these GPUs are listed in tables 7.3 and 7.4. The speedup results are done against the serial implementation with the results from table 7.2.

Figure 7.3 shows a log-log plot of the runtime per iteration versus *N1* or $\frac{1}{h}$. N1 is the number of elements in the width of the structured finite element mesh over the unit square as shown in figure 7.4.

Table 7.3: Important specifications for the Tesla C1060. (2008)

240 cores on **30** streaming multiprocessors
16kB shared memory.
64kB constant memory.
16384 registers per block.
1.3GHz clock rate.
4GB GDDR3 memory
CUDA Capability Major revision number **1**.
CUDA Capability Minor revision number **3**.

Table 7.4: Important specifications for the Tesla C2050. (2010)

448 cores on **14** streaming multiprocessors
48kB shared memory + **16kB** cache.
64kB constant memory.
32768 registers per block.
1.15GHz clock rate.
3GB GDDR5 memory
CUDA Capability Major revision number **2**.
CUDA Capability Minor revision number **0**.
ECC support

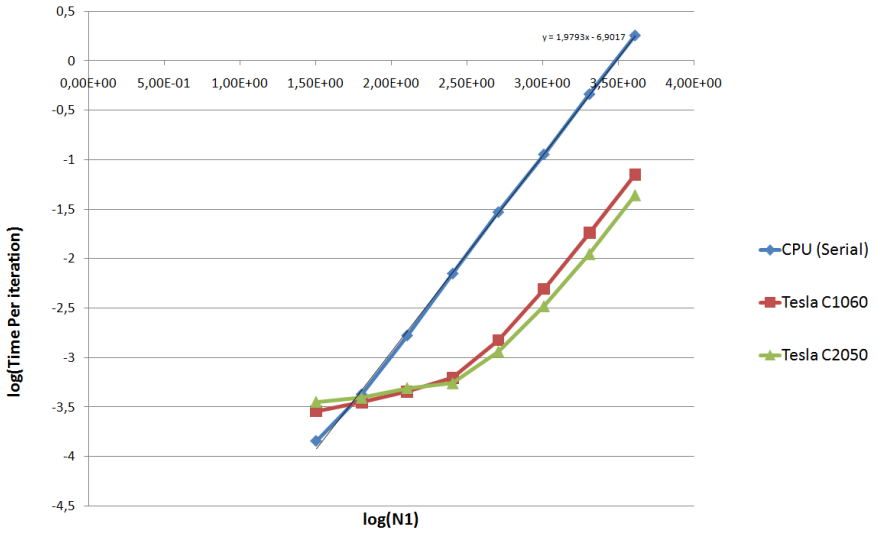


Figure 7.3: The constant source, zero boundary problem: Log-log plot of time per iteration vs. N1, the number of elements along an edge of the square domain.

The graphs in 7.3 clearly demonstrates the superior runtimes on the GPUs for higher problem sizes. As is also visible in the figure, the speedup in table 7.5 shows that for the smallest problem sizes, the CPU version performs best. This is due to the latency involved in transfering data and starting up the kernels on the GPU, which becomes insignificant for the larger problems, and the fact that there are too few partitions to utilize all multiprocessors.

The CPU runtimes shows a clear linearity. This is to be expected since the number of operations for each iteration is asymptotically bound as $O(N1^2)$, and in a log-log plot this results in a line with slope 2. For a full-matrix system, the slope would be 3. The GPU runtimes also display this linear property for larger problem sizes, but the with a lower intercept, indicating the speedup achieved in the GPU version.

Figure 7.5 shows a comparison of the speedup on the Tesla C1060 and the Tesla C2050. The behaviour of the two GPUs is similar, and peak performance seems to be achieved for the same problem sizes, and the curve flattens over the same area. For the larger problem sizes the C2050 performs approximately 1.7 times better than its predecessor.

Table 7.5: The constant source, zero boundary problem: Parallel version running on the Tesla C1060. The table gives the total runtime towards convergence with tolerance 10^{-8}*, time per iteration and speedup compared to the serial version.*

NDOF	total runtime (sec.)	time per iteration (sec.)	Speedup
3 969	0.05	0.00035	1.20
16 129	0.14	0.00045	3.43
65 025	0.41	0.00062	11.61
261 121	1.98	0.00150	19.66
1 046 529	13.58	0.00493	22.02
4 190 209	104.72	0.01822	23.58
16 769 025	925.08	0.07080	23.63

Table 7.6: The constant source, zero boundary problem: Parallel version running on the Tesla C2050. The table gives the total runtime towards convergence with tolerance 10^{-8}*, time per iteration and speedup compared to the serial version.*

NDOF	total runtime (sec.)	time per iteration (sec.)	Speedup
3 969	0.06	0.00039	1.00
16 129	0.16	0.00049	3.00
65 025	0.36	0.00055	12.28
261 121	1.51	0.00115	25.78
1 046 529	8.79	0.00331	34.01
4 190 209	62.12	0.01115	39.76
16 769 025	570.92	0.04392	38.28

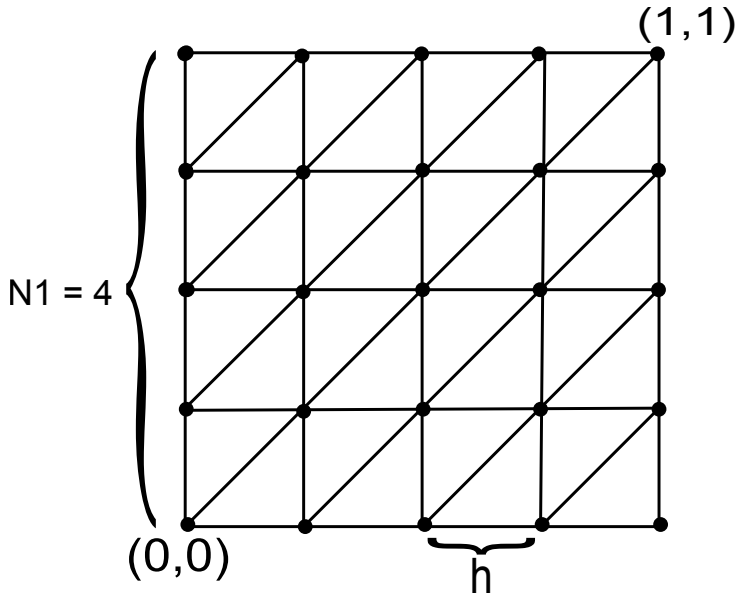


Figure 7.4: Structured finite element mesh over the unit square.

Table 7.7: Comparison between the three different approaches tried over the last year.

	Fall 2009	February 2010	June 2010
Matrix-free	Yes	Yes	Yes
Topologically structured grid	No	Yes	Yes
Topologically structured blocks	No	No	Yes
Max speedup on Tesla C1060	11	18	24
Ratio of longest total runtime	1	0.59	0.50

7.3 Discussion

The work done in [22](appendix A) and [17](appendix B) represents our first inquiries into FEA on GPUs. The trend in our work has been a move towards increasingly more structured representations, better speedup and smaller data amounts and hence the possibility of fitting larger test cases in the GPU memory. The results in the previous section strengthens this trend. Table 7.7 sums up a comparison of the three approches, and figure 7.6 shows the difference in memory requirements.

The difference between the unstructured implementation done in 2009 and the final one done for this thesis is a doubling in speed, even though the latter does a more robust handling of the load function and the boundary conditions. In

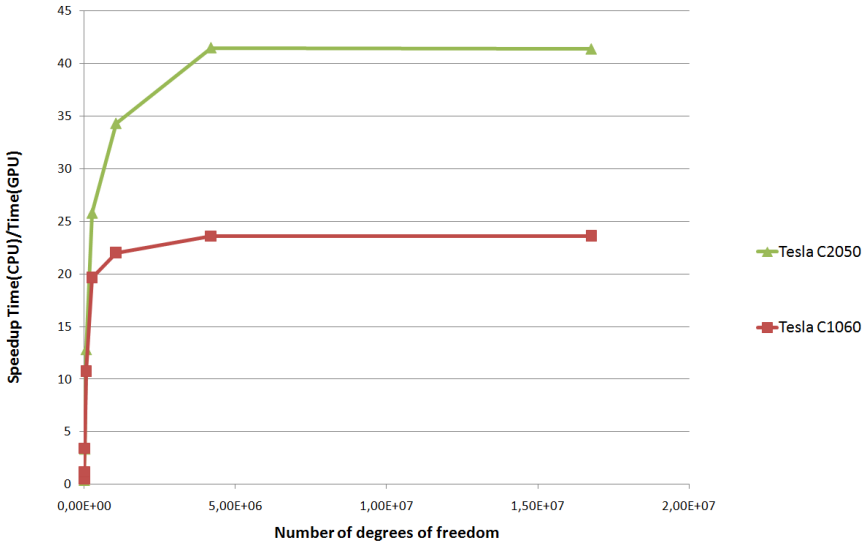


Figure 7.5: The constant source, zero boundary problem: Speedup per iteration vs. degrees of freedom.

terms of optimizations done and overall programming approach, all the versions are similar and hence the increase in performance is strictly due to the structural changes and overall algorithm.

In [1] the writers present a list of what they consider outdated conventional wisdom and their new replacements. The last point on this list replaces the old conventional wisdom that less than linear scaling for a multiprocessor application is a failure, with the new wisdom that with the switch to manycore parallel computing, any speedup via parallelism is a success. We consider our speedup of 24 to be a clearly significant sign that the problem we have investigated is well suited for implementations on the GPU hardware.

Using the small list of parameters in table 5.2 for each block, to completely define the topology and geometry of each block, helped free up the sparse memory resources attributed to each CUDA thread-block, giving a higher rate of occupancy as well as reducing time spent reading and writing data. The simple but strict definitions of global node numbering lets each block be a completely independent unit. This is of course a prerequisite on the GPU since block scheduling must be independent.

A strong trend in hardware development is towards many-core processing units, such as the recently unveiled *Many Integrated Core(MIC)* architecture from In-

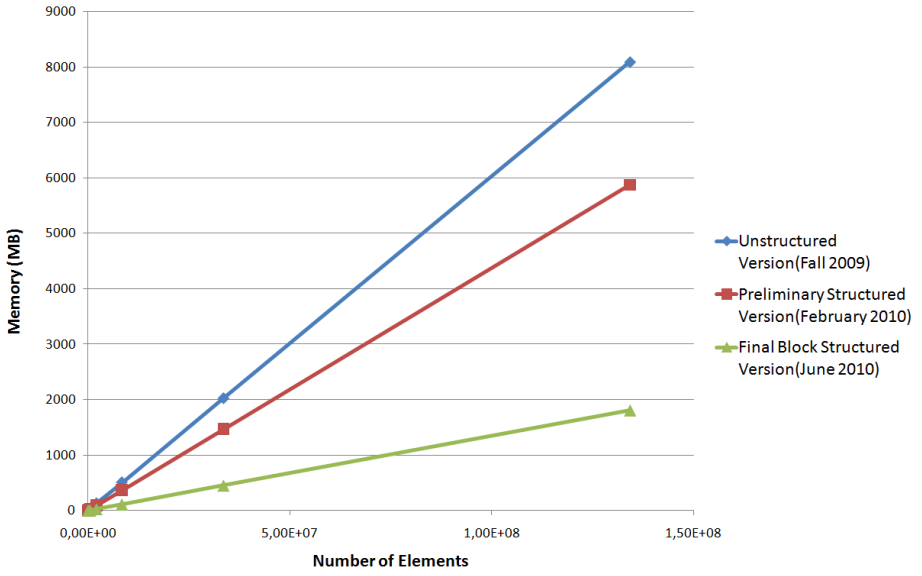


Figure 7.6: Comparison of memory requirements between the various versions of the FEM solver. All versions are tested for various resolutions of elements for the same test problem

tel [9]. Taking this into consideration with the thought expressed in [1] that a modern programming approach to parallel computing must be independent of the number of cores to allow for scaling to future architectures, the approach we have utilized with independent block structures lends itself well to a general approach for hardware architectures similar to the GPU.

Numerical algorithms for many-core architectures must be developed in such a way that performance can be gained simply by moving to an architecture with more cores, and as long as the number of blocks in the grid exceeds the number of cores, scalability should be ensured. Our tests with the newest generation of the Tesla card confirms that this indeed is the case for our programming model, as can be seen in the increase in speedup from 24 to 40.

It is interesting to note that moving from the block structured version developed in february 2010 to the final block structured version with the local computation of elemental indices gives a speedup for the GPU implementations, while the corresponding serial implementation actually runs slightly slover. This demonstrates how an approach suitable for one architecture is not necessarily ideal for another, and in bridging the gap between the specific hardware(GPUs) and the application(FEA), we may be widening the gap for some other hard-

ware.

Chapter 8

Conclusion and Further Work

8.0.1 Conclusion

In this thesis we have tried to show the feasibility of a block-structured approach for doing finite element analysis on GPUs. As is pointed out in [1], the future of processor design probably lies in many-core processors, which makes this approach not only suitable for current GPUs but potentially for mainstream hardware emerging in the future.

The three "iterations" of our work on adapting the finite element method to the GPU programming model, [22], [17] and this thesis, have all been attempts at finding a suitable "bridge" between the two "towers", applications and hardware, as illustrated in figure 1.1 of the introduction. We conclude from our results that the GPU hardware with its brute force but lack of control structures, is well suited for programming approaches that inherently provides the structure needed for correct memory usage and scalability across cores.

Our speedup results show that the effort of adapting numerical algorithms for the GPU architecture can be well worth it if speed is important for the application.

8.0.2 Suggestions for Further work

The finite element solver developed in this thesis is capable of handling grids of completely different shapes than the squares used in the tests, but there was not enough time to develop a grid generator capable of generating more com-

plex geometries. Further work should include a more advanced grid generator to allow for more interesting test cases.

We have used linear triangular elements in our implementation, but the block structured approach can be applied to square elements and elements of a higher order. Furthermore, the spline patch specifications in [20] are for for 3 dimensions, and a natural next step in development would be to expand the solver to handle 3D geometries.

We achieved decent convergence rates with our conjugate gradient solver, but even better convergence could be gained by utilizing a preconditioner, and the block-structure could be enhanced to handle local refinements for an adaptive solution process.

Our experiments with the new fermi architecture from NVIDIA(the Tesla C2050) showed that the program scales well for newer architectures, but some automatic decision parameters could be developed to ensure that full use is taken from future improvements like increased shared memory and greater block size.

Bibliography

[1] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: A view from berkeley. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183*, pages 2006–183, 2006.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM SIGGRAPH 2003 Papers*, page 924. ACM, 2003.

[3] Cecka C., Lew A.J., and Darve E. Assembly of finite element methods on graphics processors. *IntâĂŹl. J. Numerical Methods in Engineering*, 2009.

[4] G.H. Golub and C.F. Van Loan. *Matrix computations*. Johns Hopkins Univ Pr, 1996.

[5] gpgpu.org. General-purpose computation on graphics hardware, webpage. `http://gpgpu.org`, 2010.

[6] Donald Hearn and M. Baker. *Computer Graphics with OpenGL*. Prentice-Hall, third edition, 2004.

[7] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J*, 1952.

[8] T.J.R. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Dover Publications New York, 2000.

[9] intel. Intel unveils new product plans for high-performance computing. `http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm`, 2010.

[10] David B. Kirk and Wen mei W. Whu. *Programming Massively Parallel Processors*. Elsevier Inc., 2010.

[11] Ø.E. Krog and A.C. Elster. Fast GPU-based Fluid Simulations Using SPH. Para 2010 âĂŞ State of the Art in Scientific and Parallel Computing, 2010.

[12] Y.W. Kwon and H. Bang. *The finite element method using MATLAB*. CRC, 2000.

[13] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[14] MIT. Fem for the poisson problem in r2, lecture notes, 2003.

[15] NVIDIA. *NVIDIA CUDA Programming Guide Version 3.0*, 2010.

[16] JT Oden. Finite elements- An introduction. *Handbook of numerical analysis.*, 2:3–15, 1991.

[17] R. Refsnœs and T. Kvamsdal. A matrix-free conjugate gradient implementation for finite element simulations on gpu. 2010.

[18] J. Rodriguez-Navarro and A. Susin. Non structured meshes for Cloth GPU simulation using FEM, 2006.

[19] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

[20] SINTEF. Isogeometric representation - integrating cad and analysis. `http://www.sintef.no/Projectweb/Isogeometric-Analysis/Projects/ICADA/`.

[21] Z. Taylor, M. Cheng, and S. Ourselin. Real-time nonlinear finite element analysis for surgical simulation using graphics processing units. *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2007*, pages 701–708, 2007.

[22] Runar H. Refsnœs Trond Kvamsdal and Gagandeep Singh. Poster on matrix-free conjugate gradient solvers for fem. presented at super computing 2009, 2009.

[23] Barry Wilkinson and Michael Allen. *Parallel Programming*. Prentice-Hall, second edition, 2005.

[24] S.J. Wright and J. Nocedal. *Numerical optimization*. Springer, 2006.

Appendix A

Poster from Supercomputing 2009.

Appendix A contains a poster made for presentation on the showfloor of the SC09, supercomputing conference in Portland, OR USA. The poster sums up the first work on finite element analysis using a matrix-free conjugate gradient method on GPUs. The work was done in the fall of 2009 by Trond Kvamsdal(Supervisor), Runar H. Refsnæs and Gagandeep Singh.

Finite Element Method on GPU Using Matrix-Free Conjugate Gradient Method

NTNU
Norwegian University of Science and Technology

Trond Kvamsdal | Runar H. Refsnæs | Gagandeep Singh

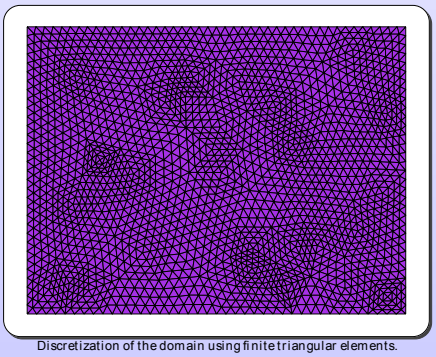
Trondheim, Norway

SC Portland, OR 2009

The Project

- This poster is part of our final year master's project at the Norwegian University of Science and Technology - NTNU.
- Our goal is to investigate the feasibility and potential speed-up of a Matrix-Free Finite Element solver on a GPU using Nvidia's CUDA.
- The main strategy is to partition the domain into smaller subdomains, then distribute each of these subdomains to a seperate block on the GPU, and let every thread on the block calculate the contribution from one element.
- The main features are considered to be a Matrix-Free solver using Conjugate Gradient Method and maximizing speed-up compared to a serial implementation.

The Finite Element Method



Discretization of the domain using finite triangular elements.

- The Finite Element Method (FEM) is a numerical technique for solving partial differential equations (PDE). FEM solves a given PDE by breaking up a problem into small regions, and solutions are found for each region by only taking into account the regions that are right next to the one being solved. Mathematically, it also means discretization of the infinite space in which our solution is found.

Basic Steps in FEM Approach

- Establish the Strong Formulation.
- Obtain the Weak Formulation.
- Choose approximations for the unknown functions.
- Choose the weight functions.
- Solve the system.

- As a model problem, we have used the well-known Poisson Equation. The region on which we solve the equation is arbitrary. The right-hand load vector $f(x,y)$ is chosen to be 1 in order to avoid numerical quadrature and only homogeneous Dirichlet boundary conditions are used for simplicity.

The Poisson Equation: Strong Formulation

$$\nabla^2 u(x,y) = f(x,y) \ , (x,y) \in \Omega$$
$$u(x,y) = 0 \text{ on } \partial\Omega$$

Weak and Final FEM Formulation

- Here we only state the final result of the Weak Formulation.

Weak Formulation

$$a(u,v) \equiv -\int_\Omega \nabla v \cdot \nabla u dA = \int_\Omega f v dA \equiv l(v)$$
$$u, v \in X = \{h | \int_\Omega h^2 dA, \int_\Omega \nabla h \cdot \nabla h dA < \infty \text{ and } h|_{\partial\Omega} = 0\}$$

- The function v is the so-called test function. The last condition in the space X, $h|_{\partial\Omega} = 0$, is required because we have $u = 0$ along the boundary. The Dirichlet Boundary Condition is always reflected in the space X. The unknown function u lies in X, but X is infinite dimensional, so we need to discretize X to approximate u. This is the second big step in FEM.
- Again, we only state the final result. Here we define a subspace X_h of X to be the space of all piecewise continous functions h which restricted to one element (call it T) are linear polynomials in x and y.

Discretization of X

$$X_h = \{h \in X | h|_T \in P_1\}$$
$$P_1 = \{v = a + bx + cy | a,b,c \in R\}$$

- The finite subspace X_h is formed by finding the basis/form functions $\varphi_i(x,y)$. The number of basis functions is the same as the total number of nodes. The properties of each function are: the form function $\varphi_i(x,y)$ associated with node i has the value 1 at (x_i, y_i) and zero at every other node: $\varphi_i(x_j, y_j) = \delta_{ij}$.
- The approximate solution $u(x,y)$ (call it u) can then be written as a linear combination of these form functions. $u = \sum_{i=0}^{N} u_i \varphi_i$. This expression in combination with the Weak Form gives us the final linear system. After applying Boundary Conditions, we can solve the system.

Discretization of X

$$a\left(\sum_{j=1}^N v_j\phi_j, \sum_{i=1}^N \hat{u}_i\phi_i\right) = l\left(\sum_{j=1}^N v_j\phi_j\right)$$
$$= \sum_{j=1}^N a\left(v_j\phi_j, \sum_{i=1}^N \hat{u}_i\phi_i\right)$$
$$= \sum_{j=1}^N \sum_{i=1}^N a(v_j\phi_j, \hat{u}_i\phi_i)$$
$$= \sum_{j=1}^N v_j \sum_{i=1}^N a(\phi_j, \phi_i)\hat{u}_i = \sum_{j=1}^N v_j l(\phi_j)$$

Final Linear System

$$\mathbf{A}\hat{\mathbf{u}} = \mathbf{F}$$

A Matrix-Free CG implementation

$$\underline{r}_0 = \underline{b} - \underline{A}\underline{x}_0, \ \underline{p}_0 = \underline{r}_0$$
For $k = 0, 1, 2, ..$ until convergence
$$\alpha_k = \frac{\underline{r}_k^T \underline{r}_k}{\underline{p}_k^T \underline{A}\underline{p}_k}$$
$$\underline{x}_{k+1} = \underline{x}_k + \alpha_k \underline{p}_k$$
$$\underline{r}_{k+1} = \underline{r}_k - \alpha_k \underline{A}\underline{p}_k$$
$$\beta_k = \frac{\underline{r}_{k+1}^T \underline{r}_{k+1}}{\underline{r}_{k+1}^T \underline{r}_k}$$
$$\underline{p}_{k+1} = \underline{r}_{k+1} + \beta_k \underline{p}_k$$
End

- When solving the linear system, we can as usual form the matrix A explicitly. Another way is to calculate the matrix elements each time they are needed in a calculation, for example as in each iteration when solving the system in the conjugate gradient method. This makes the memory requirements very low, but arithmetic intensity gets higher. This transforms the problem from data intensive to arithmetic intensive, which is very suitable when implementing the solver in CUDA.
- The main reason why it is possible to calculate the matrix elements as and when needed, is that we have explicit expressions for each element, and a well-defined node numbering makes this even easier.
- The important step in our implementation of the algorithm is the matrix-vector product Ap. Most of the implementational details are aimed at making this step efficient through the before mentioned method of never actually forming the matrix. The other steps are performed by CUBLAS library functions.

Partitioning the Mesh

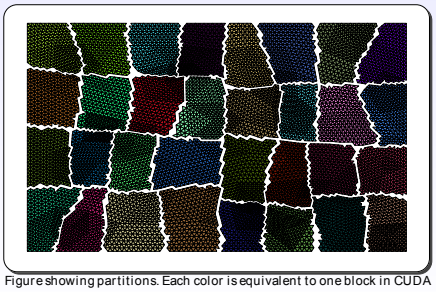


Figure showing partitions. Each color is equivalent to one block in CUDA

- The domain on which our PDE is defined is partitioned into subdomains using the spectral bisection algorithm. The resulting partition is then RESTRUCTURED and ORGANIZED into a form optimized for CUDA. Especially, each thread in every block can read/write its data in a coalesced matter, a reading/writing technique regularly mentioned when considering computation on GPUs.

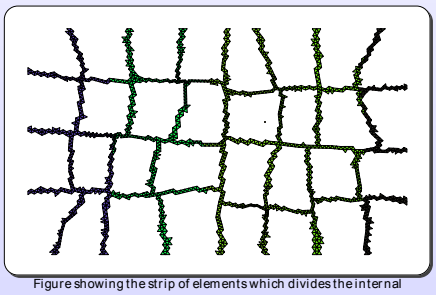


Figure showing the strip of elements which divides the internal partitions. Each color is equivalent to one block in CUDA

- Each partition is associated with a block in CUDA. Between the partitions, a stripe of elements is put to make all the internal blocks independent of each other. The stripe itself is divided into separate partitions, which has to be synchronized because of shared nodes.

Results at The Moment

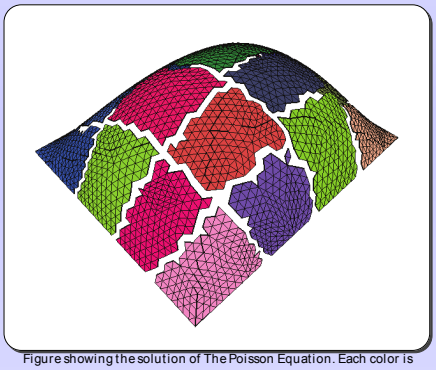


Figure showing the solution of The Poisson Equation. Each color is equivalent to one block in CUDA. The stripe is not shown

- We are finished with more than 2/3 of our project, but have not yet fine-tuned and trimmed our solver. We have not tested the speed-up yet, but the optimizations and the method we have used seems to adapt quite nicely to CUDA. We hope on a significant speed-up compared to a serial version of the Matrix-Free version.
- To get some preliminary results, we testet the untrimmed version of our solver with a serial version on MATLAB implementing the same Matrix-Free version of CG. We testet on 30000 nodes, and got a speed-up of approximately 40×. The program was compiled on NVIDIA GeForce 9800 GX2. It has two GPUs on one chipset, but we used only one of them. The Compute Capability is 1.1 on this card, so we would expect even better performance on a 1.3 architecture. This is because the rules of coalescing on 1.3 are relaxed compared to 1.1.

Further Work

- Since the blocks, except those made from the stripe, are independent of each other, it is possible to solve the problem using multiple GPUs. There is a relatively small size of data that is needed to synchronize between the different GPUs, so the communication overhead will be small compared to the overall work on each GPU.
- Multiple GPUs can only show advantage when the domain on which the PDE is to be solved is large. Large grid size is most likely to require double precision in order to secure convergence, so there are many possibilities on the new-coming FERMI architecture of NVIDIA.

Acknowledgements

We would like to give many thanks to the staff of HPC-Lab at NTNU for giving us the opportunity to travel to SC09 in Portland. Special thanks goes to Anne C. Elster, head of the HPC-Lab, and our supervisor Trond Kvamsdal.

Appendix B

Results from february 2010.

Appendix B contains a summary of the second iteration of work on FEM on GPU using matrix-free conjugate gradient methods, and represents a shift in focus from unstructured meshes to a more block-structured approach.

A Matrix-free Conjugate Gradient Implementation for Finite Element Simulations on GPUs

T. Kvamsdal, R. Refsnæs

February 2010

1 Introduction

The implementation of numerical solution methods on graphical processing units can result in a significant speedup. In this article we implement and examine a matrix-free conjugate gradient solver for the finite element method. The goal is to find the feasibility of a matrix free approach as opposed to assembling the stiffness matrix and solving the linear system.

In the first sections we will be describing the model problem and the numerical techniques applied. We then discuss some of the challenges involved in implementing these techniques on current NVIDIA GPUs, and present our approach to solving them. In the final part we present the time and speedup results achieved, and discuss possible future improvements.

2 Theoretical Background

2.1 The Finite Element Method

Formulation. Our test problem for this project is the Poisson equation on a two-dimensional domain with homogeneous Dirichlet boundary conditions and a load vector $f = 1$. The strong formulation of this problem given in (1), where Ω is the domain, and $\partial\Omega$ is the boundary.

$$\Delta u = \nabla^2 u = f = 1,\ u \in \Omega \in \mathbb{R}^2,\ u = 0 \in \partial\Omega. \tag{1}$$

The finite element method (FEM) is based on the weak, variational formulation of boundary and initial value problems. Multiplying (1) with a test-function $v \in X$ on both sides of the equation, and performing integration-by-parts using the boundary conditions, gives the weak formulation:

Find $u \in X$ such that

$$a(u, v) = l(v),\ \forall\ v \in X\ , \tag{2}$$

where

$$X = \{v \in H^1(\Omega) | v|_{\partial\Omega} = 0\} \equiv H_0^1(\Omega)\ ,$$

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v d\Omega = \int_\Omega f v d\Omega = l(v)$$

$$H^1(\Omega) = \left\{ v \mid \int_\Omega v^2, \int_\Omega {v_x}^2, \int_\Omega {v_y}^2 < +\infty \right\}$$

Discretization. In order to solve our system numerically, the space X must be discretized. We have chosen linear triangular elements in our treatment of the method. This gives the domain:

$$\overline{\Omega} = \bigcup_{T_h \in \tau_h} \overline{T}_h$$

where T_h is a triangle in the triangulation τ_h.

With piecewise linear basis functions, the discretized function space becomes

$$X_h = \{v \in X |\ v|_{T_h} \in \mathbb{P}_1(T_h),\ \forall T_h \in \tau_h\}$$

That is, all functions v in the space X that are linear polynomials over each element in our

triangulation. As a basis for X_h we choose a nodal basis, such that

$$v(\mathbf{x}_j) = \sum_{i=1}^{n} v_i \phi_i(\mathbf{x}_j) = \sum_{i=1}^{n} v_i \delta_{ij}.$$

where $\mathbf{x}_j$ are the nodes. The space X_h can then be written as

$$X_h = \text{span}\{\phi_1, ..., \phi_n\} :$$
$$\phi_i \in X_h,\ \phi_i(x_j) = \delta_{ij},\ 1 \leq i, j \leq n$$

By expressing u and v in terms of the basis functions we get the final discrete formulation:

Find $u_h \in X_h$ such that

$$a(u_h, v) = l(v)\ \forall v \in X_h \tag{3}$$

This leads to the set of algebraic equations in (4)

$$\underline{A}_h \underline{u}_h = \underline{F}_h$$
$$A_{hij} = a(\phi_i, \phi_j) = \int_\Omega \frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y}\frac{\partial \phi_j}{\partial y} d\Omega \tag{4}$$
$$F_{hi} = l(\phi_i)\ ,\ 1 \leq i \leq n.$$

$\underline{u}_h$ is the vector of nodal values of u_h.

Assembly. We want to compute the integral over each element T_h^k in the triangulation of the domain.

$$\int_{T_h^k} \frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y}\frac{\partial \phi_j}{\partial y} dA \tag{5}$$

In terms of programming, it is easier to calculate the integral (5) by use of substitution. We define a linear and affine mapping between T_h^k and a reference element $\widetilde{T_h^k}$ as shown in figure 1.

Error Analysis. A general result on the error can be found in equation 6.

$$|||u - u_h||| = \inf_{w_h \in X_h} |||u - w_h||| \tag{6}$$

Equation (6) shows that the error $u - u_h$ between the exact solution u of (2) and the solution u_h of the discretized problem is the smallest possible in the energy norm. In other words

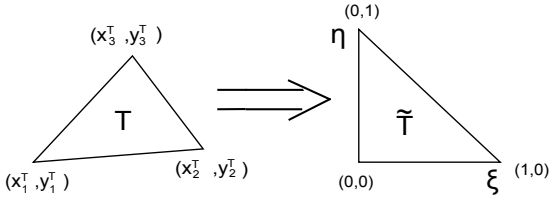


Figure 1: Mapping between physical and reference element.

u_h is the optimal choice of all $w_h \in X_h$, it is the projection of u on X_h. The energy norm is defined as

$$|||v|||^2 = a(v, v) = |v|^2_{H^1\Omega} = \int_\Omega |\nabla v|^2$$

.

2.2 The Conjugate Gradient Method

The discrete system of equations from the finite element method must be solved by a numerical solver. The Conjugate Gradient method was originally proposed by Magnus R. Hestenes and Eduard Stiefel in 1952 [3] as a method for solving systems of linear equations. The method requires a symmetric positive-definite system of equations, that is a system of the form $\underline{Ax} = \underline{b}$ where $\underline{A}$ is such that $\underline{x}'\underline{Ax} > 0\ \forall \underline{x} \in \Omega$. The stiffness matrix from the finite element method applied to the Poisson equation satisfies this requirement, given the elliptic nature of the Laplace operator.

The conjugate gradient algorithm in its standard form as given in [5] is stated in algorithm 1. $\underline{r_0}$ and $\underline{p_0}$ are respectively the initial residual vector and the initial search direction. When the initial guess $\underline{x_0}$ is $\underline{0}$, $\underline{r_0}$ and $\underline{p_0}$ simply becomes $\underline{b}$ which for the finite element method is the right hand side of (4), $\underline{F}_h$. α_k is the steplength, and β_k is used to determine the next search direction.

Our implementation of the finite element method never forms the actual stiffnes matrix $\underline{A}$, and hence the matrix-vector product $\underline{Ap}_k$ in

Algorithm 1 The Conjugate Gradient Method

$\underline{r}_0 = \underline{b} - \underline{A}\underline{x}_0$, $\underline{p}_0 = \underline{r}_0$
For $k = 0, 1, 2, ..$until convergence
$\alpha_k = \frac{\underline{r}_k{}^T \underline{r}_k}{\underline{p}_k{}^T \underline{A}\underline{p}_k}$
$\underline{x}_{k+1} = \underline{x}_k + \alpha_k \underline{p}_k$
$\underline{r}_{k+1} = \underline{r}_k - \alpha_k \underline{A}\underline{p}_k$
$\beta_k = \frac{\underline{r}_{k+1}{}^T \underline{r}_{k+1}}{\underline{r}_{k+1}^T \underline{r}_k}$
$\underline{p}_{k+1} = \underline{r}_{k+1} + \beta_k \underline{p}_k$
End

the algorithm is performed implicitly by doing the calculations per-element as can be seen in algorithm 2. θ is the local-to-global mapping of indices. The terms $A_{\alpha\beta}$ are the entries in the elemental matrix $\underline{A}^k$.

Algorithm 2 Evaluation of $\underline{y} = \underline{A}\underline{p}_k$ without explicit contruction of A.

$\underline{y} = 0$
For $k = 1, 2, ..K$(elements)
For $\alpha = 1, 2, 3$
$i = \theta(k, \alpha)$
For $\beta = 1, 2, 3$
$j = \theta(k, \beta)$
$y_i = y_i + A^k_{\alpha\beta} p_j$
End
End
End

In the GPU-implementation, the outer for-loop in algorithm 2 is replaced by a parallel distribution where calculations for each element are done on one thread.

Theoretically the Conjugate Gradient method will converge and reach the exact solution in at most N steps for N degrees of freedom. Each step k projects the exact solution into the k-dimensional solution space spanned by the A-conjugate basis vectors.

In practice, each step will not be solved exactly due to round-off error in the computations. The method will however in general converge to an acceptable error-tolerance in far less than N iterations. This rapid convergence is one of the greatest strengths of the method.

It can be shown [5] that the convergence in the A-norm is given as in equation (7), where $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ is the condition number of $\underline{A}$, $\underline{x}_*$ is the exact solution, and $\underline{x}_k$ is the approximate solution after k steps.

$$\|\underline{x}_* - \underline{x}_k\|_A \leq 2 \left[\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right]^k \|\underline{x}_* - \underline{x}_0\|_A \quad (7)$$

3 The CUDA Architecture

Launched in 2007, the CUDA architecture from NVIDIA was an attempt to provide a more accessible approach to programming of scientific applications on GPUs. Some of the advantages offered over traditional general-purpose programming on GPUs includes an exposed shared memory structure and scattered reads from arbitrary addresses in memory.

Development on the CUDA architecture can be done with two different interfaces. The first is *C for CUDA* which consists of a small set of extensions to the C language, and is the easiest to pick up for people accustomed to C programming. The other interface is the *CUDA driver API* that provides a more low-level approach, allowing for a better level of control, but also requires more code and is harder to program. Our implementation was done using *C for CUDA*.

3.1 Thread Hierarchy

The smallest unit of parallelization in CUDA is the *thread*, which is a small low-maintainance process running on a single streaming processor core. In *C for CUDA* the programmer defines functions, or *kernels* that run in parallel on a set number of threads. Multiple threads are gathered in *thread blocks*, and are given a *thread-id* which can be used for conditioning in the kernel. A thread block can have up to 512 threads on current NVIDIA GPUs, and the entire block must run on the same multiprocessor. Synchronization of threads in thread

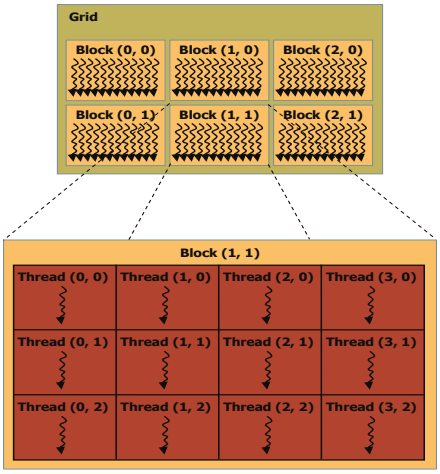


Figure 2: Schematic illustration of the GPUs grid structure. From [4]

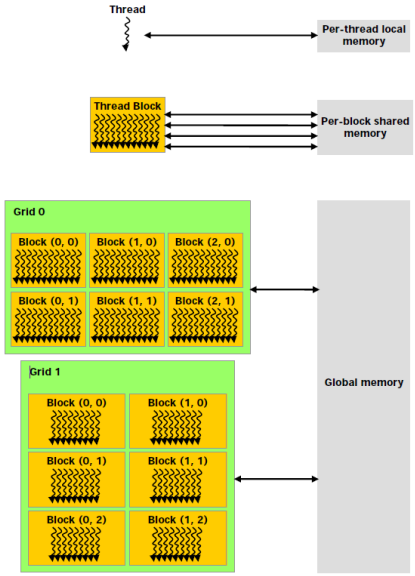


Figure 3: Schematic illustration of the GPUs memory structure. From [4]

block is done with a call to *__syncthreads()*. Thread blocks are gathered in *grids*. Blocks in a grid must run independently, and the only synchronization possible is achieved after a kernel has run, and all blocks have finished their work. The thread hierarchy is illustrated in figure 2.

3.2 Memory Structure

The memory structures that are part of the CUDA architecture are designed to work well with the parallelization structure mentioned above. All threads have access to a private local memory. Threads in a block will in most programs have to exchange data, and a shared memory of maximum size 16kB for each block allows for this. The shared memory will have the same lifetime as the block it is attached to, while the global memory is persistent throughout the program, and hence can be used for communication between different kernels.

In addition to the shared and global memory spaces, threads have read access to the texture memory and the constant memory. The constant memory is cached and so it is very fast. It is however very limited in space, only 64kb, and is hence of limited use. The host and the device will maintain different memory spaces, and pointers from one, can not be dereferenced in the other. Before executing a kernel function, the necessary memory must be allocated on the GPU, and data must be transferred from the host system. This transfer goes over the PCIe interface. Figure 3 shows the various memory spaces available for a CUDA program, and how they relate to each other.

4 Implementation

4.1 Challenges

Shared Memory. In order to achieve good performance, thread blocks should take advantage of the shared memory. Typically the data set needed for a block is read from the global memory into the shared memory once, and all consecutive operations are done in the shared and local memory space. The final result is then written back for storage in the global memory. Reading and writing to the global memory should be *coalesced*, and this put demands on the structuring of the data

from the finite element mesh.

Synchronisation A minimum of communication between blocks in a partitioned finite-element problem is unavoidable. The CUDA architecture has no support for message-passing or synchronization between thread-blocks, and all communication must be done through the global memory. Additional data structures are hence needed to collect the correct data on the boundaries of each block. Communication between threads in a thread block is also a problem, as race conditions means that additional work must be put into the reduction of data.

Double Precision. Double-precision is cheap on a CPU, while on the GPU most of the processing units are dedicated to single precision computations. On the Tesla C1060, the ratio is 8 to 1. In our implementation we have only used single precision in both the GPU and CPU version of the program.

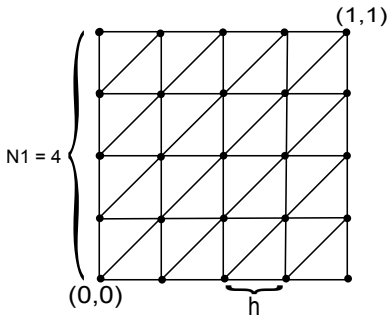


Figure 4: A structured finite element mesh.

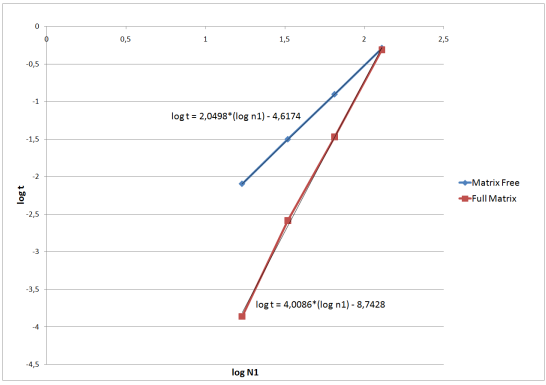


Figure 5: Log-log plot of iteration time for implementations of the CG algorithm in MATLAB.

4.2 Why matrix-free?

The main advantage of a matrix-free implementation of the conjugate gradient method is not having to store the matrix. For a structured finite element mesh such as the one in figure 4, with $N1 + 1$ nodes in each direction, the storage requirement for a full matrix grows as $O(N1^4)$. For a mesh with 257 * 257 nodes, the matrix requires more than 16 GB of data, 4 times the available memory on the Tesla C1060 GPU. By comparison the same mesh is one of our smallest test cases in the GPU speed tests. Our biggest mesh has over 250 times the amount of nodes. Storing the full matrix for that system would require more than 1000 TB.

The operation count of the matrix-vector product $\underline{Ap}_k$ also scales as $O(N1^4)$ for the full matrix system. The matrix-free system requires $O(N1^2)$ calculations. The rest of the work done in the CG-iteration are primarily inner-products that scales as $O(N1)$ This means that runtime is bound by the matrix-vector product.

Serial implementations of the matrix-free, and the full matrix conjugate gradient algorithms were done in MATLAB to illustrate the differences between the two approaches. Figure 5 shows a log-log plot of the time per iteration versus $N1$(or $\frac{1}{h}$), for both the matrix-free and the full matrix case. The respective $O(N1^2)$ and $O(N1^4)$ growth in runtime turns in to linear relations with slope 2 and 4 as can be seen in the figure, and for increasing problem sizes the matrix-free approach clearly becomes the fastest.

In reality an assembled implementation would probably use a sparse matrix format. But the matrix-free implementation is easier to implement. For our test problem with the structured mesh, an implementation could be relatively easy to implement, but general meshes resulting in unstructured matrices would be more difficult, and would suffer from indirect adressing.

4.3 Race conditions

The different nodes in a partition can be shared by up to 6 surrounding elements. When all these elements run on seperate threads in a thread block, race conditions will occur when these threads try to write to the same address in the shared memory. To avoid this, memory is allocated in the shared memory for all contributions to the node, and these are then collected and summed up once all contributions are calculated.

An alternate strategy to avoid race conditions is by coloring the elements in such a way that no elements sharing a node have the same color, and then running the computations in passes based on colors. This approach is explored in [1].

4.4 Partitioning

We have focused on a structured quadratic mesh in our implementation, and was hence able to construct a very simple mesh generation and partitioning program with perfect load balancing. All partitions have the same amount of elements and nodes, and share only entire boundary edges. The resulting data is made up of two arrays P and T, containing respectively the nodal coordinates and the coordinate indices for the elements. The arrays are structured so that the $3 \cdot N \cdot (i-1), ..., 3 \cdot N \cdot (i-1) + 3 \cdot t$ entries in T are the indices for elements $t = 1$ to N in partition i with N elements in each partition, and the $2 \cdot M \cdot (i-1), ..., 2 \cdot M \cdot (i-1) + 2 \cdot p$ entries in P are nodal coordinates for all nodes $p = 1$ to M in partion i with M nodes in each partition.

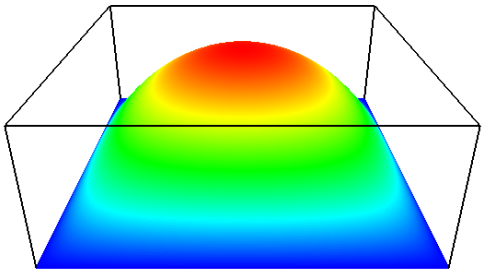


Figure 6: A solution of the Poisson problem on a mesh with 131 072 elements.

Each partition has its own copy of the coordinates of nodes it shares on the boarder to another partition. The reason for this redundant data is to ensure faster and coalesced reading when blocks on the GPU transfers data from P and T in the global memory into the shared memory reserved for that particular partition. The storage size for an unpartitioned mesh with 8.4 million elements stored without redundancy is 97% of a partitioned mesh, so the additional memory needed is negligable.

4.5 Conjugate Gradient Iteration

Before starting the conjugate gradient iteration, the needed arrays are allocated on the GPU, the load vector F is calculated, and the residual vector r_0 and search direction p_k are initialized. The calculation of F is done in parallel in a CUDA kernel function. The general procedure follows the steps described in algorithm 3, but with different mathematical expressions.

Matrix-Vector Product. The output from the CUDA Visual Profiler shows that most of the runtime is spent in the two kernels computing the matrix vector product. In the first kernel each element resides on a single thread in the thread block. A thread block corresponds to a block in the partitioning. The necessary data for the block is read into

shared memory and elemental contributions for a node is written in a unique place in memory adjacent to contributions from elements sharing the same node. All threads in the thread block are then synched by a call to *_syncthreads()*, ensuring that all threads in the block are done calculating and storing their contributions. The values stored for each node are then summed up and stored in the global memory space.

The second kernel involved in calculating the matrix-vector product has as its sole task to gather nodal values on the borders between partitions. This can be done by using a look-up table created in the partitioning step, containing border indices. Our implementation takes advantage of the structure of the mesh and the simple indexing relationship between partitions.

Additional Calculations. The additional innerproducts and SAXPY operations of each conjugate gradient iteration step in algorithm 1 are performed by utilizing the CUBLAS linear algebra library provided with the CUDA distribution.

For smaller problem sizes, the GPU runtimes are dominated by the start-up of the CUBLAS library, initiated by the *cublasInit()* function call. The initialization alone takes about 0.33 seconds. We have not included the *cublasInit()* call in our results since a typical application solving multiple problems would only need to run the function once, assosiating the added time only with the first run. It is worth mentioning that implementing our own linear algebra functions would eliminate this cost, but would probably perform worse overall.

5 Results

5.1 Hardware

All tests of the serial version of the program were done on a Linux 2.6.31 system with 2 GB of RAM and an Intel Core 2 Quad Q9559 processor with 2.83 Ghz, running on one core. The GPU version of the program was run on the same host system as above, but the GPU-specific parts were run on a NVIDIA Tesla C1060. The Tesla C1060 has 4 GB of DDR3 embedded memory and 240 streaming processor cores with a frequency of 1.3 GHz.

5.2 Convergence

As a convergence criteria we have used the expression in 8.

$$\textbf{while}((||\underline{r}_k||_2 > \epsilon||\underline{r}_0||_2)\textbf{and}(k < k_{\max})) \quad (8)$$

as suggested in [2] where $\underline{r}_0$ and $\underline{r}_k$ are the initial and k'th residual vectors respectively, $k_{\max}$ is a fixed maximum of iterations appropriate for the problem size, and ϵ is the tolerance, in our tests set to $\epsilon = 10^{-8}$.
All results were obtained by running problems of different complexity, or number of degrees of freedom, *NDOF*, for both the CPU and the GPU-implementations, until the the criteria given in (8) is satisfied.

5.3 Timing and speedup

Only the actual solving of the system is included in the timing results. Grid generation is not included due to the fact that its runtime is negligable compared to the solution step, and optimization of the grid generation has also not been the focus of the study. Visualization is for similar reasons not included, but in practice the visualization step is slightly faster for the GPU version given that most data needed is already on the GPU, while in the CPU case it must be transfered via the PCIe bus. For a time dependent solution with visualization done in realtime, this advantage would propably be more significant.

Table 1 lists the test results. t_{CPU} and t_{GPU} are the runtimes for the serial CPU version and the paralell GPU version respectively. *NDOF* is the problem size in degrees of freedom, and the speedup is given as $\frac{t_{\text{CPU}}}{t_{\text{GPU}}}$.

Figure 7 shows the speedup as a function of problem size, and figure 8 shows a log-log plot

Table 1: Total runtime and speedup results for various problem sizes.

NDOF	t_{CPU} (sec.)	t_{GPU} (sec.)	Speedup
3969	0.03	0.03	1.00
16129	0.26	0.09	2.89
65025	3.71	0.36	10.31
261121	31.60	2.03	15.57
1046529	267.61	15.52	17.24
4190209	2238.88	121.70	18.40
16769025	19123.90	1095.69	17.45

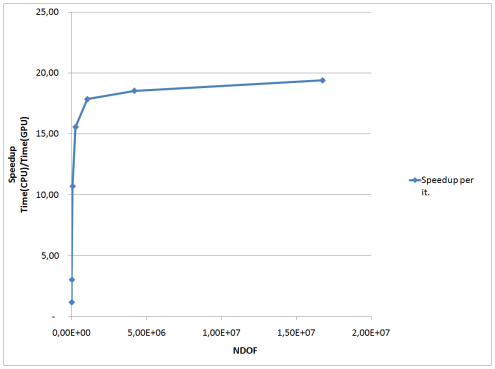


Figure 7: Speedup per iteration vs. degrees of freedom.

of runtime results per iteration vs. *N1* for both the serial and the parallel GPU version. The $O(N1^2)$ behaviour of the runtime is clearly visible for both the GPU and the CPU version, similar to the plot of the matrix-free MATLAB program in figure 5. While the slopes of the two plots in 8 are the same, the constant term is higher for the CPU version, illustrating the speedup achieved in the GPU version.

6 Conclusions and future work

With our GPU implementation of the matrix-free method, we have achieved a significant speedup of approximately 18 over the CPU version, for large test problems. Given the relative ease of implementation and the small memory requirements as compared to other

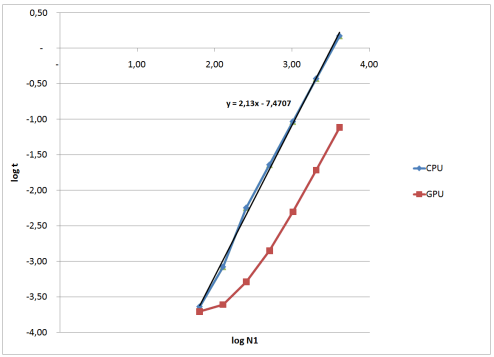


Figure 8: Log-log plot of time per iteration vs. *N1*.

methods, we consider our work to be a viable approach to solving finite element simulations on GPU systems.

A natural extension of our work would be to allow for even greater problem sizes, by running the program on multiple GPUs. Support for unstructured domains can be achieved by mapping to deformed geometries, or by modifying the code to handle non-uniform blocks.

References

[1] C. Cecka, A.J. Lew, and E. Darve. Assembly of Finite Element Methods on Graphics Processors. 2000.

[2] G.H. Golub and C.F. Van Loan. *Matrix computations*. Johns Hopkins Univ Pr, 1996.

[3] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J*, 1952.

[4] NVIDIA. *NVIDIA CUDA Programming Guide Version 2.3.1*, 2009.

[5] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

Appendix C

ICADA - Spline Patch Specifications.

Appendix C contains the spline patch specifications for the ICADA framework developed by SINTEF.

ICADA - Note 2010-001:

Specification of spline patch topology to SIM

Knut Morten Okstad and Trond Kvamsdal

June 23, 2010

1 Assumptions

We only allow for completely matching blocks for now. That is, two adjacent blocks are topologically connected with a common surface if, and only if, all control points (nodes) defining the geometry of the surface are identical for the two blocks. Otherwise, they are assumed not connected (we have a "crack").

2 Input description, topology

The simulation module must generate a unique global node number for all control points in the model. The input to this process should be as compact as possible, but should contain sufficient information such that the node number generation can be carried out locally within each patch in arbitrary order or in parallel. Only topology information is included, all geometry data goes via the g2-files from GoTools.

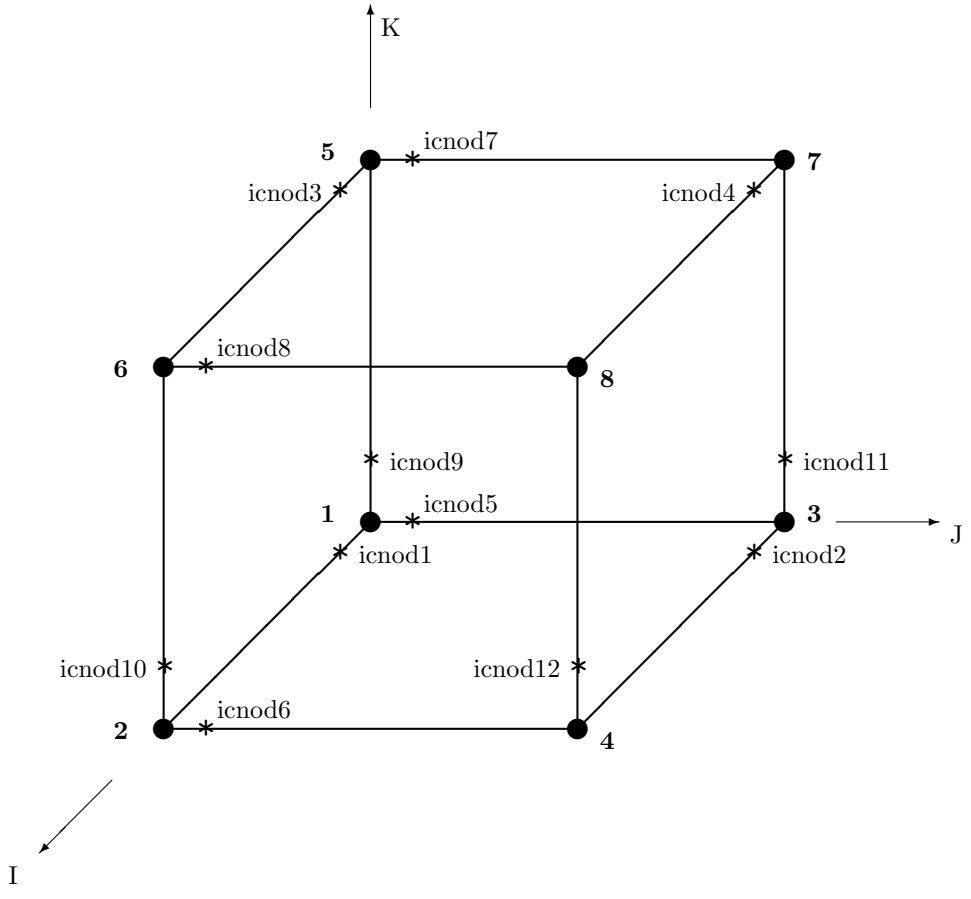


Figure 1: Local vertex numbering convention (•) and the first node along each edge (⋆).

For each patch, the input is as follows (see Figures 1, 2 and 3 for the interpretation of the variables ICNODi, ISNODi and IINODi):

```
IBLOCK
IBNOD1   IBNOD2   ...   IBNOD8
ICNOD1   INCR1
ICNOD2   INCR2
...
ICNOD12  INCR12
ISNOD1   INCI1    INCJ1
ISNOD2   INCI2    INCJ2
...
ISNOD6   INCI6    INCJ6
IINOD1
```

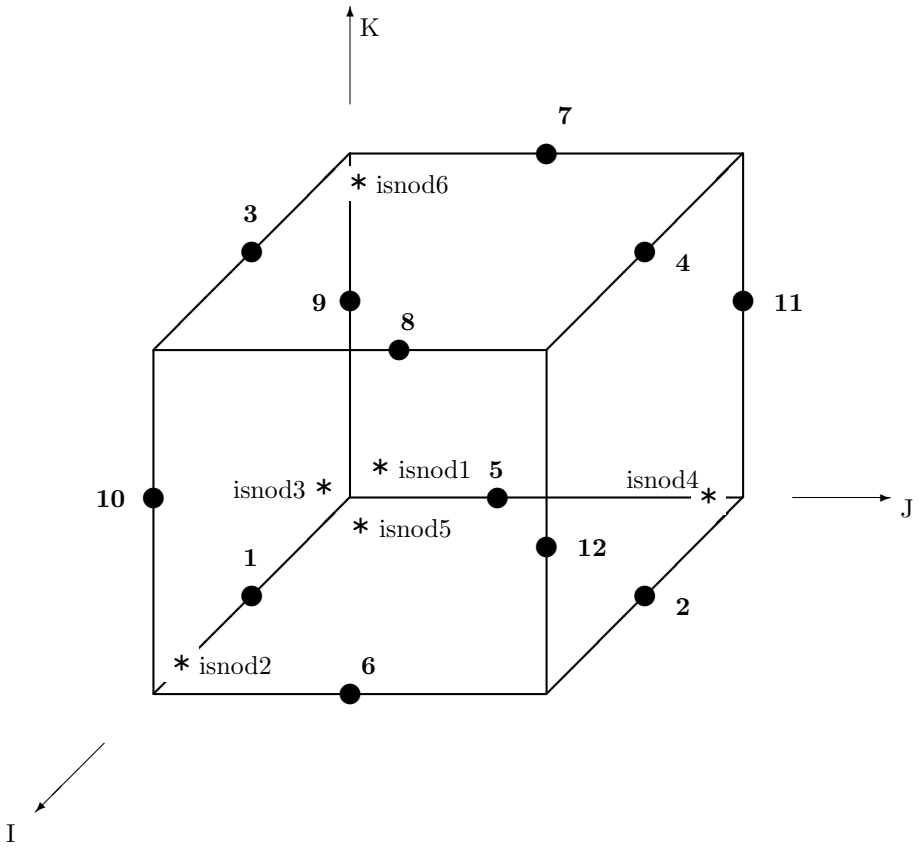


Figure 2: Local edge numbering convention ($\bullet$) and the first node on each surface ($\star$).

IBLOCK Spline patch index

IBNODi Global node number of vertex i

ICNODi Global node number of second point along edge i

INCRi Increment in global numbering along the edge (± 1)

ISNODi Global node number of first interior point on face i

INCIi Increment in global numbering in local I-direction on the face (± 1)

INCJi Increment in global numbering in local J-direction on the face (± 1)

IINOD1 Global node number of the first interior point of the patch

The local I and J directions for a face are defined as the two remaining directions when removing the index defining the normal direction of that face from the $I-J-K$ triplet. That is, for local faces 1 and 2, the local $I-J$ directions correspond the "global" $J-K$ directions (depicted in Figure 3). For local faces 3 and 4, the local $I-J$ directions correspond to the "global" $I-K$ directions, respectively, whereas for local faces 5 and 6 they coincide with the global $I-J$ directions.

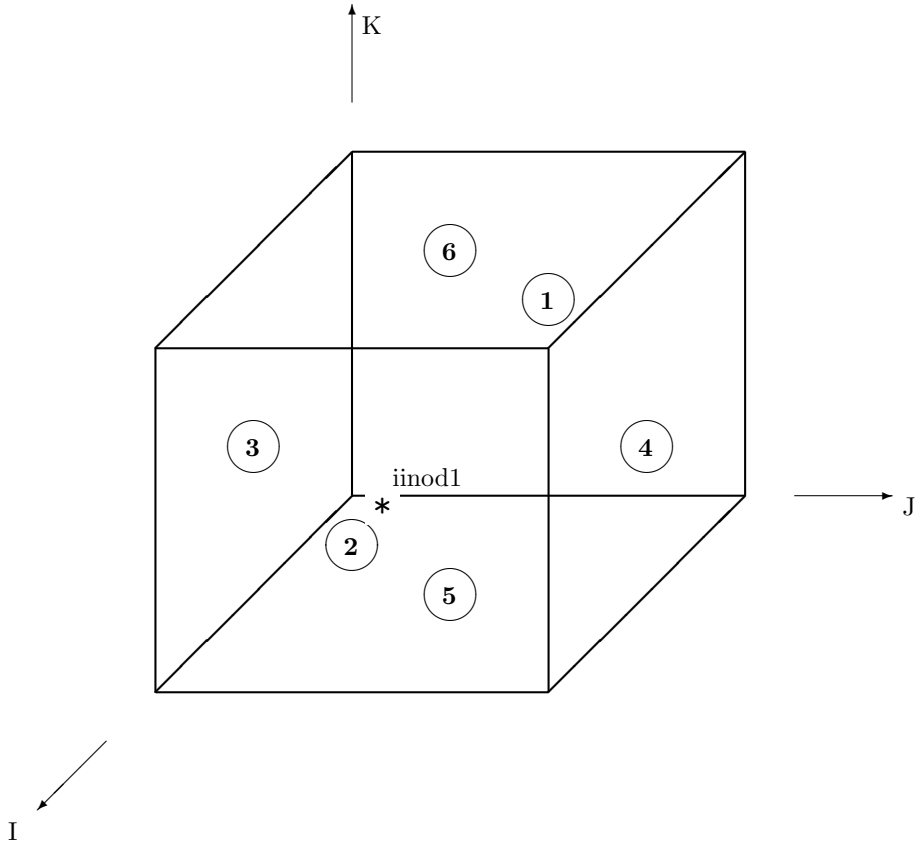


Figure 3: Local face numbering convention ($\circ$) and the first interior node ($\star$).

3 Properties and boundary conditions

All physical properties defined in the GPM-module are mapped onto the spline model through a set of user-defined codes. The actual interpretation of each code is defined within SIM itself, via a separate input file. The property codes are specified through the following syntax:

PCODE	IBLOCK	LDIM	LINDEX

PCODE Property or boundary condition code (either a string or an integer value)

IBLOCK Spline patch index

LDIM Local entity dimension flag (0, 1, 2, or 3)

LINDEX Local entity index which is assigned the property

- Local vertex if LDIM = 0
- Local edge if LDIM = 1
- Local face if LDIM = 2
- Not referenced if LDIM = 3

The local ordering of the vertices, edges and faces follows the convention defined in Figures 1, 2 and 3, respectively.