UNIVERSITY OF OSLO
Department of Informatics

GPU accelerating the FEniCS Project

Master's thesis

Fredrik Heffer Valdmanis

May 2, 2012

[simula . research laboratory]

Abstract

In the recent years, the graphics processing unit (GPU) has emerged as a popular platform for performing general purpose calculations. The high computational power of the GPU has made it an attractive accelerator for achieving increased performance of computer programs. Although GPU programming has become more tangible over the years, it is still challenging to program GPUs efficiently to achieve good performance.

The FEniCS Project is a collection of software components for the automated solution of partial differential equations. It allows users to specify input problems in a high-level, domain specific language. Novel techniques such as code generation and just-in-time compilation is used to automate the implementation of complicated code. This ensures computational efficiency of the software while retaining a high-level user interface that accepts a wide range of problems as input. To further increase the performance of the software, it is of interest to accelerate certain parts of it via GPUs.

This thesis presents a GPU-accelerated implementation of the FEniCS Project. By integrating GPU-accelerated libraries for linear algebra, the solution of linear systems of equations is accelerated. With this implementation, the rate of floating point operations performed on the GPU is up to six times higher than that of the CPU. This leads to significant decreases in the time required to solve the linear systems. Throughout, emphasis was placed on keeping the resulting implementation clean and easy to maintain for the software developers. In the spirit of the FEniCS Project, the user interface remains easy to use, with simple parameters for activating GPU acceleration. The distribution and compilation of the software is kept simple by extending the FEniCS build system.

Through this implementation, it is shown that integrating GPU-accelerated libraries in an existing software system is a viable approach to increase the system's performance. It makes it possible to keep the overall structure and layout of the system intact while still utilizing the computational power of the GPU.

Acknowledgements

First, I would like to thank my supervisors Marie Rognes and Anders Logg for providing excellent guidance throughout the work with this thesis. I also thank my fellow students for making the work with this thesis even more enjoyable. I would also like to thank professors Hans Petter Langtangen and Knut Mørken for introducing me to numerical methods and scientific computing through their excellent courses.

Last, but not least, thanks to my beautiful Rebekka for her infinite love and support.

Fredrik Heffer Valdmanis
Oslo, May 2012

Contents

1 Introduction

1.1 Problem statement

In the field of scientific computing, programmers have always striven for high performance. Fast running code is crucial to be able to solve larger problems, solve problems of a fixed size faster and implement more complex algorithms. During the last decade, the graphics processing unit (GPU) has shown itself to be a powerful accelerator for providing increased performance of general purpose programs. Starting out as a field for experts and enthusiasts, GPU programming has become more tangible over the years. However, programming GPUs effectively to achieve good performance is still very much a challenging and time-consuming task.

The FEniCS Project [12, 76] is a collaborative effort to develop a set of tools for automated scientific computing. The aim is to provide a framework where the user can specify any partial differential equation through a simple, high-level user interface and have its solution computed and visualized with a minimum of programming effort and computational resources. This is achieved through innovative techniques such as domain-specific languages and code generation. The FEniCS Project has support for parallel execution with both distributed and shared memory via MPI [69] and OpenMP [77]. However, support for parallel computing on GPUs is not present in the project yet.

As such, a seamless integration of GPU computing in the FEniCS Project is of definite interest for the scientific computing community. The overall goal of this project is as follows:

> *Integrate GPU computing in key components of the FEniCS Project and evaluate the performance of the resulting implementation. The final implementation should be made generally available as part of the FEniCS Project.*

It should be emphasized that a prototype implementation is not sufficient: the provided code must be production-ready, in the sense that the code is well-designed, tested, benchmarked and documented.

Two possible approaches to solve this problem are either to hand-code the desired functionality using GPU computing programming languages, or to utilize GPU-accelerated libraries in appropriate parts of the software. The approach chosen will affect the following aspects of the software:

Maintenance
How complex it will be to maintain and extend the resulting implementation.

Backward compatibility
How to maintain backward compatibility and at the same time take advantage of technological advances as programming APIs and hardware evolve.

Overall quality of implementation
How much effort it requires to provide high-quality, production-ready code.

Functionality
How much functionality it can be expected that it is possible to implement.

User interface
How the user interface will be affected by the inclusion of GPU support.

Compilation and distribution
How the compilation and distribution of the software will be affected by the inclusion of GPU support.

The latter two aspects are in practice completely under the control of the developer, and will be discussed in Chapter 8. The former four, on the other hand, are to a large extent affected by the approach taken. These will be discussed in further detail below.

In the case of maintenance and further development, much of the work will be taken care of by external developers if one chooses to use a library. When using a library, intimate knowledge about GPU computing programming APIs and the writing of GPU code is only needed by the library developers. They will be concerned with the low-level, performance-critical technicalities, which the users of the library will benefit from.

When it comes to maintenance with respect to backwards-compatibility, one must establish a policy regarding support for older hardware, old versions of software dependencies, etc. When using a library, many of these considerations are taken care of by the library developers. Intimate knowledge of hardware and software versions, compatibility between components and so forth is not needed by the developers using the library. The process of maintaining the software with respect to such issues is thus left to experts. When hand-coding a GPU-based implementation on the other hand, one must pay close attention to the releases of new software and hardware and establish a policy on how to deal with this.

The aspect of code quality is essentially just a matter of how much effort one puts into the development. It is always possible to produce production-ready code, given that enough thought is put into the design, testing, benchmarking and documentation. When providing a complete implementation without the use of external libraries, all of this must be taken care of by the developer. When utilizing a library on the other hand, a lot of this work is outsourced to the library developers. Many libraries are

made for use in production settings. By carefully evaluating and choosing a fitting library, one can achieve production-ready quality of the end-product with less work compared to a hand-coding approach.

For the code to be made publicly available as part of the FEniCS Project, it must be merged with the main codebase. For this to be feasible, the code must be clean and well designed, easy to maintain, and efficient. In this regard, the three aspects discussed above are important. They all speak in favor of choosing a library approach. Therefore, I conclude that integration of a library is the approach that will be pursued in this project.

It remains to investigate what functionality we can expect to accelerate using GPUs. In order for the library approach to be fruitful, parts of the system that are amenable to library outsourcing must be identified. In fact, the FEniCS project already makes heavy use of libraries, especially in the field of numerical linear algebra.

Crucial to the finite element method is the forming ("assembling") and solution of linear systems of equations. The current implementation of the FEniCS Project uses libraries to handle data structures and algorithms for linear algebra. The currently integrated libraries include PETSc, Trilinos, uBLAS and MTL4. Traditionally, MPI support has been present in some of these libraries, and the recent popularization of GPU computing is starting to have an impact on these libraries as well. Hence, a possible approach for integrating GPU-enhanced functionality in the FEniCS Project would be to utilize a GPU-accelerated linear algebra library in the assembling and solution of linear systems.

In conclusion, the following, more specific task is presented:

> *Investigate the currently available GPU-accelerated linear algebra libraries. Evaluate the libraries carefully. If a fitting candidate library can be found, attempt to integrate it with the FEniCS Project.*

Some further comments about functionality will be made in Chapter 8.

1.2 Outline of thesis

The first part of this thesis is devoted to relevant background topics. The basic theory of the finite element method is covered in Chapter 2. This chapter also provides a short introduction to the computational considerations of finite element assembly and a presentation of some selected example problems. A thorough introduction to the theory and practice of modern general purpose GPU programming is given in Chapter 3. This chapter also contains a review of the current generation of GPU hardware architecture. An introduction to the theory of numerical linear algebra is given in Chapter 4, along with a discussion of important considerations about the implementation of (sparse) numerical linear algebra on GPUs. Chapter 5 concludes the review of background topics by introducing the FEniCS Project.

The problem statement presented in the previous section is augmented with some further requirements in Chapter 6. This chapter also contains a review previous work in this area. A survey of the currently available GPU-accelerated linear algebra libraries is given in Chapter 7. Furthermore, it gives a detailed introduction to the PETSc library. Chapter 8 starts by giving an introduction to the layout of the linear algebra submodule of the FEniCS Project. It then goes through the new implementation, including the changes necessary to integrate the GPU-accelerated functionality of the PETSc library into the FEniCS Project. The results from this integration are discussed in Chapter 9, where the performance of the new implementation is studied. The thesis is summarized and concluded in Chapter 10, were I also outline possible paths for future work in this area.

2 The finite element method

The finite element method is a method for the numerical solution of ordinary and partial differential equations. This chapter gives an introduction to the basic concepts of the method. Several books are written on the subject, and this chapter is a summary of some standard references [4, 6].

2.1 Formulation

Using the finite element method to solve a given problem can be divided into four steps:

i. Formulate the problem in strong form.

ii. Reformulate the problem in weak form.

iii. Discretize by restricting the problem to discrete subspaces.

iv. Derive an algorithm to solve the discretized problem.

Each step is described in more detail below.

Step i.

As an example, we consider Poisson's equation. First, it is formulated in strong form. The problem reads: find $u = u(x)$ such that

$$-\Delta u = f \quad \forall\, x \in \Omega, \tag{2.1.1}$$

where u is our unknown function, f is a given source function and Ω is a domain in $\mathbb{R}^d$, for some integer d. The Laplace operator Δ equals the divergence of the gradient; i.e., $\Delta u = \nabla \cdot \nabla u$, where ∇ denotes the *del* operator. More specifically, in three dimensions, we have

$$\nabla = \hat{x}\frac{\partial}{\partial x} + \hat{y}\frac{\partial}{\partial y} + \hat{z}\frac{\partial}{\partial z}, \tag{2.1.2}$$

where $\{\hat{x}, \hat{y}, \hat{z}\}$ are the three unit vectors in a Cartesian coordinate system. The system is completed by the boundary conditions

$$u = u_0 \text{ on } \Gamma_D, \tag{2.1.3}$$

$$-\partial_n u = g \text{ on } \Gamma_N, \tag{2.1.4}$$

where $\partial_n u$ is a shorthand notation for $\nabla u \cdot n$. The boundary $\partial\Omega$ is decomposed into the Dirichlet boundary Γ_D and the Neumann boundary Γ_N. At Γ_D, u has a prescribed constant value u_0. At Γ_N, the value of the normal component of the derivative of u is determined by the function g.

Step ii.

The weak form is obtained by multiplying (2.1.1) by a so-called *test function* and integrating over the domain Ω. Using integration by parts, we get

$$\begin{aligned} -\int_\Omega \Delta u v \, \mathrm{d}x &= \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x - \int_{\partial\Omega} \partial_n u v \, \mathrm{d}x \\ &= \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x - \int_{\Gamma_D} \partial_n u v \, \mathrm{d}s - \int_{\Gamma_N} \partial_n u v \, \mathrm{d}s \\ &= \int_\Omega f v \, \mathrm{d}x. \end{aligned}$$

We let the test function vanish on Γ_D since the value of u is known there. Using the Neumann boundary condition, we formulate the *variational problem*:

$$\begin{gathered} \text{Find } u \text{ in } V \text{ such that} \\ \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x - \int_{\Gamma_N} g v \, \mathrm{d}s \quad \forall\, v \in \hat{V}. \end{gathered} \tag{2.1.5}$$

Here, V and $\hat{V}$ denote the *trial* and *test spaces*, respectively.

Step iii

Now, we restrict our problem to discrete subspaces; that is, we let V_h and $\hat{V}_h$ be finite dimensional linear subspaces of V and $\hat{V}$, respectively. Looking for a solution in the discrete subspace V_h and enforcing (2.1.5) to hold for all test functions in $\hat{V}_h$, we obtain the following discrete variational problem:

$$\begin{gathered} \text{Find } u_h \text{ in } V_h \text{ such that} \\ \int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x - \int_{\Gamma_N} g v \, \mathrm{d}s \quad \forall\, v \in \hat{V}_h. \end{gathered} \tag{2.1.6}$$

Step iv

Since V_h and $\hat{V}_h$ are linear spaces, there exist bases that span them. We let these bases be

$$\begin{aligned} V_h &= \operatorname{span}\{\phi_j\}_{j=1}^N, \\ \hat{V}_h &= \operatorname{span}\{\hat{\phi}_i\}_{i=1}^M. \end{aligned} \tag{2.1.7}$$

One characteristic of the finite element method is that we choose these basis functions such that they have *local support*. This means that they are non-zero only at a small part of the domain on which they are defined. With this property, the functions are "nearly orthogonal" [11, p. 157], because the inner product of two functions is non-zero only for functions with overlapping support. Here, we let the inner product of two functions f and g be

$$\langle f, g \rangle_\Omega = \int_\Omega f \cdot g \, \mathrm{d}x. \tag{2.1.8}$$

We see that this inner product is precisely the one used in the variational problems. How to choose what the basis functions look like over their support is dependent on the problem — different problems benefit from different choices. A common choice is to let the functions be continuous piecewise polynomials over the domain. One can for example have continuous piecewise linear or quadratic functions.

Now, we make an ansatz for u_h that it can be expressed as a linear combination of these basis functions:

$$u_h = \sum_{j=1}^{N} U_j \phi_j(x). \tag{2.1.9}$$

We next take $v = \hat{\phi}_i$ in (2.1.6) for $i = 1, 2, \ldots, M$ and insert our ansatz for u_h. This gives us

$$\sum_{j=1}^{N} U_j \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \, \mathrm{d}x = \int_\Omega f \hat{\phi}_i \, \mathrm{d}x - \int_{\Gamma_N} g \hat{\phi}_i \, \mathrm{d}s, \text{ for } i = 1, \ldots, M.$$

We see that this can be formulated as a linear system of equations with the unknowns $U = (U_1, \ldots, U_N)^T$. Hence, our problem is reduced to

$$AU = b, \tag{2.1.10}$$

$$A_{ij} = \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \, \mathrm{d}x, \tag{2.1.11}$$

$$b_i = \int_\Omega f \hat{\phi}_i \, \mathrm{d}x - \int_{\Gamma_N} g \hat{\phi}_i \, \mathrm{d}s, \tag{2.1.12}$$

where $A \in \mathbb{R}^{M,N}$, $U \in \mathbb{R}^N$ and $b \in \mathbb{R}^M$. This demonstrates that the finite element method can be viewed as a method for approximating the solution of a partial differential equation by transforming it into a linear system of equations. We note here that it is common in practice to let the trial and test spaces be equal, such that the resulting matrix A is square.

2.1.1 Abstract formulation and canonical form

In the present example, the *bilinear* and *linear forms* are

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{2.1.13}$$

$$L(v) = \int_\Omega fv \, \mathrm{d}x - \int_{\Gamma_N} gv \, \mathrm{d}s, \tag{2.1.14}$$

respectively. We may now formulate the problem in a *canonical form*:

$$\begin{gathered}\text{Find } u \text{ in } V \text{ such that} \\ a(u,v) = L(v) \quad \forall\, v \in \hat{V}.\end{gathered} \tag{2.1.15}$$

This notation is useful when conducting existence and uniqueness analysis of the problem. It will also be used extensively in the next section about finite element assembly. Combining (2.1.11), (2.1.13) and (2.1.12), (2.1.14), it follows that

$$A_{ij} = a\left(\phi_j, \hat{\phi}_i\right), \tag{2.1.16}$$

$$b_i = L\left(\hat{\phi}_i\right). \tag{2.1.17}$$

We see that the matrix depends on the problem-specific bilinear form and the bases for the chosen trial and test spaces. In this example, the operator A is a matrix, which is a rank 2 tensor. Generally, a may be a multilinear form of *arity* ρ, which makes the corresponding A a tensor of rank ρ.

Here, we also note that A is a *sparse matrix*, which means that only some of its entries are non-zero. This is due to the previously noted fact that the basis functions are chosen to have local support. To see this mathematically, we let Ω_i and Ω_j denote the parts of Ω on which $\hat{\phi}_i \neq 0$ and $\phi_j \neq 0$, respectively. By the definition of the matrix A, we see that

$$\Omega_i \cap \Omega_j = \varnothing \Rightarrow A_{ij} = 0, \tag{2.1.18}$$

where $\cap$ denotes set intersection and $\varnothing$ denotes the empty set.

2.2 Finite element assembly

We now seek to derive an algorithm for constructing the matrix A. This process is called *assembling* A, or simply "assembly". The following discussion will be restricted to bilinear forms, still using Poisson's quation as an example. A general discussion of assembly is given by Logg, Mardal, and Wells [13].

Before describing the algorithm, we must introduce the concept of computational *meshes*. A mesh is a partition of the domain Ω into *cells*. These cells can for example be triangles or quadrilaterals if the mesh is two-dimensional, or tetrahedra if the mesh

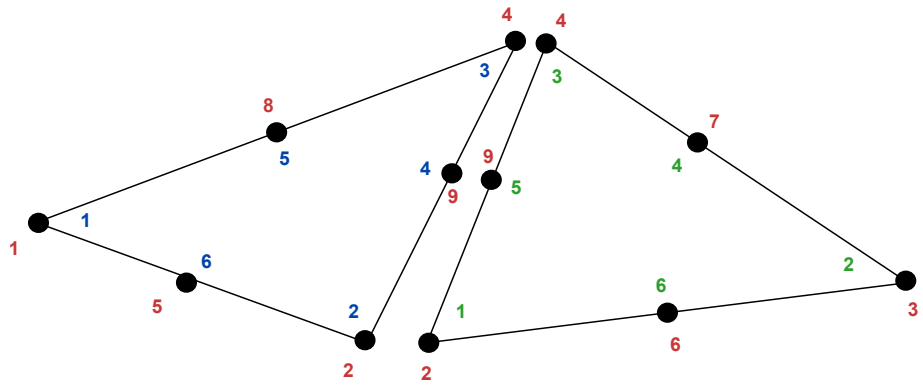


Figure 2.1: Local and global numbering of the nodes on two cells [9].

is three-dimensional. The support of a basis function will thus typically consist of several cells.

A straightforward assembly algorithm would be to iterate over all indices i and j and compute the bilinear form for every combination of test and trial basis functions. This algorithm has two major disadvantages. First, it does not take into account that A in general is a sparse tensor. Second, it does not take into account that each entry in A is a sum of contributions from the cells that form the support of the basis functions. Hence, each cell is visited several times.

An alternative algorithm is a cell-based one. Rather than iterating over indices, this algorithm iterates over the cells and computes the contribution to the global tensor A from each cell. This contribution is often called the *local cell tensor*. This algorithm represents an improvement because the number of total iterations decreases when we only visit each cell of the mesh once. The local cell tensor is inserted into the global tensor in each iteration using the so-called *local-to-global mapping*, which relates the indices of the local cell tensor to the indices of the global tensor.

2.2.1 Local-to-global mapping

We previously stated that the basis functions are chosen to have local support. Furthermore, we often let them form a *nodal basis*. Mathematically, we say that

$$\phi_i(x_j) = \delta_{ij},^1 \tag{2.2.19}$$

where δ_{ij} denotes the *Kronecker delta*, which is zero for $i \neq j$ and one for $i = j$. Here, x_j is one of the *nodes* of the mesh; these are typically points on each cell, as illustrated in Figure 2.1. Hence, the basis functions have value 1 at a single node in the mesh and vanishes at all other nodes.

1 This expression is valid only when choosing your *degrees of freedom* in a certain way. Ciarlet's definition of a finite element precisely defines the degrees of freedom [4, ch. 3, 9]. A thorough discussion of different choices for the degrees of freedom is given by Kirby and Logg [9].

There exists a local numbering of the nodes (and basis functions) on each cell of the mesh. The local-to-global mapping relates this numbering scheme to the global numbering of the nodes. We take the mesh in Figure 2.1 as an example, where we denote the left and right cells by T and T', respectively. They have 6 nodes each, numbered by an index $i = 1, \ldots, 6$. The nodes of the mesh are numbered globally by an index $g = 1, \ldots, 9$. The local-to-global mapping can be viewed as a list of integers ι, where the i-th entry in the list gives the global number g for local node number i. Hence, in this example, we have:

$$\iota_T = (1, 2, 4, 9, 8, 5),$$
$$\iota_{T'} = (2, 3, 4, 7, 9, 6).$$

The mappings for adjacent cells have to agree on common nodes, which we see that they do in this case:

$$\iota_T(2) = \iota_{T'}(1) = 2,$$
$$\iota_T(4) = \iota_{T'}(5) = 9,$$
$$\iota_T(3) = \iota_{T'}(3) = 4.$$

In general, we let ϕ_i^T denote the i-th basis function in the local numbering scheme of cell T. Using the local-to-global mapping, we obtain the following relationship:

$$\phi_{\iota_T(i)}|_T = \phi_i^T \Leftrightarrow \phi_g|_T = \phi^T_{\iota_T^{-1}(g)}. \tag{2.2.20}$$

Given this relationship, we are equipped with the necessary tools to derive the assembly algorithm.

2.2.2 The assembly algorithm

We first define $\mathcal{T}$ as the mesh (or triangulation) of the domain Ω. The derivation of the cell-based algorithm starts by assuming that the bilinear may be rewritten as

$$a(u, v) = \sum_{T \in \mathcal{T}} a_T(u, v), \tag{2.2.21}$$

where a_T is the contribution to the bilinear form a from the cell T. We also define the local element tensor,

$$A_{ij}^T = a_T(\phi_j^T, \hat{\phi}_i^T), \tag{2.2.22}$$

which is typically a dense matrix. We let $\mathcal{T}_{ij} \subset \mathcal{T}$ be the set of cells T such that

$$\phi_j(\hat{x}) \neq 0 \quad \text{and} \quad \hat{\phi}_i(\hat{x}) \neq 0$$

for at least one point $\hat{x}$ in T. Equivalently, we may use the notation from (2.1.18) and write

$$\mathcal{T}_{ij} = \{T \in \mathcal{T} : T \cap \Omega_i \cap \Omega_j \neq \varnothing\}. \tag{2.2.23}$$

We may then write the global element tensor as a sum of these local element tensors,

$$\begin{aligned}
A_{ij} &= a\left(\phi_j, \hat{\phi}_i\right) \\
&= \sum_{T \in \mathcal{T}} a_T\left(\phi_j, \hat{\phi}_i\right) && \text{By assumption (2.2.21)} \\
&= \sum_{T \in \mathcal{T}_{ij}} a_T\left(\phi_j, \hat{\phi}_i\right) && \text{By definition of } \mathcal{T}_{ij} \\
&= \sum_{T \in \mathcal{T}_{ij}} a_T\left(\phi^T_{\iota_T^{-1}(j)}, \hat{\phi}^T_{\iota_T^{-1}(i)}\right) && \text{(2.2.20)} \\
&= \sum_{T \in \mathcal{T}_{ij}} A^T_{\iota_T^{-1}(i), \iota_T^{-1}(j)}. && \text{(2.2.22)}
\end{aligned}$$

Given this expression, it is natural to let the algorithm iterate over each cell T, compute the local element tensor and add it to the global element tensor. An outline of the assembly algorithm is given in Algorithm 2.1.

Algorithm 2.1 The assembly algorithm

$A = 0$
for $T \in \mathcal{T}_h$
    (1) Compute ι_T
    (2) Compute $A_T(m \times n)$

    for $i = 1, \ldots, m$
        for $j = 1, \ldots, n$
            $A_{\iota_T(i), \iota_T(j)} += A^T_{ij}$
        end for
    end for
end for

The dimensions m and n of the local cell tensor are given by the dimension of the function space on each cell. While all global basis functions are indeed defined on each cell, only a limited number of them are non-zero there. In the present example, each cell has 6 nodes, so 6 of the basis functions have non-zero values somewhere on that cell. Hence, the local function space has dimension 6, which makes the local cell tensor a 6×6 matrix. The test and trial spaces are equal in this example, so the matrix becomes square. In general, the matrix may be rectangular.

Note here that the bilinear and linear forms may also contain terms that imply integration over *facets*, which are the boundaries between cells. In the case of two dimensional meshes, the facets are the edges of the triangles or quadrilaterals. We see this in (2.1.14), where the second term involves integration over the mesh boundary, called

exterior facets, denoted $\mathrm{d}s$. Integration over the facets that are not on the boundary, called the *interior facets*, is denoted $\mathrm{d}S$.

This concludes the discussion of the theoretical aspects of the finite element method. Now follows a presentation of a few example problems.

2.3 Example problems

A set of example problems will be used to test the implementation later on. These are divided into linear and nonlinear problems and are presented here one by one. Throughout this section, we let V_h and $\hat{V}_h$ denote the trial and test spaces, respectively. As usual, $a(u, v)$ and $L(v)$ denote the bilinear and linear forms.

2.3.1 Linear problems

Poisson's equation

The first linear problem is Poisson's equation, which has already been thoroughly introduced. We have the following forms and function spaces for this equation:

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \langle \nabla u, \nabla v \rangle_\Omega, \tag{2.3.24}$$

$$L(v) = \int_\Omega fv \, \mathrm{d}x - \int_{\Gamma_N} gv \, \mathrm{d}s = \langle f, v \rangle_\Omega - \langle g, v \rangle_{\Gamma_N}, \tag{2.3.25}$$

$$V_h = \{v \in CG_k : v|_{\Gamma_D} = u_0\}, \tag{2.3.26}$$

$$\hat{V}_h = \{v \in CG_k : v|_{\Gamma_D} = 0\}. \tag{2.3.27}$$

Here, u_0 is the Dirichlet (or essential) boundary condition, as in (2.1.3). CG_k denotes the space of continuous piecewise polynomials of degree k. We will let $k = 1, 2, 3$ in our test runs.

Poisson's equation will be solved in two and three spatial dimensions, on a unit square and unit cube, respectively. The meshes of these domains will vary in resolution, to vary the number of unknowns in the resulting linear systems. Poisson's equation results in a symmetric linear system, since $a(u, v) = a(v, u)$.

Static, linear elasticity

The second problem is the equation for static, linear elasticity, which governs the deformation of elastic media. For this problem, we have:

$$a(u,v) = \int_\Omega \sigma(u) : \epsilon(v)\ \mathrm{d}x = \langle \sigma(u), \epsilon(v) \rangle_\Omega, \tag{2.3.28}$$

$$L(v) = \int_\Omega f \cdot v\ \mathrm{d}x + \int_{\Gamma_N} g \cdot v\ \mathrm{d}s = \langle f, v \rangle_\Omega + \langle g, v \rangle_{\Gamma_N}, \tag{2.3.29}$$

$$V_h = \left\{ v \in CG_k^d : v|_{\Gamma_D} = v_D \right\}, \tag{2.3.30}$$

$$\hat{V}_h = \left\{ v \in CG_k^d : v|_{\Gamma_D} = 0 \right\}. \tag{2.3.31}$$

Here, CG_k^d is the space of continuous piecewise vector valued polynomials (of length d) of degree k. We will let $k = 1, 2, 3$ in our test runs for this problem as well. We also have the following definitions:

$$\begin{aligned}
\sigma(u) &= 2\mu \cdot \epsilon(u) + \lambda \cdot \mathrm{tr}\left(\epsilon(u)\right) I, \\
\epsilon(u) &= \frac{1}{2}\left(\nabla u + \nabla u^T\right), \\
\mathrm{tr}(A) &= \sum_{i=1}^{n} A_{ii} \quad \forall\, A \in \mathbb{R}^{n \times n}, \\
A : B &= \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{ij} \quad \forall\, A, B \in \mathbb{R}^{n \times n}.
\end{aligned}$$

Here, μ and λ are the *Lamé parameters*: these are positive constants that describe the elastic properties of the material under deformation.

The elasticity problem will be solved on the same domains as Poisson's equation. It also results in a symmetric linear system, since the bilinear form is symmetric.

DG discretization of pure advection

Discontinuous Galerkin (DG) discretizations involve discontinuous basis functions, as opposed to the continuous functions used in the previous problems. When introducing the notation used for this problem, we assume that we have two cells, denoted T^+ and T^-, that share a single facet S. Given a function v, the values v^+ and v^- denote the values of v on the facet S as seen from the cells T^+ and T^-, respectively. The *jump* of v, denoted $[\![v]\!]$, is the difference in function value across the facet S:

$$[\![v]\!] = v^+ - v^-.$$

Furthermore, we define u_* as

$$u_* = \frac{u \cdot n + |u \cdot n|}{2},$$

where n is the normal to each facet. We may then state the forms and function spaces:

$$a(\phi, v) = \langle \nabla v, -u \cdot \phi \rangle_\Omega + \langle [\![v]\!], u_*^+ \cdot \phi^+ - u_*^- \cdot \phi^- \rangle_{\mathrm{dS}} + \langle v, u_* \cdot \phi \rangle_{\Gamma_N}, \tag{2.3.32}$$

$$L(v) = \langle v, f \rangle_\Omega, \tag{2.3.33}$$

$$V_h = \left\{ v \in DG^1 : v|_{\Gamma_D} = u_0 \right\}, \tag{2.3.34}$$

$$\hat{V}_h = \left\{ v \in DG^1 : v|_{\Gamma_D} = 0 \right\}, \tag{2.3.35}$$

where $\langle \cdot, \cdot \rangle_{\mathrm{dS}}$ denotes integration over interior facets. The trial function is denoted ϕ in this problem. The function u is a prescribed velocity field. The space DG^1 is the space of discontinuous piecewise polynomials of degree 1. For more details about DG discretizations in general, and in particular applied to this problem, see the paper by Ølgaard, Logg, and Wells [30].

2.3.2 Nonlinear problems

Hyperelasticity

The only nonlinear example problem is the three-dimensional, anisotropic hyperelasticity problem. It will not be benchmarked in detail, but rather be solved once to demonstrate that it can be correctly and efficiently computed by the new implementation. Details about this specific problem are given by Usyk, LeGrice, and McCulloch [33].

3 Graphics processing units

Since the 1960s, the evolution of central processing units (CPUs) has been governed by the famous Moore's law. It states that the number of transistors possible to put on a CPU doubles approximately every two years [29].[1] Increasing the clock speed of processors has for years been a reliable way to increase the current performance of computing.

However, engineers have been forced to look elsewhere for increased performance due to physical limitations in how integrated circuits are produced. The solution has been to increase the *number* of processor cores rather than the speed of each processor core itself. Today, almost all computers in the consumer market ship with CPUs with at least two cores, and the number of cores on a chip increases steadily.

Parallel to the evolution of CPUs, the architecture of graphics processing units (GPUs) has undergone major changes. Starting out as an accelerator for 3D graphics in computer games and simulation and visualization applications, the current generation of GPUs is a hybrid one, well suited for both graphics and general purpose computations. This chapter aims to give the reader a good understanding of what GPUs are, what the architecture looks like, and why GPUs are suitable for doing general purpose computations as well as graphics rendering. First, the basic concepts of 3D graphics will be explained, to give an understanding of why the architecture looks the way it does. It will be shown why and how the GPU can be programmed for doing general computations, and the current generation of GPU hardware will be reviewed. Here, it will also be shown how the GPU programming model maps to the components of the hardware. Furthermore, some problems that map well to the GPU architecture will be discussed. Last, some theory about performance of parallel programs will be presented.

3.1 The graphics rendering pipeline

When writing an application for rendering 3D graphics, one starts with some representation of a *scene*. A scene typically consists of geometric models, lighting, a camera with a viewpoint, and a view frustum. A *frustum* is a perspective box containing the elements of the scene. The programmer also defines the *transformations* that define how these input elements should be placed in relation to each other. This input is then

1 In his 1965 paper, Moore stated that transistor counts doubled every year. He later increased this estimate to every two years.

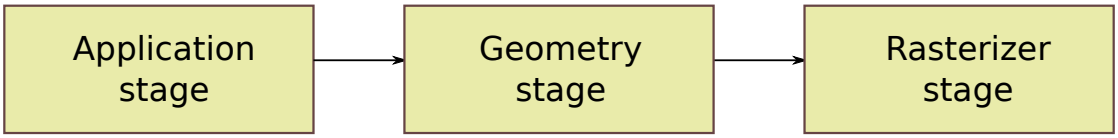


Figure 3.1: The three main stages of the graphics rendering pipeline.

processed through the *graphics rendering pipeline* [1, p. 11], which is a series of computational stages that transform the scene into a 2D image for displaying on a screen. These computational stages can be divided into three main stages, as shown in Figure 3.1. They are described in further detail in the following.

3.1.1 The application stage

The first stage of the graphics rendering pipeline is the application stage, in which the programmer defines the scene and transformations. This stage is purely a software stage, over which the programmer has full control. The output of this stage contains all information needed by the later stages to produce a complete 2D image of the scene.

The geometric models that are defined in this stage consist of *vertices* and *primitives*. A vertex is a point in 3D space, with associated attributes such as color and normal vector. Several vertices constitute a primitive, such as a line, a triangle or a quadrilateral.

The rest of the pipeline consists of some *fixed-function steps* [1, p. 26] as well as *programmable shaders*. As opposed to the programmable shaders, the fixed function steps are not programmable. This implies that the computations in those stages are hard-coded in the hardware, and cannot be affected by the programmer. The programmable stages, on the other hand, can be controlled by custom *shader programs*. The first generations of GPU architectures were purely fixed-function, without any programmer control at all. The degree of programmability has grown gradually with newer generations [1, p. 26].

3.1.2 The geometry stage

The next stage of the pipeline is the geometry stage. It performs geometric transformations of the vertices and primitives of the geometric models defined in the scene. The *vertex shader* program performs transform and lighting operations on each vertex. In the *model transform*, the spatial coordinates of the models are transformed by affine transformations. 4×4 floating point matrices are used to rotate, scale, shear and translate the coordinates. This defines the position of the models in the global coordinate system. In addition, the *view transform* defines the position of the camera in the global coordinate system.

Following the transformations, a *lighting model* is evaluated per vertex. Based on all the lighting set up in the scene and the color and the orientation of the object, the lighting model calculates the color at each vertex. The color is then linearly interpolated over the primitive.

Following the per-vertex operations, the *geometry shader* is executed per primitive. It outputs zero or more primitives, hence, extra primitives may be generated at this stage. See the book by Akenine-Möller, Haines, and Hoffman [1, p. 40-42] for several use cases for the geometry shader.

When the geometry is processed, we have a scene defined in a perspective view frustum; that is, objects further away from the camera are smaller than those closer to the camera. The projection transformation maps this sheared box into a *clip space*, which is a cubic box containing the scene. The parts of the scene that fall outside the box are clipped away, and are not processed further. The vertices that are not clipped are then transformed into the screen space, which means that they are given a pixel position on the screen, as well as a z-value.

3.1.3 The rasterizer stage

The last stage is the rasterizer stage. In this stage, a *fragment* is generated for each screen pixel that is covered by a primitive. The *fragment shader* processes each of these fragments. The shader program gives the fragments colors, or applies texture images. They may also alter the z-value assigned in the previous stage.

Each fragment is then exposed to a series of tests to determine if it should be discarded or not. These tests are programmable and include the z-test, which discards fragments that lie behind others.

After this final stage, the 2D image is ready for displaying on the screen. To update the image, a new pass through the pipeline is needed in order to generate the next frame.

3.2 General purpose GPU programming

The parallel nature of the pipeline stages is apparent: all vertices can be transformed independently of each other, the same goes for primitives. Also, all pixels can be colored independently of each other. This allows for parallel execution of vertex shaders for several vertices at once, fragment shaders for several fragments at once, and so forth. Hence, one can use several processors at each pipeline stage to process the data in parallel.

In real time rendering, high performance means rendering many high resolution images per second. In computer games, a typical frame rate can be 60 frames per second. If each frame consists of, say, 1600×1200 pixels, the GPU must process 115 million pixels per second, in addition to processing vertices, primitives, etc. for each

frame. This demands large amounts of computational power.

To handle such massive workloads, the GPU has been shaped into a *throughput oriented* architecture: an architecture made for processing large amounts of data in parallel. This feature of has attracted the attention of computational scientists seeking to improve the performance of programs performing computations other than graphics. The first attempts at this was undertaken using graphics APIs [10, p. 7], where the programmers had to "disguise" their problems as graphics problems in order to utilize the GPU.

During the last decade, specialized APIs for general purpose GPU programming have emerged, with CUDA [57] and OpenCL [71] being the most notable. In order to fully understand the potential of general purpose GPU programming, it is important to have an understanding of the programming model these APIs are based on. This includes learning about the limitations of the programming model and the hardware, and thus the considerations one must take when programming GPUs. The following sections go through the fundamental concepts and terminology of these APIs. CUDA terminology is used throughout, as this project uses CUDA-based libraries (see Section 7.4). The concepts are the same for OpenCL, but the terminology differs [10, ch. 11].

3.2.1 Kernels, threads and blocks

When programming in CUDA, you write *kernels*, which are functions that execute on the GPU. The GPU is referred to as the *device* in CUDA terminology, whereas the CPU (or rather the system in which the GPU is installed) is referred to as the *host*. The kernels are executed on the device by a grid of *thread blocks*, where each block is a grid of threads. Hence, the kernel is invoked by a set of threads, where each thread runs the kernel once. Within the kernel, one uses built-in variables such as `threadIdx`, `blockIdx` and `blockDim` to determine the position of each thread in the grid. These indices may for example be used to read input data from global memory. An overview of the thread grid can be seen in Figure 3.2.

The layout of the grid is controlled by the programmer through kernel invocation parameters. The block grid and thread blocks may be one, two or three dimensional, depending on the application at hand. For vector-based calculations, one typically lets the block grid and thread blocks be one dimensional, with the total number of threads equal to the length of the vector. This way, each thread is responsible for processing one vector element. In the case of image processing, the natural decomposition is to let one thread process one pixel (or a tile of pixels), so a two dimensional layout of the grid is practical in this case.

GPUs are made for massive amounts of parallelism. To fully utilize the available resources and maximize throughput, you need to launch thousands, or tens of thousands of threads. For a thorough discussion of how to optimize the execution configuration of CUDA programs, see Section 7.3 of the *CUDA C Best Practices Guide* [37].

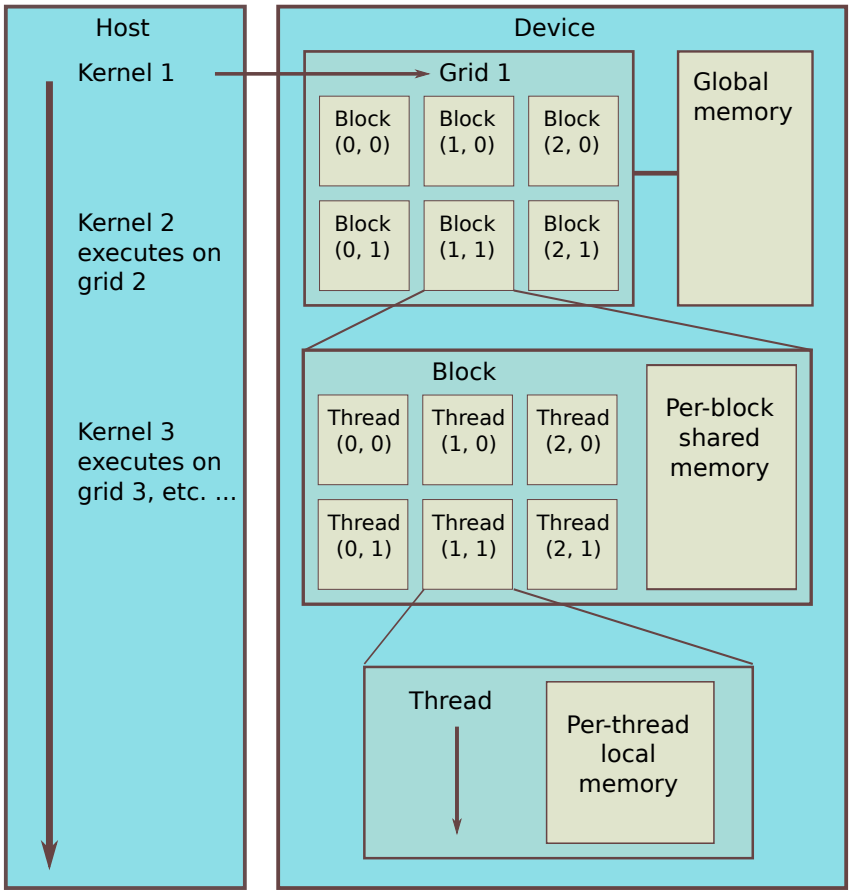


Figure 3.2: Overview of kernels, grids, blocks and threads.

3.2.2 Memories and atomics

CUDA exposes the programmer to a hierarchy of memories, which is also depicted in Figure 3.2. Starting at the top, each thread can access the global memory. The host can copy values into this memory before kernel execution, providing the threads with input data. Furthermore, all threads have access to a chunk of memory that is shared between all threads in a block. At the lowest level, each thread has its own local memory. There are also several special memories available to the programmer, with different properties, including constant memory and texture memory.

When operating on global or shared memory, threads are able to read from and write to the same memory locations. This may lead to *race conditions*. As an example, consider the incrementation operation. This is a compound operation, consisting of three steps: the value is read, modified and written back. Incrementing a value twice in serial is perfectly safe and will always lead to the same results. When doing it in parallel however, the order of operations is non-deterministic. It may happen that the order of operations becomes the same as in the serial case — if so, the result gets computed correctly — but it may also very well happen that the order becomes something like the following:

- Thread A reads the value 2.
- Thread B reads the value 2.
- Thread A increments the value to 3.
- Thread B increments the value to 3.
- Thread A writes the value 3.
- Thread B writes the value 3.

In this case, the result is incorrectly computed as 3, not 4. As the correctness of a program can not be left to chance, a way must be found to avoid these race conditions.

To this end, *atomic operations* can be used. Atomic operations lock the values in question during the entire operation, so that no other threads can access the value before the current thread is finished updating the value. This ensures thread safety and correctness of the program. It is important to note that atomics are costly, since they may lead to threads waiting in queue to perform their tasks. It is best to avoid their usage altogether, but some applications can not be implemented without them. One such application is the insertion of the local element matrix into the global element matrix, as mentioned in Section 2.2.2. In a parallel implementation of the assembly algorithm, several local matrices will be computed and inserted concurrently. Since values from different local matrices may have to be inserted at the same position in the global matrix, atomic operations must be used to ensure correctness of the result.

3.2.3 Thread execution

Threads execute in groups of 32 threads called *warps*. All threads in a warp execute the same instructions simultaneously, an execution style called *single-instruction, multiple-thread* (SIMT) [10, p. 98]. Warps can be ignored for functional correctness, but many of their properties must be taken into account to achieve maximum performance. Two important concepts in this regard are *execution divergence* and *memory divergence* [36, p. 3].

Execution divergence occurs when threads in a warp take different execution paths. The SIMT execution style performs best when all threads in a warp run the same code. If threads in a warp take different paths in a conditional code branch, the hardware will serialize these instructions. One pass will be used to execute the instructions for the threads that take the first code path, and another pass will be used to execute the instruction for the threads that take the other code path [10, p. 98]. If your code has conditional branching, it is important to try to formulate the if-tests and loop conditions such that warps do not diverge.

As an example, assume we have a problem with a one dimensional thread grid with a single thread block. The following if-test checks if the current thread has ID lower than a given limit N:

CUDA code

```
if (threadIdx.x < N) {
    // Do something ...
} else {
    // Do something else ...
}
```

Unless N is a multiple of the warp size, this test will lead to warp divergence. To avoid this issue, the if-test must be reformulated in terms of the warp size. The following if-test will never lead to warp divergence for any value of N:

CUDA code

```
if (threadIdx.x/32 < N) {
    // Do something ...
} else {
    // Do something else ...
}
```

These code snippets are obviously not equivalent, so changes to the code logic are needed elsewhere as well in order for the program to perform correctly after the if-test modification. It could therefore be claimed that it is best to include such considerations in the design process from the start in order to ease the process of writing efficient CUDA code.

Memory divergence has to do with memory bandwidth utilization when reading from global memory. Reads are issued per warp, which means that the requested

values by all 32 threads in a warp are read in a single operation. These reads depend on the *load granularity* of the global memory, which can for example be 128 bytes. This means that each request results in at least a line of 128 bytes being read from memory, even though fewer bytes are requested. In this context, a "line" means a series of consecutive values in memory.

As an example, assume we have a kernel in which each thread attempts to read a 4-byte value. A warp of 32 threads will then read 128 bytes in total. The actual amount of data loaded depends on how these 4-byte values are arranged in memory:

Aligned, consecutive
: If the values lie after each other in memory, they are called *aligned*. We also differentiate between the cases when consecutive threads read consecutive values, and the cases where they do not. Consecutive reads means that thread i reads memory location j, thread $i + 1$ reads memory location $j + 1$ and so on. Since a request for aligned, consecutive values falls within a contiguous 128 byte line in memory, this exact line is loaded, leading to a bus utilization of 100 percent.

Aligned, permuted
: A permuted access pattern has no performance penalty over the consecutive case, as long as the values are aligned. In the aligned, permuted case, no threads read the same value, but all values in a continuous 128 byte line are read. Hence exactly 128 bytes are read with 100 percent bus utilization.

Misaligned
: If the requests are misaligned, for example with a strided access pattern where only every other value is read, the values will fall within two 128 byte lines. Hence 256 bytes are fetched even though only 128 are requested, which gives a bus utilization of only 50 percent.

Overlapping
: If every thread requests the same value, only four bytes are wanted, but 128 bytes are fetched. This leads to a bus utilization of a mere 3.125 percent. This low percentage is also achieved with a strided access pattern with large strides of for example 128 bytes. This will lead to 32 lines of 128 bytes being fetched, with only one relevant value in each line.

Performing aligned memory accesses is crucial to performance. This is often referred to as *coalesced access*. For a detailed analysis of memory access requirements on the most recent architecture, see the talk by Micikevicius [42]. It is important to note that the coalescing requirements only apply to global memory. When reading from e. g. shared memory, coalescing is not relevant. Algorithms that exhibit irregular memory access patterns may therefore benefit from shared memory usage. By first using coalesced reads to load the relevant values into shared memory, the threads can thereafter access them in an irregular pattern without any performance penalty.

3.2.4 Compute to global memory access ratio

The global memory has high access latencies and limited bandwidth [10, p. 77]. Performing many global memory loads in a kernel may thus lead to thread congestion in the memory access paths, again leading to idle threads. To avoid this issue, the programmer must be aware of the *compute to global memory access ratio* [10, p. 78] of the kernel.

Let us consider a GPU with a global memory access bandwidth of N bytes/s. Given an algorithm that loads 4-byte single precision data from global memory and performs one floating point operation per datum, the algorithm will be limited to $N/4$ floating point operations per second ("flops"[2]). This can be a serious performance limitation. This is due to the fact that the rate of data transfer of a GPU (in terms of the number of bytes loaded from global memory per second) is much lower than its floating point computing rate (in terms of the number of floating point operations performed per second).

Hence, in order to fully utilize the computational resources of a GPU, an application must have a high compute to global memory access ratio; that is, it must perform several floating point operations per datum loaded from global memory. Applications with low such ratio are called *memory bandwidth limited*, since the memory bus is saturated long before the flop rate has approached its limit. If, on the other hand, the ratio is too high, the flop rate will peak while the memory bus is working below its potential. Such applications are called *compute limited*. Since a moderately high ratio is usually necessary to obtain good performance, a skilled CUDA programmer must know how to limit the amount of global memory accesses performed in a kernel. A variety of techniques to do so are presented in Chapters 8 and 9 of the book by Kirk and Hwu [10].

An example of the ratio between memory bandwidth and peak flop rate in the most recent compute GPUs is the NVIDIA Tesla C2075. It has a global memory bandwidth of 144 GB/s and a single precision peak performance of 1030 gigaflops [47]. Its double precision peak performance is 515 gigaflops. Given an example application where all global memory fetches are perfectly coalesced such that the memory bus utilization is 100 percent, the optimal compute to global memory access ratio for this processor is $28 : 1$. Note that this is a purely theoretical example, as a bus utilization of 100 percent is nearly impossible to achieve in real applications.

3.3 GPU hardware architecture

Knowledge of the GPU programming model is probably the most important prerequisite in order to understand how GPUs can best be utilized. However, it is also

2 Floating point operations per second is commonly abbreviated as FLOPS, flops, flop/s or flops/s. To be consistent, we will use "flops" or "flop rate" throughout this text.

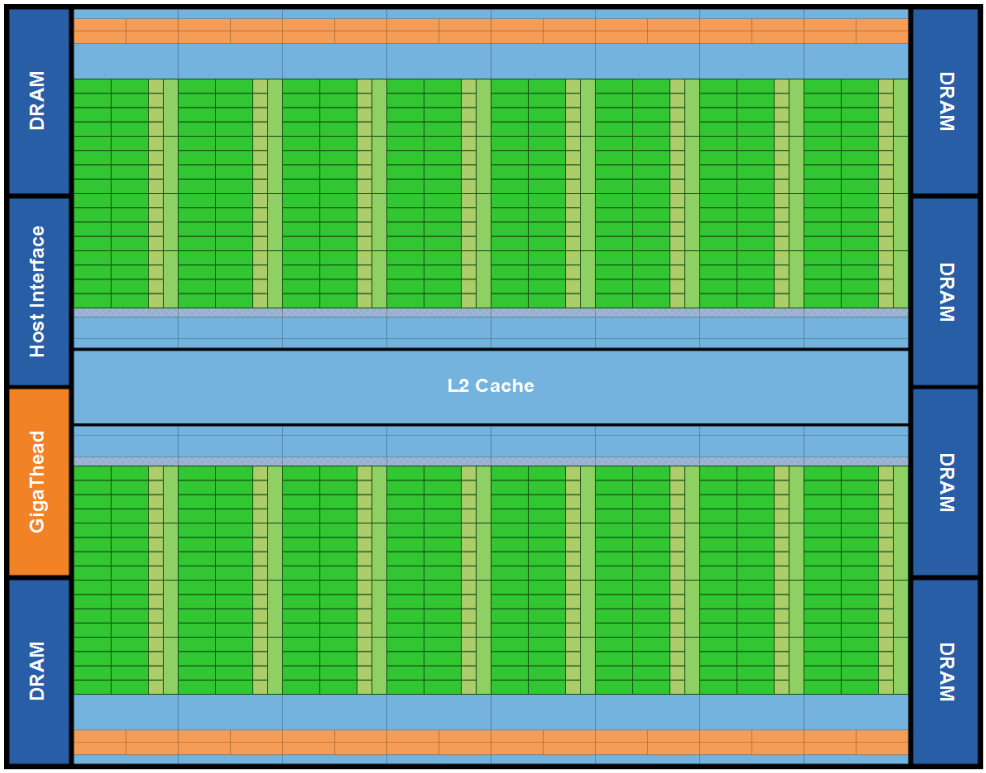


Figure 3.3: Overview of the Fermi architecture [46].

advantageous to know the basic properties of the hardware in order to understand how the programming model maps to the hardware, and thus how programs are executed on the GPU. This section will therefore go through the basic properties of the current generation of GPU hardware architecture.

In 2006, NVIDIA released the GeForce GTX 8800, the first GPU based on the G80 architecture. The G80 was the first GPU architecture that was specifically designed with both graphics and general purpose computations in mind. While chips of the previous generations had distinct processors for the different stages of the graphics pipeline [26, p. 40], the G80 came with a unified processor array. The vertex, geometry and fragment shaders were then executed on the same type of processor. This made it easier to balance workloads and achieve good processor utilization for graphics rendering, where the number of vertices and sizes of primitives vary from application to application.

The Fermi architecture is the latest generation[3] of unified architectures by NVIDIA, following the G80 and GT200. The performance benchmarks in this project uses Fermi

3 Note that at the time of printing, NVIDIA has just released the first GPU featuring the Kepler architecture, the successor to Fermi.

GPUs. This section sums up the properties of the architecture, as covered in the *Fermi Architecture white paper* [46]. It is important to keep in mind that GPU architecture is evolving at a very fast pace. The quantitative details of this chapter will definitely be outdated very soon. The qualitative aspects however, are likely to remain fairly constant, as they have remained through generations of GPU architectures.

3.3.1 Streaming multiprocessor

Figure 3.3 on the previous page shows the layout of an example GPU featuring the Fermi architecture. The green section of the figure depicts the processor array. Each of the green blocks of cells represents a *streaming multiprocessor* (SM, see Figure 3.4 on the following page), of which there are 16 in total in this case. When a CUDA program is run, the SMs execute one or more thread blocks each. Unless one uses global memory atomics or some other form of synchronization or communication across blocks, the thread blocks can be executed completely independently of each other. Thus, the result will be the same regardless of the order of execution. Hence, depending on the available resources, the thread blocks can be scheduled for execution serially on one multiprocessor, partially in parallel over several multiprocessors, or completely in parallel on one multiprocessor each. This allows applications to scale freely from GPUs with few SMs to GPUs with several SMs. This is referred to as transparent scaling [10, 26, 44].

As seen in Figure 3.4, the multiprocessors are built up of several components:

Streaming processor

Each streaming multiprocessor contains 32 *streaming processors* (SPs, see Figure 3.5 on the next page), sometimes referred to as *CUDA cores*. The SP is the core computation unit of a Fermi GPU. It contains one *integer arithmetic logic unit* (ALU) and one *floating point unit* (FPU) for doing arithmetic operations. The SP can compute with both single and double precision floating point numbers, but the double precision performance comes at a $2\times$ performance penalty compared to single precision.[4]

When performing the operation $x \times y + z$ with floating point numbers, GPUs from NVIDIA may perform them as a *fused multiply-add* (FMA) operation [51]. This operation performs only a single rounding of the end result; that is, $\mathrm{rn}(x \times y + z)$, where rn denotes the rounding operation. Without FMA, a separate rounding step is performed for both the multiplication and the addition; that is, $\mathrm{rn}(\mathrm{rn}(x \times y) + z)$. Hence, the FMA yields higher accuracy since it only performs one rounding step. There is no support for hardware FMA on CPUs [51],

4 The $2\times$ penalty is valid for GPUs in the Tesla series, which are compute GPUs targeted at the HPC (high performance computing) market segment. The GeForce series, which are mainly targeted at the consumer/gaming segment, have an $8\times$ performance penalty for double precision operations, as confirmed by an NVIDIA employee in this forum post: `http://forums.nvidia.com/index.php?showtopic=165055`.

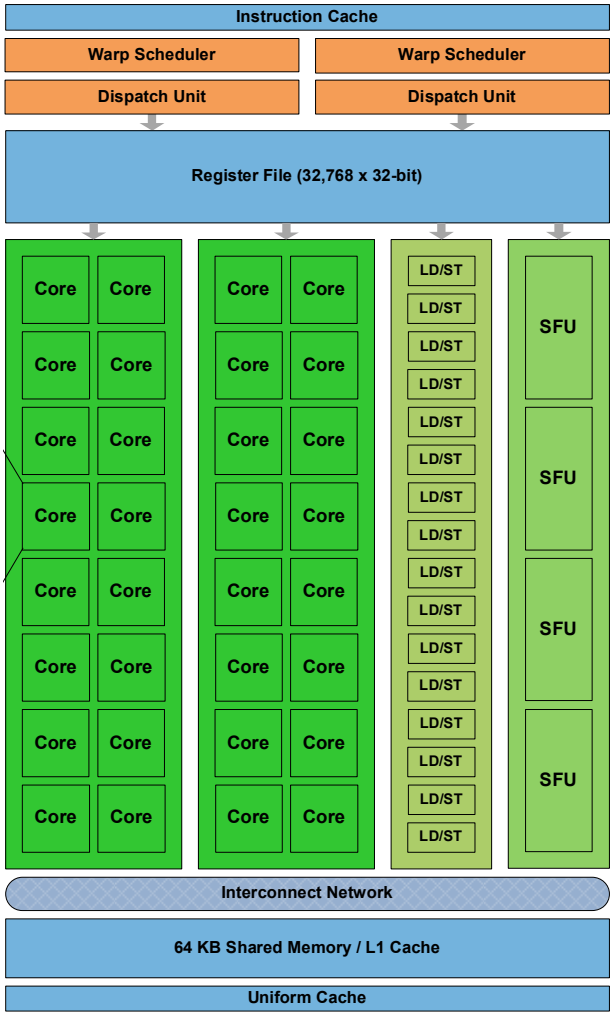


Figure 3.4: Layout of a Fermi streaming multiprocessor [46].

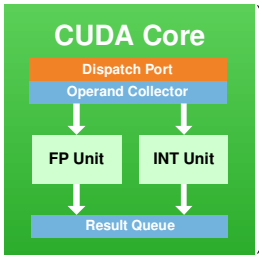


Figure 3.5: Layout of a Fermi streaming processor [46].

which may lead to differences in the results computed on the CPU and the GPU. This should be kept in mind when comparing the results from equivalent CPU and GPU implementations of the same algorithm.

Special function units
The streaming multiprocessor contains four *special function units* (SFU) for computing transcendental functions such the trigonometric, exponential and logarithmic functions.

On-chip memory
The streaming multiprocessor has 64 KB of on-chip shared memory, a fourfold increase over previous architectures. This can be configured in two ways; either as 48 KB of shared memory and 16 KB of L1 cache, or 16 KB shared memory with 48 KB L1 cache. The cache-heavy option may yield extra performance when developing new CUDA-based algorithms that are not yet optimized with regard to memory usage. On the other hand, choosing the configuration with more shared memory will have performance benefits for existing applications that make use of shared memory.

The Fermi architecture contains several other special features, such as dual warp scheduling for concurrent execution of two warps at the same time on a single SM. For more details about the architecture, the reader is referred to the *Fermi Architecture white paper* [46] and the *Fermi website* [62].

3.4 Data parallel problems

When a problem is labeled as "suitable" for solving on the GPU, it is meant that when implementing a GPU-based algorithm to solve the problem, it is feasible to get significantly improved performance over CPU-based implementations. Problems that are suitable for computing on the GPU are, in short, ones that consume large amounts of floating point computing power. These are typically applications that process large amounts of data, perform many iterations on the data, or both [10, p. 193]. Lindholm et al. [26] provides an extended list of properties that define GPU-friendly problems, which they refer to as *throughput applications*:

High degree of data parallelism
Throughput applications perform many computations on the data elements, and the elements can be processed completely independently of each other.

Low degree of task parallelism
There is a small amount of independent tasks to be performed, and groups of threads can cooperate to perform these tasks.

Intensive floating point arithmetic
Throughput applications perform lots of floating point operations.

Latency tolerance
Throughput applications tolerate the long latency of global memory fetches. The important thing is the amount of work that is completed in a given time.

Streaming data flow
Throughput applications possess a large amount of data to be processed. High memory bandwidth is required in order to transfer this data fast. There is relatively little reuse of this data.

Modest inter-thread synchronization and communication
The less inter-thread communication and synchronization an application performs, the more parallel it is. This allows independent execution of threads and blocks, and transparent scaling, as mentioned previously.

A problem possessing these characteristics is likely to be successfully implemented on the GPU. That being said, a problem can show good speedup on the GPU even if it does not fulfill all of the criteria. The following two sections go through a few problems, not related to the finite element method, that fit the GPU architecture well. Considerations regarding finite element computations on the GPU, specifically linear algebra computations, will be discussed in Chapters 4 and 6.

3.4.1 Digital image processing

Some algorithms in digital image processing are inherently data parallel. One such example is *spatial filtering* and *convolution*. In convolution, the value of a pixel in the output image is calculated by combining the value of the pixel in the input image with some of its neighboring pixels. A simple example is smoothing of an image using a uniform 3×3 filter. Here, the output pixel value is just the average of its original value and its eight surrounding pixels. The filtered image will be blurred, or smoothed. Obviously, all pixels can be processed in parallel. In fact, such problems are called *embarrassingly parallel* [16, p. 98], since there is no dependence or need for communication between threads.

It is easy to outline a CUDA-based algorithm to solve this problem. A naive approach would be to store the image in global memory, let one thread be responsible for one pixel and perform a straightforward filtering of the image. However, this wouldn't lead to very good performance because of the frequent uncoalesced memory accesses and the repeated loads of the same values by neighboring threads. A better solution would be to divide the image into tiles that are as large as the thread blocks, plus one row/column of padded values on each side to hold neighboring values. Assigning each tile to a thread block, the threads in each block could cooperate

to perform coalesced loads of the image tiles into shared memory. Since the shared memory does not have any restrictions on access pattern to obtain good performance, we could proceed from here with the naive algorithm.

For a detailed reference on image processing, we refer to the book by Gonzales and Woods [7]. Several CUDA code examples with convolution can be found in the *GPU Computing SDK* [66].

3.4.2 Molecular dynamics

In certain applications related to molecular dynamics, one sometimes have to work with something called *electrostatic potential maps*. An algorithm for computing such maps, called *direct Coulomb summation*, is highly data parallel and a very good candidate for GPU implementation. This section will briefly introduce the algorithm and discuss its parallelism. For more details about molecular dynamics in general and this algorithm in particular, we refer to the paper by Stone et al. [32]. Based on this paper, Chapter 9 of the book by Kirk and Hwu [10] gives a friendly introduction to this topic.

Direct Coulomb summation is an algorithm for calculating the electrostatic potential value of a point in a three dimensional grid. The grid surrounds a system of atoms, and all atoms contribute to the electrostatic potential of all grid points. The contribution of an atom i to a grid point j is simply the charge of the atom divided by the distance from the atom to the grid point. Given a grid with M points and a system with N atoms, a straightforward iteration to calculate the electrostatic potential maps has complexity $O(MN)$. Even with the simple formula to evaluate per atom, this algorithm is computationally demanding for large grids and systems.

Since the grid points do not affect each other, the potential at each point can be calculated completely in parallel. The same goes for the atoms; they do not affect each other, so each atom's contribution to all grid points can be calculated in parallel. To choose which part of a problem to parallelize is a common consideration when designing GPU algorithms. In this case, however, the atom-centric approach has a significant downside. Computing the contributions from all atoms in parallel will have many threads write to the same grid point at once, leading to data races. This entails using atomic operations, which are costly. Having instead the threads cooperating to calculate the potential at each grid point, we achieve good parallelism without data collisions.

3.5 Measuring the performance of parallel programs

When using parallelization techniques to speed up a program, one needs some way of measuring the resulting performance. This section provides a brief introduction to performance analysis of parallel programs. Although the present discussion is about parallel programs, the formulas presented here may also be applied to programs that

are improved by means of optimizations techniques other than parallelization.

3.5.1 Speedup

When measuring the performance of a parallel algorithm, one usually compares its performance to an equivalent serial algorithm. The *speedup* is a factor that says how many times faster the parallel algorithm is compared to the serial one. Given the serial run time, T_s, and the parallel run time, T_p, the speedup is defined as

$$S = \frac{T_s}{T_p}. \tag{3.5.1}$$

The speedup depends on many factors, including the quality of the serial and parallel implementations, and the hardware on which they are run. In the context of GPU programming, the relative computing power of the CPU and GPU will have a large impact on the speedup. The speedup obtained using a low-performance CPU and a high-performance GPU will be much larger than if the CPU was high-performing.

As such, speedup figures for GPU algorithms are specific to one hardware setup. Speedup figures for a GPU algorithm bears little meaning without clearly stating what hardware they are obtained with. A more objective metric to measure is the previously mentioned *flops*, the number of floating point operations per second. The amount of flops that it is possible to obtain is also dependent on the hardware. However, the flop rate does not depend on the relative performance of the CPU to the GPU, as the speedup does.

3.5.2 Amdahl's law

In parallel programming, one is often faced with problems that contain inherently serial parts. This means that it is only possible to program a portion of the problem in parallel. This may for example be the case in a program that reads two matrices from file and multiplies them. The matrix multiplication itself may be run in parallel, but it may not be possible to read the input files in parallel. Given that this takes a non-negligible amount of time, the total speedup of the program will be less than the speedup of the parallel portion.

To calculate the overall speedup that it is possible to obtain for such problems, we first assume that the serial algorithm has a run time of 1 for some unit of time. Denoting the parallelizable part of the problem by P, the run time of the serial part will be $(1 - P)$. Assuming that the parallelizable portion can be sped up by a factor S_p, the total run time of the program becomes $(1 - P) + P/S_p$. Dividing the original run time by the new run time, we obtain the following expression for the overall speedup, known as *Amdahl's law*:

$$S_{\text{total}} = \frac{1}{(1 - P) + P/S_p} \tag{3.5.2}$$

In the limit as S_p approaches infinity, we obtain the maximum possible overall speedup. As an example, consider the case there 25 percent of the computation is inherently serial. The portion P is 0.75 in this case, and we get

$$\lim_{S_p \to \infty} \frac{1}{0.25 + 0.75/S_p} = 4.$$

Thus, the maximum possible overall speedup is 4 in this case.

4 Numerical linear algebra

We have seen that using the finite element method involves solving a linear system of equations. Since we are interested in computational methods for solving problems with the finite element method, it is relevant to review the theory for numerical solution of linear systems. The field of numerical linear algebra studies algorithms for performing linear algebra computations. In particular, it seeks to find efficient algorithms to solve linear systems. This chapter provides an introduction to this theory, and sums up the current state of research into numerical linear algebra on GPUs.

4.1 Theory

This section provides only a very brief introduction to the vast theory of numerical linear algebra. For more details, the lecture notes by Lyche [39] is a good introduction.

The fundamental problem in linear algebra is a linear system of equations,

$$Ax = b. \tag{4.1.1}$$

Here, A is an $m \times n$ matrix, x is a vector of length n and b is a vector of length m. Though A is rectangular in general, the systems arising from discretization with the finite element method are most often square, as mentioned in Section 2.1. Techniques for solving this fundamental problem include *direct methods*, such as factorization, and *iterative methods*. Two examples of such methods, LU factorization and Krylov subspace methods, are described in the next two sections.

4.1.1 LU factorization

An LU factorization is a factorization $A = LU$ where $A, L, U \in \mathbb{R}^{n,n}$, L is lower triangular and U is upper triangular. In addition, L has to be unit triangular; that is, it has to have ones on the diagonal.

As an example, we take a simple 2×2 matrix,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ l_1 & 1 \end{bmatrix} \begin{bmatrix} u_1 & u_3 \\ 0 & u_2 \end{bmatrix} = \begin{bmatrix} r_1 & u_3 \\ l_1 u_1 & l_1 u_3 + u_2 \end{bmatrix}. \tag{4.1.2}$$

Equating the elements and solving the equations, we get

$$L = \begin{bmatrix} 1 & 0 \\ \frac{c}{a} & 1 \end{bmatrix},$$
$$U = \begin{bmatrix} a & b \\ 0 & \frac{ad-bc}{a} \end{bmatrix}.$$

The point of finding the LU factorization of A is that the system (4.1.1) may be rewritten as two simpler systems:

$$Ax = LUx = b \Rightarrow \begin{cases} Ly = b \\ Ux = y \end{cases}. \tag{4.1.3}$$

These two systems may now be solved directly by forward and backward substitution.

There exist many different algorithms for computing the LU factorization of a matrix. Details may be found in any book on direct methods, for instance the book by Davis [5].

4.1.2 Krylov subspace methods

Krylov subspace methods (or simply "Krylov methods") are iterative methods, which means they generate a sequence of approximations $\{x_n\}$ to the solution x_* of the system. Iterative methods in general have several advantages over direct methods. First, using direct methods for large systems may consume too much memory and computational power to be feasible in practice [17, preface]. In addition, iterative methods are easier to implement in parallel. Before we introduce the methods themselves, we cite the definition of a Krylov space:

Definition 4.1. *Given a nonsingular $A \in \mathbb{C}^{n,n}$ and $y \neq 0 \in \mathbb{C}^n$, the nth Krylov subspace $\mathcal{K}_n(A, y)$ generated by A and y is*

$$\mathcal{K}_n := \mathcal{K}_n(A, y) = \text{span}(y, Ay, \ldots, A^{n-1}y). \tag{4.1.4}$$

It does not seem as if mathematicians have agreed upon a common definition of a Krylov method. Several definitions exist, some more detailed and exact than others. To give an impression of what approach these methods take, two such definitions will be cited here. The first is taken from an article by Ipsen and Meyer [22]:

Definition 4.2. *A Krylov method is a method that solves the system $Ax = b$ by repeatedly performing matrix-vector multiplications involving A.*

This definition is not very detailed, neither does it say anything about the Krylov subspaces. A more in-depth definition is due to Gutknecht [38]:

Definition 4.3. *A Krylov method for solving the system $Ax = b$ is an iterative method starting from some initial approximation x_0 and the corresponding residual $r_0 := b - Ax_0$ and generating for all, or at least most n, until it possibly finds the exact solution, iterates x_n such that*

$$x_n - x_0 = q_{n-1}(A)r_0 \in \mathcal{K}_n(A, r_0) \tag{4.1.5}$$

with a polynomial q_{n-1} of exact degree $n-1$. For some n, x_n may not exist or q_{n-1} may have lower degree.

The exceptions mentioned, and the vague expressions "for all, or at least most n", are due to some special cases that may occur using certain Krylov methods on certain types of problems. Gutknecht also states the following lemma related to the residuals r_k:

Lemma 4.1. *The residuals r_n of a Krylov method satisfy*

$$r_n = p_n(A)r_0 \in r_0 + A\mathcal{K}_n(A, r_0) \subseteq \mathcal{K}_{n+1}(A, r_0), \tag{4.1.6}$$

where p_n is a polynomial of degree n, called nth residual polynomial, which is related to the polynomial q_{n-1} of (4.1.5) *by*

$$p_n(\zeta) = 1 - \zeta q_{n-1}(\zeta) \tag{4.1.7}$$

In particular, it satisfies the consistency condition $p_n(0) = 1$.

The main idea behind Krylov methods is to generate a sequence of approximate solutions $\{x_n\}$ that lie in the affine space $x_0 + \mathcal{K}_n(A, r_0)$. We want these solutions to be such that the corresponding residuals $r_n \in \mathcal{K}_{n+1}(A, r_0)$ converge to 0. In fact, if the residuals are linearly independent, the residual will equal 0 after a finite number of iterations [38, p. 4].

The Conjugate Gradient Method

Several different Krylov methods exist, with two of the most well known being the Conjugate Gradient method [21] and the Generalized Minimal Residual method [31]. The former has the limitation that it can only be applied to systems where A is a symmetric positive definite matrix, whereas the latter is developed to be applicable to nonsymmetric systems. The Conjugate Gradient algorithm is listed in Algorithm 4.2. It takes as input the matrix A, vector b and an initial guess to the solution, x. In addition, it takes a tolerance ϵ and the maximum number of iterations to perform, $i_{\max}$. The iteration stops when the maximum number of iterations have been performed, or when $||r_i|| \leqslant \epsilon \, ||r_0||$. Further details about the Conjugate Gradient method are omitted here. A very thorough and intuitive introduction to the method is given by Shewchuk [49].

Algorithm 4.2 The Conjugate Gradient method

$i = 0$
$r = b - Ax$
$p = r$
$\delta = r^T r$
$\delta_0 = \delta$
while $i < i_{\max}$ and $\delta > \epsilon^2 \delta_0$
    $t = Ap$
    $\alpha = \frac{\delta}{p^T t}$
    $x = x + \alpha p$
    $r = r - \alpha t$
    $\delta_{\text{old}} = \delta$
    $\delta = r^T r$
    $\beta = \frac{\delta}{\delta_{\text{old}}}$
    $p = r + \beta p$
    $i = i + 1$

Preconditioning

In practice, Krylov methods often converge slowly [38, p. 5]. To avoid this, they are almost always used with *preconditioning*. This means replacing the system $Ax = b$ with

$$P^{-1}Ax = P^{-1}b,$$

where we let $\hat{A} = P^{-1}A$, $\hat{b} = P^{-1}b$, and solving the system $\hat{A}x = \hat{b}$ instead. This type of preconditioning is called left preconditioning, but there also exist other types of preconditioning where the matrix P is applied from the right, or as a split preconditioner consisting of two matrices. Ideally, the matrix P should be a good approximation to A [11, p. 770] such that P^{-1} is an approximate inverse of A; i.e., $P^{-1}A \approx I$.

A simple form of preconditioning, which will be used later in this project, is *diagonal* preconditioning, often called Jacobi preconditioning. With this scheme, the preconditioning matrix P is chosen as the diagonal of A; that is, $P = \text{diag}(A)$.

More details on Krylov methods and preconditioning can be found in the book by Saad [17], a standard reference on iterative methods for sparse linear systems.

4.2 Basic linear algebra subprograms

The solution of linear systems is the kind of computation that is of most interest in the context of the finite element method. However, in relation to numerical linear algebra, it is also relevant to discuss more basic linear algebra computations. Algorithms

for computing e.g. matrix-matrix products and matrix-vector products are essential building blocks in more complex algorithms, and efficient implementations of these are thus of high importance.

The de facto standard API for libraries that implement such basic computations is called *Basic Linear Algebra Subprograms (BLAS)* [53]. This interface provides specifications for a large set of functions for basic linear algebra computations. All functions are given abbreviated names based on the computation they perform and the data type of the values. For instance, the GEMV routine (general matrix vector multiply) exists in the following four versions:

SGEMV For single precision real numbers.

DGEMV For double precision real numbers.

CGEMV For single precision complex numbers.

ZGEMV For double precision complex numbers.

The name xGEMV can be used to refer to the general routine, regardless of data type. The BLAS interface is divided into three levels, which are defined as follows:

Level 1
The first level consists of functions for performing vector computations, as well as computations whose result is a scalar, such as dot products and vector norms. One example of a level 1 BLAS function is *xAXPY*,

$$y = \alpha x + y,$$

where α is a scalar and x and y are vectors of equal length. The name AXPY is an abbreviation of "alpha x plus y".

Level 2
The functions in level 2 BLAS are concerned with matrix-vector operations, such as the *xSYMV* function,

$$y = \alpha Ax + \beta y,$$

for scalars α and β, a symmetric $n \times n$ matrix A and vectors x and y of length n.

Level 3
Level 3 BLAS routines perform matrix-matrix operations. One such routine is the *xGEMM* (general matrix multiply) function,

$$C = \alpha AB + \beta C,$$

for scalars α and β, matrices $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ and $C \in \mathbb{R}^{m \times n}$.

Optimized BLAS implementations exist for several different hardware architectures [53, FAQ, sec. 5]. In particular, NVIDIA provides an implementation for their GPUs called cuBLAS [56].

4.3 Sparse linear algebra on GPUs

Sparse linear algebra computations are involved in many fields within computational science. High performance of such computations can thus be critical to overall performance. Some characteristics of sparse linear algebra makes it a challenge to implement effectively on the GPU, and it is as such not an obvious candidate for GPU implementation. Regardless, its importance in computational science have made programmers perform extensive research into effectively utilizing the GPU for sparse linear algebra.

This section seeks to give an overview of where the parallelism lies in linear algebra and what makes it challenging to exploit this parallelism on the GPU for sparse problems. To this end, a comparison with *dense* linear algebra is useful.

4.3.1 Data parallelism

The data parallel nature of many linear algebra computations is obvious. We list three kernels and identify where the parallelism lies:

Vector inner product
: The inner (or dot) product of two vectors lies at the heart of many linear algebra problems. Given two vectors of equal length, the inner product is computed by multiplying the elements of the vectors pairwise, then summing all the products to one scalar value. The first part of this computation is inherently parallel and straightforward to implement, e.g. using one CUDA thread per pair of elements. The reduction step is also relatively easy to implement in parallel, though somewhat harder to optimize. See Section 5.3.1 of the book by Sanders and Kandrot [18] for a thorough discussion of vector inner product in CUDA.

Matrix-matrix multiplication
: When computing the product of two matrices, every output item can be computed independently of each other. A result matrix with dimensions $M \times N$ can be computed by $M \times N$ independent dot products of rows and columns in the input matrices. Using CUDA, this could for example be implemented using one thread per output value. With this choice of decomposition, all dot products are computed concurrently, but each dot product itself is computed serially by one thread. Another choice could be to use several threads per output value, and compute each dot product itself in parallel.

Matrix-vector multiplication
: Computing the matrix-vector product is basically just a special case of matrix-matrix multiplication. The parallelism is the same: every output element can be computed in parallel, using e.g. one thread per row of the matrix.

In theory, all matrix-matrix multiplications could be computed using the same algorithm, regardless of the characteristics of the matrix. The same goes for matrix-vector

multiplications and other problems involving matrices. However, to achieve decent performance, algorithms must take the structure and properties of the matrix into account. Specifically, most algorithms for performing matrix computations are designed either for *dense* or *sparse* matrices.

4.3.2 Dense linear algebra

Dense matrices have few zero values and lack the *sparsity patterns* that sparse matrices have. Hence, dense matrices can be stored explicitly in a full matrix storage format, for example as two-dimensional arrays of values. Such data structures provides direct random access, and the values can be stored in contiguous memory locations. Algorithms exhibit a regular memory access pattern when iterating such structures, which means they map well to the GPU [36]. This is the case since it is easier to perform coalesced reads and utilize the memory efficiently with regular than irregular memory access patterns. Dense linear algebra algorithms on the GPU are therefore often compute limited [36].

4.3.3 Computational characteristics of sparse linear algebra

As mentioned, one could in principle implement sparse matrix computations exactly the same way as one does dense computations. This would be a huge waste of memory though, as storing a sparse matrix in a dense format implies storing a lot of zero values explicitly. In practice, one always uses a sparse matrix storage format, such as *compressed sparse row* (CSR), sometimes referred to as compressed row storage (CRS). Given the following matrix,

$$A = \begin{bmatrix} 0 & 4 & 0 & 1 & 0 \\ 7 & 0 & 0 & 0 & 3 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 2 \\ 6 & 5 & 0 & 0 & 0 \end{bmatrix},$$

the CSR representation is

$$\begin{aligned} \texttt{data} &= [4, 1, 7, 3, 8, 9, 2, 6, 5], \\ \texttt{indices} &= [1, 3, 0, 4, 2, 1, 4, 0, 1], \\ \texttt{ptr} &= [0, 2, 4, 5, 7, 9]. \end{aligned}$$

These arrays contain the following information:

- `data` contains all the values of A in row-major order; that is, they are stored from left to right, top to bottom. The length of `data`, which is the number of nonzero values of the matrix, is often abbreviated *nnz*.

- `indices` also has length *nnz*, and contains the column index for each value of `data`. Hence, if `data[k]` $= a_{ij}$, then `indices[k]` $= j$.
- `ptr` contains the offsets into `data` and `indices` for each row. For instance, the values of the fourth row (with row index 3) start at index 5 in `data`, therefore, `ptr[3] = 5`. For an $M \times N$ matrix, `ptr` has length $M + 1$, with *nnz* as the last entry.

Note that the values in `data` are sorted with respect to rows, but the values corresponding to a given row may not be sorted with respect to columns. Also note that feature of the CSR format is that the difference of two adjacent entries in `ptr` equals the number of nonzeros of a particular row. For instance, there is one nonzero in row three, hence `ptr[3]-ptr[2] = 1`.

In this example, the CSR representation stores 24 numbers while the dense representation stores 25. While this storage saving is not very impressive, the storage savings are significant for large, sparse matrices. In general, a dense representation of an $M \times N$ matrix will store MN values, while the CSR representation will store $2 \times nnz + M + 1$. Hence, for CSR to be more storage efficient than a dense format, the following requirement must be met:

$$2 \times nnz + M + 1 < MN \Rightarrow nnz < \frac{M(N-1)-1}{2}. \tag{4.3.8}$$

The average of this limit for matrices of varying dimensions is around half the total number of elements. In our example, this limit is 9.5. Since A has 9 nonzeros, we are just below the limit, which explains the storage saving of a single value.

We see that the CSR format does not provide any means of direct access to an element of A. When retrieving element a_{ij}, the `ptr` array must first be checked to find the parts of `indices` and `data` that correspond to row i. Then these parts must be searched to find the column index j and the value a_{ij}. Accessing matrix values in this way may pose a challenge when implementing certain algorithms on the GPU, since it may be difficult to derive general algorithms that perform coalesced access and achieve good performance for arbitrary matrices.

4.3.4 Summary of research on sparse linear algebra on GPUs

As stated in Definition 4.2, matrix-vector multiplication is repeatedly applied in Krylov methods. Sparse matrix-vector multiplication, often abbreviated *SpMV*, is hence of importance in sparse problems. Bell and Garland [36] point out that SpMV represents the dominant computational cost in many iterative methods. Because of its significant role, there has been conducted a lot of research into implementing SpMV effectively on GPUs.

Bell and Garland [36] have conducted an extensive study of SpMV on the GPU. They explore different sparse matrix storage formats and discuss their properties and

possible optimizations. For the CSR format, they evaluate a parallel granularity of one thread per row of the matrix. Even though each thread would then access contiguous indices and values in memory, they would be accessed sequentially, resulting in uncoalesced access. They note that instead using a 32-thread warp per row ensures coalesced access, but will lead to idle threads when there are fewer than 32 nonzeros per row. They also propose a hybrid format, which is simply a combination of two other formats. In their benchmarks, this hybrid format is performance-wise among the best of the evaluated formats. For further details about storage formats and their performance, we refer to their paper [36]. A general note is that the performance of different formats vary with the structure and the sparsity pattern of the matrix.

Baskaran and Bordawekar [35] further explore optimizations of an SpMV kernel based on the CSR format. To facilitate coalesced access, they suggest a zero padding scheme such that the number of values stored in the `data` array for each row is a multiple of 16.[1] Since the values in `data` corresponding to a given row may not be aligned, they suggest an improved access scheme to avoid accessing the whole row in a non-coalesced manner. Since elements of the multiplied vector are reused across rows, they implement caching of this vector. The performance of caching in texture memory is compared against caching in shared memory. The texture caching option comes out as the most efficient. Their implementation outperforms the CSR-based algorithms presented by Bell and Garland for a series of test matrices. When compared to the hybrid format however, their approach is less efficient in some cases. They conclude that their overall performance is on par or better than the performance demonstrated by Bell and Garland.

Buatois, Caumon, and Lévy [20] have implemented a preconditioned conjugate gradient solver on the GPU. They use a blocked version of the CSR format, which gives fewer memory fetches per SpMV than the regular CSR format. This shortens the length of the `ptr` and `indices` arrays, but also implies storing some unneeded (i.e., zero) values.

Markall [40] provides an extensive discussion of SpMV on GPUs. He identifies the performance issues and lists several performance optimizations, and evaluates existing GPU-based SpMV implementations. As such, his report [40] provides a good reference for further study of the research into this field.

1 Memory operations in previous GPU architecture generations were performed per half-warp, i.e., a group of 16 threads.

5 The FEniCS Project

When solving real-life problems with the finite element method, one often uses some kind of finite element software. The FEniCS Project [12, 76] is one such software project, which uses novel programming techniques to ensure both user-friendliness and computational efficiency. This chapter will go through the components of the FEniCS Project. Emphasis will be put on the underlying design of the project and how it is used, rather than implementation details.

5.1 Overview

In the introduction to the finite element method, we saw that it involves several mathematical aspects, including the differentiation and integration of basis functions and the solution of linear systems. The FEniCS Project seeks to automate all these aspects, leaving only a minimal amount of work to the user. In order to do this, the project is comprised of several components that take care of the different tasks involved in automating the finite element method.

The structure of the FEniCS Project is outlined in Figure 5.1 on the following page. The figure indicates the layout of the different components and how they interact with each other. DOLFIN is a software library that acts as the main user interface to the FEniCS Project. When using the FEniCS Project, the user writes an application that imports and uses DOLFIN. The other components are in turn imported by DOLFIN, but may also be used directly by the user. The most central among the other components are UFL, FFC and UFC. FFC relies on several different backends, including FIAT, Instant and FErari. FIAT is a backend for the evaluation of basis functions, Instant is a just-in-time compiler, and FErari is an optimizing backend. FErari is optional and not needed for using DOLFIN, while the former two are essential parts of the toolchain. The Viper module is a stand alone plotting utility that is imported alongside DOLFIN. Viper allows plotting of DOLFIN functions, meshes, finite elements and more.

Several external libraries are used to perform different tasks in the system. These include SCOTCH [72]; a mesh partitioning library, CGAL [55]; the Computational Geometry Algorithms Library and MPI [69]; the Message Passing Interface for parallel computing. In particular, the user may choose among different linear algebra backends for constructing vectors and matrices and solving linear systems. Among these are PETSc [52], MTL4 [70], Trilinos [79] and uBLAS [80].

The FEniCS Project heavily relies on the concept of *code generation*. This means that

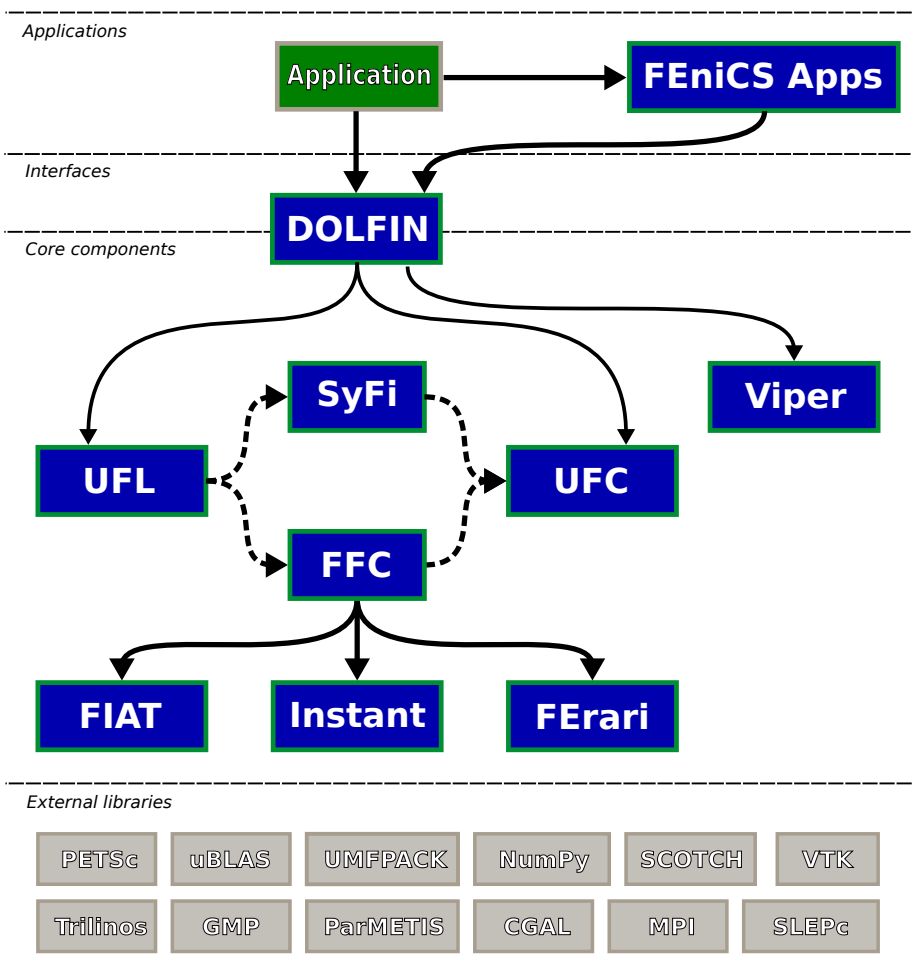


Figure 5.1: Overview of the structure of the FEniCS Project [14].

a program generates code for another program. This code is compiled and used in the next stages of the program execution. Details on the code generation can be found in Section 5.4.

5.2 The finite element library

DOLFIN [27, 14] is a C++/Python library that is written in C++, with a Python interface semi-automatically generated by SWIG [75]; in addition, some parts of the Python interface is hand-written. Below is a simple DOLFIN program for solving Poisson's equation, given in Eq. (2.1.5), on the unit square using piecewise linear functions. This program is written using DOLFIN's Python interface, as is all other DOLFIN examples in this text. A hardcopy of the output of the `plot`-command can be seen in Figure 5.2.

Python code

```
from dolfin import *

# Define mesh, function space, trial and test functions
mesh = UnitSquare(40,40)
V = FunctionSpace(mesh, 'Lagrange', 1)
u = TrialFunction(V)
v = TestFunction(V)

# Define the Dirichlet boundary, x = 0 or x = 1
def boundary(x):
    return near(x[0], 0.0) or near(x[0], 1.0)

# Define Dirichlet boundary value
u0 = Constant(0.0)
bc = DirichletBC(V, u0, boundary)

# Define source function and Neumann boundary value
f = Expression('10*exp(x[0] + x[1])')
g = Expression('cos(x[0] + x[1])')

# Define the bilinear and linear forms
a = inner(grad(u), grad(v))*dx
L = f*v*dx - g*v*ds

# Compute and visualize solution
u = Function(V)
solve(a == L, u, bc)
plot(u, interactive=True)
```

We see that when defining a problem in DOLFIN, the syntax lies close to the mathematical notation used in Chapter 2. This code will not be dissected in detail. Rather, it will serve as a reference when describing the components of the FEniCS Project that are invoked in the different stages of the program.

When comparing the above code to the forms and function spaces for Poisson's equation that were presented in Section 2.3, we see that there is a difference with regard to the function spaces. Only a single function space is defined in the code (using the DOLFIN `FunctionSpace` class), whereas the mathematical definition had one test function space and one trial function space. This difference is due to the fact that in DOLFIN, Dirichlet conditions are enforced on the linear algebra level. They are hence not part of the definition of a DOLFIN `FunctionSpace`. Since the only difference between the test and trial spaces for Poisson's equation were the boundary conditions, a single `FunctionSpace` instance can be used in the code.

5.2.1 General design

As mentioned, some parts of DOLFIN relies on code generation. Specifically, code generated by FFC is used when assembling the linear system of equations. The assembly

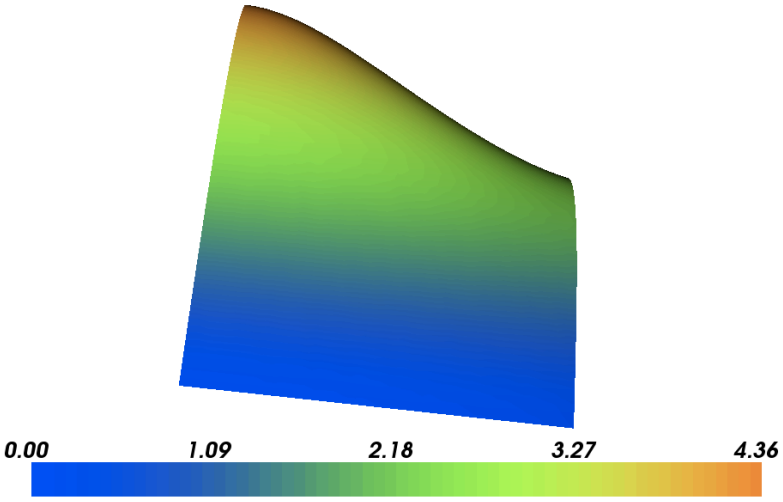


Figure 5.2: Plot of the solution of Poisson's equation.

algorithm, as described in Algorithm 2.1, is implemented generically in DOLFIN. The inner loop of this algorithm depends on the problem, and is handled by the generated code.

In the previous code example, the assembly was performed behind the scenes. Explicit assembly of the system can be performed in DOLFIN by the following code:

Python code

```
# Define bilinear and linear forms
a, L = ...
# Define a list of boundary conditions
bcs = [bc1, bc2, ...]
# Assemble LHS and RHS and apply boundary conditions
A = assemble(a)
b = assemble(L)
[bc.apply(A, b) for bc in bcs]
```

The parts of DOLFIN that relies on code generation are isolated from the rest of the system. The other parts are implemented as reusable library components. Among these are computational meshes, representations of functions, finite elements, function spaces and variational forms, as well as handling of input and output. In addition, there are algorithms and data structures for linear algebra.

DOLFIN partitions user input into two subsets [27, p. 6]: the input that must be handled by specialized (i.e., generated) code, and input that can be handled by the general purpose library functions. The first set of data is compiled into C++ code by FFC, and the second set is used as input to this code. For a thorough explanation of

the design of DOLFIN, see the paper by Logg and Wells [27].

5.2.2 Linear algebra

DOLFIN relies on one of several backends to perform linear algebra computations. These backends are integrated through a generic, unified interface. Thus, the application code is often identical regardless of which backend is used. This interface implies that wrappers have to be written for each backend. These wrappers will necessarily only expose the functionality defined in the interface. Therefore, the user may access the underlying wrapped objects to operate directly on them if needed [27, p. 8].

The interface implementation is based on C++ polymorphism. The `GenericTensor` class defines the top-level superclass for all tensors of arbitrary rank. This allows for the implementation of a generic assembly algorithm that can assemble an element tensor of any rank. Furthermore, the classes `GenericMatrix` and `GenericVector` are subclasses of `GenericTensor` . They act as superclasses for the backend-specific implementations of matrices and vectors.

When solving linear systems, the user has fine-grained control over the process. One may choose between different linear solvers and preconditioners and adjust their parameters to fit individual needs. In the example program for solving Poisson's equation, the call to the `solve` function did not specify anything about how to solve the system, so the default linear solver and preconditioner was used. One may also choose to use a specific solver and preconditioner by passing the names of these as optional arguments:

Python code

```
# Solve using Conjugate gradients with Jacobi preconditioner
solve(A, x, b, 'cg', 'jacobi')
```

It is also possible to create an instance of the `KrylovSolver` class. This makes it possible to adjust various solver parameters, such as the tolerance for the iteration and the initial guess for the solution vector:

Python code

```
# Create Generalized minimal residual method solver with
# default preconditioner and adjust its parameters
solver = KrylovSolver('gmres')
solver.parameters.nonzero_initial_guess = True
solver.parameters.relative_tolerance = 1e-12
solver.solve(A, x, b)
```

If one wants to switch to a different linear algebra backend, this can be done by changing a parameter in DOLFIN's global parameter set:

Python code

```
# Choose a different linear algebra backend
# When switching backends, the system must be reassembled
parameters.linear_algebra_backend = 'Epetra'
A, b = assemble(a), assemble(L)
solve(A, x, b)
```

5.3 The finite element form language

The Unified Form Language (UFL) [2] is a domain specific language in which a problem must be formulated in order to be solved by DOLFIN. It provides a rich language for defining finite elements and finite element forms. The example program from Section 5.2 used UFL functionality to define the problem. More precisely, it used UFL integrated with the Python interface of DOLFIN. The problem was defined as:

UFL code

```
a = inner(grad(u), grad(v))*dx
L = f*v*dx - g*v*ds
```

Further details about representation of expressions, computation of derivatives and other aspects of UFL are given by Alnæs [2].

5.4 The finite element form compiler

The assembly algorithm, though generic in nature, has a highly problem specific inner loop. It involves evaluating the local-to-global mapping ι_T and the local element tensor A^T. Hence, if one were to write a finite element program from scratch, one would need to write code for computing both ι_T and A^T. This process is automated by the FEniCS Form Compiler (FFC) [24, 15].

When executing a FEniCS application that contains a description of a bilinear form, FFC compiles this expression into C++ code for evaluating ι_T and A^T. More specifically, one specifies multilinear forms in UFL which are passed as input to FFC, which in turn outputs C++ code that conforms to the UFC interface (see Section 5.5). This is illustrated in Figure 5.3. When writing a FEniCS application in C++, this compilation of the forms must be done explicitly. The forms are defined in a UFL-file that is compiled using the `ffc` command line compiler. The resulting header file is then

Figure 5.3: FFC generates UFC compliant code from a problem specification in UFL [15].

included in the C++ program. When using Python, this process is automated and happens behind the scenes.

Although using the finite element method to solve a PDE usually only involves linear and bilinear forms, FFC is designed to accept multilinear forms of any *arity* ρ. This is covered in depth in the paper on FFC by Kirby and Logg [24]. To illustrate multilinear forms, we let $\{V_i\}_{i=1}^r$ be a set of discrete function spaces defined on a triangulation $\mathcal{T}$ of Ω [24, p. 423]. Then, a is a multilinear form,

$$a : V_1 \times V_2 \times \ldots \times V_r \to \mathbb{R}. \tag{5.4.1}$$

We let $\{\phi_i^1\}_{i=1}^{M_1}, \{\phi_i^2\}_{i=1}^{M_2}, \ldots, \{\phi_i^r\}_{i=1}^{M_r}$ be bases for $V_1, V_2, \ldots, V_r$. We also define a multi-index $i = (i_1, i_2, \ldots, i_r)$. The element tensor A of rank r is then defined by

$$A_i = a(\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r). \tag{5.4.2}$$

In the previous examples, we considered a bilinear form a, which makes A a matrix, while in the case of a linear form, A is a vector.

As we have seen, the input to FFC is limited to multilinear forms. Domain specific knowledge may hence be exploited in order to generate efficient code [24, p. 420].

5.5 Other components

Among the other components of the FEniCS Project, two of the most important are UFC and FIAT. The UFC (Unified Form-assembly Code) [3, 19] interface specifies the structure and layout of code generated by FFC to be used by DOLFIN. That is, UFC provides an interface between problem specific (problem specified in UFL, compiled by FFC) and general purpose (DOLFIN) parts of the program.

The Finite Element Automatic Tabulator (FIAT) [23, 8] is a module for evaluation of finite element basis functions. It is used a basis function backend by FFC, and is only rarely used directly by the user.

Details about the remaining components may be found in *Automated Solution of Differential Equations by the Finite Element Method* [12].

6 Project requirements and related work

The previous chapters have given a thorough introduction to the background theory that lays the foundation for this project. This chapter will review some previous related work towards using GPUs for finite element computations in relation to the FEniCS Project. However, we will first discuss some further requirements of the implementation that is provided as part of this project.

6.1 Requirements of the implementation

In Chapter 1, we saw how the library approach is advantageous with respect to backend- and development-specific issues. When it comes to the frontend of the system; i.e., the parts of the system that the users interact with, there are no specific advantages of choosing a library approach over hand-coding. The choice of approach mainly affects the backend, and the developer remains in full control over the frontend. When introducing new functionality in a system, it must be presented to the user through a clear and efficient user-interface, regardless of how the functionality is implemented behind the scenes. It must also be straightforward to configure and build the software with the new features enabled. The following two requirements are set for these parts of the implementation:

- The user interface should be left as simple as possible. The user should be able to activate and deactivate GPU acceleration with easy-to-use options, and the general usage of the software should be the same whether or not one chooses to use GPUs. This philosophy of simplicity is central throughout the FEniCS Project, and is important to retain in this implementation.
- The distribution of the software should be as streamlined as possible. Due to the nature of how some of the dependencies are distributed, the process cannot be fully automated at this time. However, the goal is to achieve as much automation as possible.

The problem statement presented in Chapter 1 specified the functionality to focus on in this project. Here, a single requirement is set for the functionality provided through the implementation: *completeness*. This means that the parts of DOLFIN that are GPU-accelerated should support all existing DOLFIN features without any particular limitations. For instance, all forms that can be assembled by the existing implementation

must be possible to assemble by the new one, and so forth. The implementation must also be usable from both the C++ and Python interfaces of DOLFIN.

The only limitations accepted are those that are hardware related. For instance, the lower amount of memory on the GPU may result in the GPU-based implementation being unable to solve as large problems as the host, since the latter has more memory.

Having defined the requirements of a GPU-accelerated implementation of the FEniCS Project, we will review some previous related work. There are three previous projects that have worked with GPU computing in some relation to the FEniCS Project. Although they have had different approaches and goals than this project, it is important to review their results to see their relevance to this project.

6.2 Generating CUDA code from UFL forms

Markall, Ham, and Kelly [28] have presented a prototype source-to-source compiler for finite element computations on the GPU. Their compiler transforms variational forms specified in UFL into CUDA code for assembling the linear system of equations on the GPU. The linear system is also solved on the GPU using a CUDA-implementation of the Conjugate Gradient method. Their work is directed towards integrating UFL with the Fluidity [63] software project. The work presented in their paper is based on the master's thesis by Markall [41].

In the FEniCS Project, UFL is used in conjunction with FFC to ensure both generality and efficiency of finite element programs. The rationale for integrating UFL with Fluidity is similar; they want to be able to produce code for different target hardware architectures without changing the high-level description of the problem. An underlying design requirement of this compiler was that it should have platform independence built in, to ease modification to support other architectures. They achieved this by splitting the compiler into a frontend and a backend. The frontend generates an intermediate representation of the problem, which the backend parses to generate code for a specific architecture. To support other architectures, you would thus only have to write a new backend for the compiler.

In his master's thesis, Markall points out that the insertion of the local matrices into the global matrix is the main performance bottleneck during assembly on the GPU. This has two reasons:

- When assembling in parallel on the GPU, several local matrices are inserted concurrently. It may happen that several matrices have values that are to be inserted at the same index. Hence, several threads may try to add to the same location in the global matrix at once, leading to data races. To avoid this problem, costly atomic operations have to be used.
- Recall the characteristics of the CSR matrix storage format discussed in Section 4.3.3: when adding to the global matrix, the sparsity pattern of the matrix must

be searched, which leads to uncoalesced memory accesses. This, in turn, leads to bad utilization of the memory bandwidth.

Markall, Ham, and Kelly propose an alternative approach to avoid these issues. Since they use a Conjugate Gradient (CG) solver to solve the linear system, they do not need the global assembly of the matrix A at all. This is due to the fact that in the Conjugate Gradient method, listed in Algorithm 4.2, A is only needed for the computation of the matrix-vector product $y = Av$. They provide an alternative method for computing this product, called the *Local Matrix Approach* (LMA):

$$y = \left(\mathcal{A}^T \left(A^e(\mathcal{A}v)\right)\right).^1 \tag{6.2.1}$$

Here, A^e is a block diagonal matrix where the i-th block is the i-th local element matrix. The matrix $\mathcal{A}$ represents the local-to-global mapping. This formula computes the matrix vector product $y = Av$ without having an explicit representation of A. The right-hand-side vector b, however, must be explicitly assembled since it is required by the CG algorithm. They propose to assemble it by a similar approach, using a sparse matrix vector product instead of incremental insertion. All details about the Local Matrix Approach can be found in their paper [28].

The LMA avoids global assembly altogether, but introduces a matrix-vector product that is 2.5 times more costly than the standard one involving an assembled matrix [28, p. 1819]. Hence, there is a tradeoff between the decrease in assembly time and increase in matrix-vector product computation time. It can then be assumed that a problem that converges fast using the CG algorithm will benefit from this approach. This is because fewer CG iterations leads to fewer matrix-vector products. If, however, many CG iterations are needed, traditional assembly (which they refer to as the *Addto* algorithm) might be favorable.

Their benchmarks show that the LMA coupled with matrix-vector product based assembly of b performs substantially better than the classic assembly algorithm on the GPU for a given discretization of the advection-diffusion equation.

6.3 Finite element integration in CUDA using FFC

Knepley and Terrel [25] build on the work by Markall, Ham, and Kelly to perform finite element integration on GPUs. They base their work on UFL and FFC, and provide CUDA kernels to perform the computation of the local element tensors.

Similarly to the compiler discussed in the previous section, FFC produces an intermediate representation of the problem when processing a form specified in UFL. Knepley and Terrel use this intermediate representation as input to their CUDA kernels. Hence, they are able to provide generic kernels that accept "any" equation as

1 Note here that $A = \mathcal{A}^T A^e \mathcal{A}$, but it is normally not computed by performing two matrix-matrix multiplications. Rather, the assembly algorithm described in Section 2.2 is used.

input, using the form specification and translation tools already provided as part of the FEniCS Project.

Their work is only concerned with efficient computation of local element tensors, and does not cover global matrix assembly and the solution of the linear system. A stated goal of this work is to later include it in PETSc and the FEniCS Project.

6.4 A CUDA-based implementation of DOLFIN and FFC

In his master's thesis, Rathgeber [48] also builds on the work by Markall, Ham, and Kelly, to provide a complete GPU-based implementation of DOLFIN and FFC. He extends FFC to generate CUDA code from UFL forms, and extends DOLFIN with GPU counterparts of many of the existing classes, including meshes, function spaces, forms, matrices and vectors. This enables him to implement assembly and solve entirely on the GPU, with all necessary data residing in GPU memory throughout the whole computation. The generated code is not fully UFC compliant, and is therefore used in a specialized assembly algorithm. In this algorithm, the loop over cells is replaced by calls to CUDA kernels that operate on all cells in parallel [48, p. 29].

Rathgeber implements both the Addto algorithm and the Local Matrix Approach on both the GPU and the CPU and benchmarks these against each other. He identifies the number of Krylov solver iterations needed in order for the Addto algorithm to perform better than the LMA for several problems. He notes that insertion of the local matrix is very costly for higher order problems, due to the larger size of the local matrix. Thus, he concludes that the matrix-free approach is particularly beneficial for such problems.

In conclusion, he measures a total speedup between 1.5 and 3 for the whole process of assembly and solve for some different test problems. These speedups are measured for the whole problem solving process, including data transfer to and from the GPU.

This GPU-based implementation of DOLFIN represents an extension of the standard implementation. Compared to the set of functionality in the standard version of DOLFIN, the GPU-enhanced parts in this implementation has some limitations. The most significant of these are the following:

- Only a Conjugate Gradient solver is implemented. Hence, the implementation is limited to solving symmetric, positive definite systems.
- Dirichlet boundary conditions are not supported.
- There is apparently no support for DOLFIN's Python interface. The users need to write their forms in `.ufl`-files, compile them by invoking FFC on the command line, then compiling the resulting `.cu`-file with `nvcc`, the NVIDIA CUDA compiler. Finally, the program must be compiled and linked with the DOLFIN library.

6.5 Conclusion of review

The scope and focus of the related work varies. Common to all is the use of UFL as a means for providing a high-level specification of the problem, and that the problem is compiled to CUDA code using a form compiler. Hence, they have all worked towards computing the local element tensors on the GPU.

Knepley and Terrel have efficient computation of the local element tensors as their sole focus. Markall, Ham, and Kelly also investigate efficient alternatives to global assembly, and use a GPU-based Conjugate Gradient solver to solve the resulting system. Their work is oriented around the Fluidity software library. Rathgeber provides a nearly complete GPU-based implementation of DOLFIN, subject to some limitations, using the same methods as Markall, Ham, and Kelly.

However, these implementations are prototypical in nature, and does not provide functionality that can be easily leveraged into the FEniCS Project, especially not in the form of general purpose library functions. As such, this current project can not build directly upon their results. However, there may be projects in the future that attempt to integrate GPU computing in the FEniCS Project via a different approach. In that case, the results of the work reviewed here may be of definite interest.

7 Survey of GPU-accelerated libraries for linear algebra

The task of this project is to integrate a GPU-accelerated linear algebra library with the FEniCS Project. There exist several libraries for performing linear algebra computations on graphics processors. This chapter aims to evaluate the current status of these libraries and discuss their suitability for integration with the FEniCS Project. Note that this chapter is based on the status of the libraries as of January 2012, so the state of these libraries may have changed since the time of writing.

7.1 Requirements and properties

For a candidate library to be at all suitable for integration, it must fulfill certain criteria, which are set as follows:

- It must be distributed in the form of a library that can be used from other application code; that is, it must not be a stand-alone application.
- It must be distributed under a free/open license.
- It must be written for use with C++ applications.
- It must be made for sparse linear algebra computations.

There are several libraries on the market today that fulfill some or all of the above requirements. The libraries that fulfill all requirements, as well as a couple of libraries with a proprietary license, will be surveyed further. The libraries with proprietary licenses are effectively ruled out as candidates from the start, but it is interesting to take a further look at them to compare them to the free alternatives. For each surveyed library, each of the following properties will be considered:

- Functionality; that is, which features the library provides.
- Performance; that is, the performance reported by previous studies of the library.
- To what extent it is actively maintained and supported.
- Whether it is based on CUDA or OpenCL (or both).
- How beneficial it is to integrate with the FEniCS Project; that is,

Library	Interface to the GPU	Under active development	License
CULA	CUDA	Yes	Proprietary
Cusp	CUDA	Yes	Apache 2.0[1]
SpeedIT	CUDA	Yes	Proprietary/GPL
cuSPARSE	CUDA	Yes	Open[2]
ViennaCL	OpenCL	Yes	MIT[3]
PETSc	CUDA	Yes	Open[4]

1 `http://www.apache.org/licenses/LICENSE-2.0`
2 Redistributable under the terms of the CUDA Toolkit End-user License Agreement.
3 `http://www.opensource.org/licenses/MIT`
4 `http://www.mcs.anl.gov/petsc/documentation/copyright.html`

Table 7.1: Some properties of the surveyed libraries.

- Does the library provide any functionality of particular interest?
- Does choosing this library imply any complications to the DOLFIN codebase? In other words, how "clean" can the implementation be expected to be?

Some of the libraries will be deemed unfit for integration based on for instance licensing or lack of functionality. All of the above properties will not necessarily be discussed in depth for these libraries.

7.2 The libraries

Table 7.1 sums up some of the properties of the libraries that were found. More information about each of the libraries follows below.

CULA

CULA [59] is developed by EM Photonics in partnership with NVIDIA. It provides C, C++ and Fortran libraries with a large interface of linear algebra functionality. This includes algorithms for solving linear systems, least squares methods, singular value decomposition, and more.

CULA comes in two variants, for dense and sparse linear algebra, respectively. Both are licensed on a one-year subscription basis, with a discount for academic use. The dense variant also comes in a free version, with limited functionality. All versions are distributed as pre-compiled, closed source libraries. According to figures from their

web site, CULA gives a speedup of 9x-14x for various Krylov solvers on an NVIDIA C2070 vs. Intel Xeon X5560.

Although well performing and professionally developed and supported, CULA does not provide a free version with sufficient functionality for integration with the FEniCS Project, and is thus ruled out as candidate.

Cusp

Cusp [54] is a library for sparse linear algebra and graph computations. It provides several storage formats for sparse matrices, as well as several iterative solvers and preconditioners. It is developed by Nathan Bell and Michael Garland, and is based on their research into numerical linear algebra on GPUs, presented in Section 4.3.4. The programming interface of Cusp is user friendly and well documented.

Cusp includes a test for solving Poisson's equation in 2D discretized using finite differences on an $m \times n$ grid. Running this test for $m = n = 1000$ gives a speedup factor of 10 on a machine with an Intel Xeon 3.33 GHz CPU and GeForce GTX 580 GPU.

All this makes Cusp a good candidate for integration with the FEniCS Project. Integrating it would imply writing a complete Cusp wrapper layer in DOLFIN, as well as patching DOLFIN's build system to invoke the CUDA compiler, `nvcc`.

SpeedIT

SpeedIT [74] is a library similar to CULA, but with more limited functionality. It provides two iterative solvers and one sparse matrix storage format. SpeedIT is licensed in three different versions. The free version has limited functionality, and is licensed under the GNU GPL [65]. It includes a preconditioned Conjugate Gradient solver, as well as what they call "standard matrix-vector multiplication" (opposed to the "accelerated sparse matrix-vector multiplication" included in the paid versions). The free version only supports single precision floating point arithmetic.

SpeedIT has shown speedup factors of 2-14 when solving sparse linear systems. Details about these benchmarks can be found in *SpeedIT Extreme Library Programmers Guide* [50]. The developers have also performed benchmarks against Cusp and cuSPARSE specifically [73], where they time sparse matrix vector multiplication for several sparse matrices. These benchmarks place the three libraries in the same ballpark performance-wise, with all three alternating between performing best.

Regardless of the performance figures, the proprietary license makes SpeedIT unfit for integration with the FEniCS Project.

cuSPARSE

The NVIDIA CUDA Sparse Matrix library (cuSPARSE) [60] is NVIDIA's own library for sparse linear algebra. It is included in the CUDA Toolkit [58]. It provides three sparse matrix storage formats and an LU solver, but no iterative solvers. As iterative solvers are an integral part of DOLFIN, cuSPARSE cannot be successfully integrated with the FEniCS Project at this time.

ViennaCL

ViennaCL [81] is an open-source library for numerical linear algebra on the GPU. Contrary to the other libraries surveyed here, it utilizes OpenCL [71] as the interface to the GPU rather than CUDA. In addition to providing implementations of all BLAS levels, it provides both direct and iterative methods. It is equipped with a clean and well-documented user-interface.

Since ViennaCL is written in OpenCL, it can be run on CPUs as well as GPUs. The authors of the library have measured the number of double precision conjugate gradient solver iterations run per second on both CPUs and GPUs. They show speedup figures of $2-3\times$ for ViennaCL running on a CPU and an NVIDIA GPU, respectively, measured against a single core CPU implementation [82].

Overall, ViennaCL looks like a promising alternative. Integration with the FEniCS Project would require the implementation of a complete wrapper layer between ViennaCL and DOLFIN. Since OpenCL-based programs can be compiled with standard compilers such as `gcc` [64], the modifications to the DOLFIN build system would probably be smaller than for a CUDA-based library.

PETSc

The Portable, Extensible Toolkit for Scientific computing (PETSc) [52] is the default linear algebra backend used in the FEniCS Project. PETSc is open source, and provides a vast library of C functions for solving the linear systems arising from discretization of PDEs. Since PETSc is written in the C programming language, it is usable from DOLFIN, which is written in C++. PETSc has extensive support for distributed memory parallelism with MPI.

Since version 3.2, PETSc has provided GPU-based classes for vectors and matrices [43]. This implementation is based on Cusp [54], as well as NVIDIAS's Thrust [67] library. Since Cusp is invoked in the end when setting up and solving linear systems, PETSc can be expected to perform on par or somewhat poorer than Cusp. It should be expected that the wrapping of Cusp in PETSc functions will incur some overhead.

7.3 Choosing a library

Cusp, ViennaCL and PETSc stand out among the evaluated libraries as fitting candidates for integration with the FEniCS Project. They all provide extensive functionality and exhibit good performance, and are all released under open licenses. Out of these three, PETSc is by far the largest and most complex library. It is an industry standard library, used by thousands of users worldwide[1] in a large number of high-performance computing projects.[2] It is a mature software library that has been under development for over a decade, and that has just recently implemented support for GPU acceleration. The former two are, in comparison, very young libraries, developed solely for the purpose of GPU accelerating numerical linear algebra. PETSc can be expected to be under further development and support for many years to come, seeing as it is so well established in the market.

Since PETSc is based on Cusp, it can in theory provide all the same functionality as Cusp, provided the functionality is wrapped in suitable PETSc classes and functions. If it happens that Cusp has functionality that is not yet integrated into PETSc, it is possible to patch PETSc to provide the necessary extensions.

PETSc is already integrated with DOLFIN, so the only thing needed to enable GPU-support is to extend the existing DOLFIN wrappers. This will enable the continued use of the full set of PETSc functionality already integrated in DOLFIN, with some parts of it GPU-accelerated. Choosing PETSc will also simplify the user interface, only adding some sort of user parameter to control the switching between CPU- and GPU-based data structures and algorithms for linear algebra when using the PETSc backend.

Choosing PETSc will also facilitate further interesting extensions, such as shared memory parallelism via MPI on clusters of GPU-enabled compute nodes. This last point is of particular interest, as there is a trend toward including GPUs as accelerators in modern supercomputers. Three out of the top five of the 500 fastest supercomputers in the world today use GPUs [78].

7.4 An introduction to PETSc

PETSc provides a large suite of data structures and algorithms for linear algebra, equipped with a flexible interface that allows user control on several levels. This section will give a short introduction to PETSc by going through the library structure, basic usage, and how GPU support is implemented.

1 Nearly 13000 unique downloads of PETSc in 2011: `http://lists.mcs.anl.gov/pipermail/petsc-users/2012-March/012750.html`.

2 See the incomplete list of publications on applications that use PETSC: `http://www.mcs.anl.gov/petsc/publications/index.html`.

7.4.1 Structure

PETSc is divided into a set of sub-libraries, as outlined in Figure 7.1. In this diagram, the libraries depend on those below them; that is, the KSP library build on the Matrix and Vector libraries, and so on. The two dotted boxes at the bottom denote external libraries that PETSc uses, while the topmost round node denotes applications that make use of PETSc.

The sub-libraries are often referred to as classes, although that is technically untrue, as the C language doesn't support the concept of classes. However, for the sake of brevity and consistency with the PETSc documentation, they will also be referred to as classes throughout this text.

All the classes consist of an abstract interface along with several different implementations. In the case of for instance the matrix class, the interface consists of functions for performing various matrix operations. The different implementations are different matrix storage formats, such as the CSR format, the block CSR format, etc. In PETSc, the CSR format is referred to by the name "MATAIJ", or just "aij".

7.4.2 Usage

The user interface of PETSc is relatively low-level, in particular compared to higher-level scientific programming environments such as MATLAB [68]. The *PETSc Users Manual* [34] will take care of the thorough introduction to the usage of PETSc, while this section will show a few code examples to give an impression of how the library is used. First, consider the following excerpt of C code from a program using PETSc:

C code

```
Mat A;
MatCreate(PETSC_COMM_WORLD, &A);
MatSetType(A, MATAIJ);
MatSetSizes(A, 4, 4, 4, 4);
```

This code creates an empty 4×4 matrix and sets the type of the matrix to the CSR format. This matrix is currently empty. To fill the matrix with values, one needs to call a separate function:

C code

```
MatSetValues(A, 1, 0, 1, 0, 2, INSERT_VALUES);
MatSetValues(A, 1, 1, 1, 1, 4, INSERT_VALUES);
MatSetValues(A, 1, 2, 1, 2, 6, INSERT_VALUES);
```

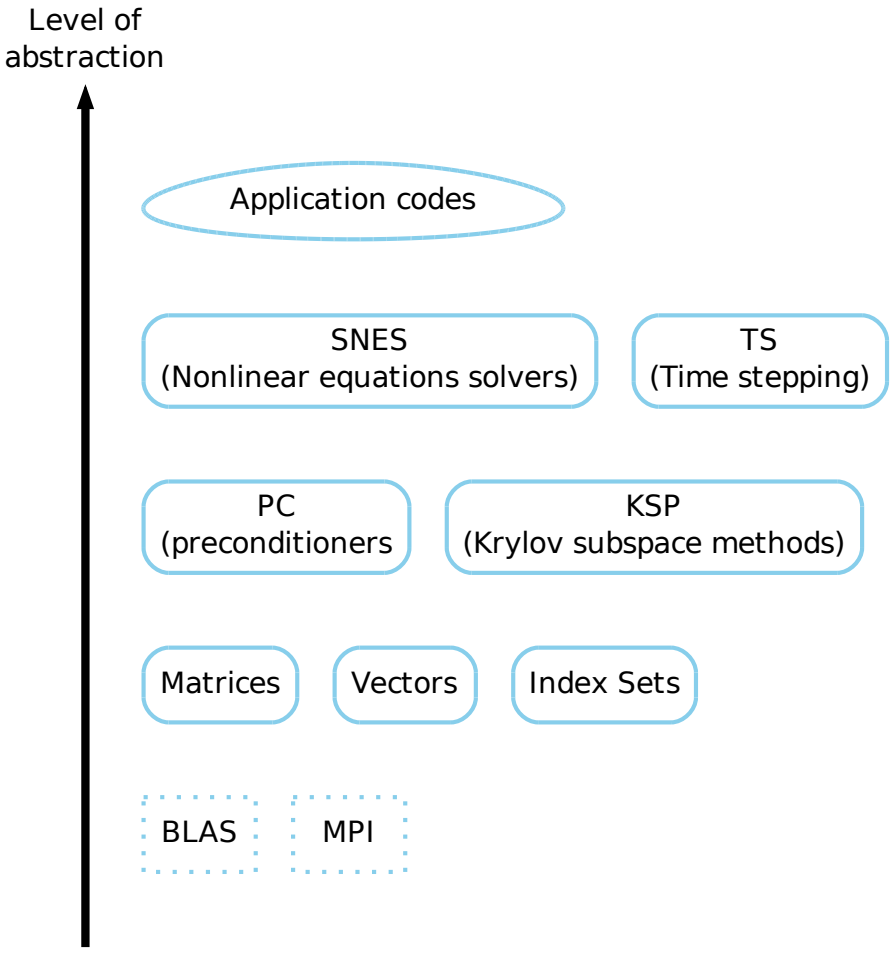


Figure 7.1: Outline of the structure of PETSc. The dotted boxes at the bottom denote libraries that PETSc depends on. The topmost box denotes applications that use PETSc, which depend on the different PETSc libraries depicted in the diagram.

This inserts three values at the locations $(0,0)$, $(1,1)$ and $(2,2)$, so that we end up with the following matrix:

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

This could also have been achieved by a single call to `MatSetValues`, which would entail passing as arguments two vectors of row and column indices, and a two-dimensional array of nine values. Inserting small blocks of values into a large matrix is exactly how one assembles the linear system during finite element assembly.

For larger problems, one should always *preallocate* storage for the nonzero values of the matrix [34, p. 59]. This is done by calling functions such as `MatSeqAIJSetPreallocation`, which can preallocate storage for a sequential[3] CSR matrix. Similar functions exist for distributed matrices, sequential block CSR matrices, and so on. These functions typically accept two parameters[4] that indicate the sparsity pattern of the matrix, and thus how many non-zero values PETSc should allocate storage for. If the number of nonzeros per row is roughly equal, it suffices to pass a single integer that estimates this number. If the number of nonzeros per row varies throughout the matrix, one can pass an array of integers that estimate the number of nonzero values for each row separately. If one does not preallocate storage, one risks that PETSc has to perform extra allocation and copying of values from the old to the new storage during the insertion of values, which is computationally very expensive [34, p. 59].

Before the matrix can be used for anything, it has to be assembled. In this context, "assembling" refers to processing the matrix after a series of value insertion calls, to finalize the matrix for subsequent operations:

C code

```
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

Vectors are created in a similar fashion.

Given a linear system of a matrix `A` and a vector `b`, the system can be solved using the PETSc KSP class, using code similar to the following:

3 In PETSc, "sequential" refers to objects that reside on a single process; i.e., they are not distributed over several processes via MPI. Distributed objects in PETSc are usually called MatMPIAIJ, VecMPI, etc.

4 The exact parameters varies from function to function. For a complete list of all available preallocation functions, see the Matrix class manual pages: `http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/Mat/index.html`.

C code

```
Vec x;
KSP ksp;
KSPCreate(PETSC_COMM_WORLD, &ksp);
KSPSetOperators(ksp, A, A, DIFFERENT_NONZERO_PATTERN);
KSPSolve(ksp, b, x);
```

This code illustrates the characteristics of the user interface. There exist a large number of functions for setting Krylov solver options, getting information about iteration numbers and convergence reasons, etc. Several different Krylov methods are implemented, again as different implementations of the same abstract interface. Any further explanation of the details of this code is omitted, and the reader is again referred to the *PETSc Users Manual* [34] for details about PETSc usage.

7.4.3 GPU acceleration

The support for GPUs in PETSc is based on introducing new matrix and vector classes. The primary design goal behind this implementation is to be able to have the matrix and vector data reside on the GPU during an entire Krylov method solve [43, p. 5]. This eliminates unnecessary copies of matrix and vector data back and forth between the host and the device. The GPU-based vector class performs all its operations on the GPU, including norms and inner products. The new matrix class is designed for sparse matrices, and performs the sparse matrix-vector product (SpMV) on the GPU. The other matrix operations are performed on the CPU as before. However, having the SpMV and all vector operations computed on the GPU ensures that most, if not all, computations during a Krylov solve are performed on the GPU. As pointed out in Section 4.3.4, the SpMV is the computationally dominating part of a Krylov method iteration, so computing this efficiently on the GPU is of particular significance to the overall computational performance of the Krylov solver.

This design allows the code for the Krylov solver algorithms to remain untouched, and they still run on the host. However, when performing vector operations and sparse matrix-vector products in each iteration, CUDA kernels are invoked, and the computations take place on the GPU. Hence, the solvers can be expected to run more efficiently than if the computations were run on the CPU as before. The matrix and vector class implementations are made to work in parallel with MPI, via the matrix and vector types “mpiaijcusp” and “mpicusp”, respectively.

For assembling matrices in PETSc, a new function for GPU-based assembly is provided. Recall the function `MatSetValues` that takes as arguments a matrix, a block of values and the indices to insert the block at. The new function, called `MatSetValuesBatch`, takes as arguments an array of blocks of values and a corresponding array of indices. This allows for insertion of all blocks with one single function call. The batch assembly function assumes that all blocks are square, and that all values are inserted in the same row and column. For instance, given an array of 3×3 blocks of values,

the function expects an array of indices with length three times the number of blocks. Every triple of values in this array represents the indices for each block. These indices are taken as indices *for both row and column*. For example, consider the 3×3 matrix A and the index set i:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \; i = \{2, 4, 5\}.$$

Inserting this into an initially empty 5×5 matrix B, we get

$$B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a_{11} & 0 & a_{12} & a_{13} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & a_{21} & 0 & a_{22} & a_{23} \\ 0 & a_{31} & 0 & a_{32} & a_{33} \end{bmatrix}.$$

As we can see, the resulting matrix is symmetric. As the sum of symmetric matrices is also symmetric, this function produces a symmetric matrix. In version 3.2 of PETSc, the most current version at the time of writing, inserting several batches is not supported. Hence, all local element matrices must be passed in a single call to `MatSetValuesBatch` . This function uses functions from Cusp [54] and Thrust [67] to first copy all local matrices to the GPU with a single copy function call, and then assemble these matrices to a global matrix on the GPU.

Given that the necessary wrappers and extensions to the DOLFIN code can be provided, PETSc's GPU-accelerated linear solve and matrix assembly can be utilized in DOLFIN. For further details about the GPU acceleration of PETSc, see the paper by Minden, Smith, and Knepley [43].

8 Implementation

The new, GPU-accelerated implementation of DOLFIN will be presented in this chapter. This implementation consists of three parts:

- Integration of GPU-based matrices and vectors to accelerate the solution of linear systems.
- A preliminary implementation of batch insertion of local element matrices on the GPU, to accelerate the assembly of linear systems.
- Extensions to the FEniCS build system to automate the compilation of PETSc and DOLFIN with GPU-support enabled.

These parts will be discussed in Sections 8.3, 8.4 and 8.5, respectively.

Some background on the previous implementation will be given before discussing the new implementation. First, the previous implementation of the assembly algorithm will be profiled in Section 8.1, to identify potential performance bottlenecks. In Section 8.2, the general layout of the PETSc wrapper layer in DOLFIN will be explained.

8.1 Profiling the assembly algorithm

The batch assembly function in PETSc was introduced in Section 7.4.3. This function uses Cusp and Thrust calls to insert all local element matrices into the global matrix at once, avoiding the incremental assembly. As mentioned in Section 6.2, it has been pointed out that the insertion of the local matrix into the global matrix is a performance bottleneck when running assembly on the GPU. In order for there to be any point in integrating the batch assembly function in DOLFIN, it should be investigated if this insertion is a bottleneck on the CPU as well. If it is, then it can be concluded that integrating the batch assembly function could be worthwhile. To this end, we first examine how the iterative insertion performs in the current implementation of the assembly algorithm.

The assembly algorithm outlined in Algorithm 2.1 is indeed the one implemented in DOLFIN, but the whole assembly process is somewhat more involved. Several helper classes and functions are put into play when assembling, e.g. when calling the `assemble` function in Python. An overview of the process can be seen in Algorithm 8.3, and a diagram of the involved classes can be seen in Figure 8.1 on page 69. The

diagram shows the classes involved when assembling a matrix. Note that it only shows the generic classes, omitting the backend-specific wrapper classes. Also note that the computed sparsity pattern is used by the `init` method of the matrix classes to perform preallocation of storage for nonzero values, as described for PETSc in Section 7.4.2.

Algorithm 8.3 Overview of the process for assembling a tensor in DOLFIN.

(1) Initialize the global tensor:
  (i) Initialize a *tensor layout*
  if tensor has rank 2, i.e., tensor is a matrix:
    (ii) Build a *sparsity pattern* (for tensors of rank 2) to be held by the tensor layout
  end if
  (iii) Call the `init` method of the tensor and pass it the tensor layout
(2) Iterate over mesh entities, compute and insert local element tensor for each entity, similar to Algorithm 2.1:
  (i) Assemble over *cells*
  (ii) Assemble over *interior facets*
  (iii) Assemble over *exterior facets*
(3) Finalize tensor by calling `A.apply()`

The computation and insertion of the local element tensor happens in two separate function calls. Hence, in order to time the significance of the run time of the insertion to the total run time, the whole process with and without the insertion method call commented out can be timed. By commenting out the method call for the tensor calculation as well, the baseline performance of an "empty" cell/facet iteration can be measured.

For profiling the assembly, the matrix arising from discretization of Poisson's equation on a unit cube was assembled. We let the resolution of the mesh of the unit cube and the basis function degree vary. The bilinear form of Poisson's equation is given in Eq. (2.1.13). As we see, it has only an integral over cells, so emphasis is put on that part of the assembly pipeline, as the other two parts will not be run in this case.

The run times for four runs can be seen in Table 8.1 on page 70. In the table, "No insertion" means that the local element tensors are only computed without being inserted. "No computation" means that they are neither computed nor inserted; that is, only an empty cell iteration is performed, with nothing being done for each cell. The "Cells" and "Total" columns represent the run time to assemble over cells, and the total time consumed by the `assemble` function, respectively. In addition to assembling over cells, the `assemble` function performs matrix initialization and finalization, computes the matrix sparsity pattern and a few other operations, which contribute significantly

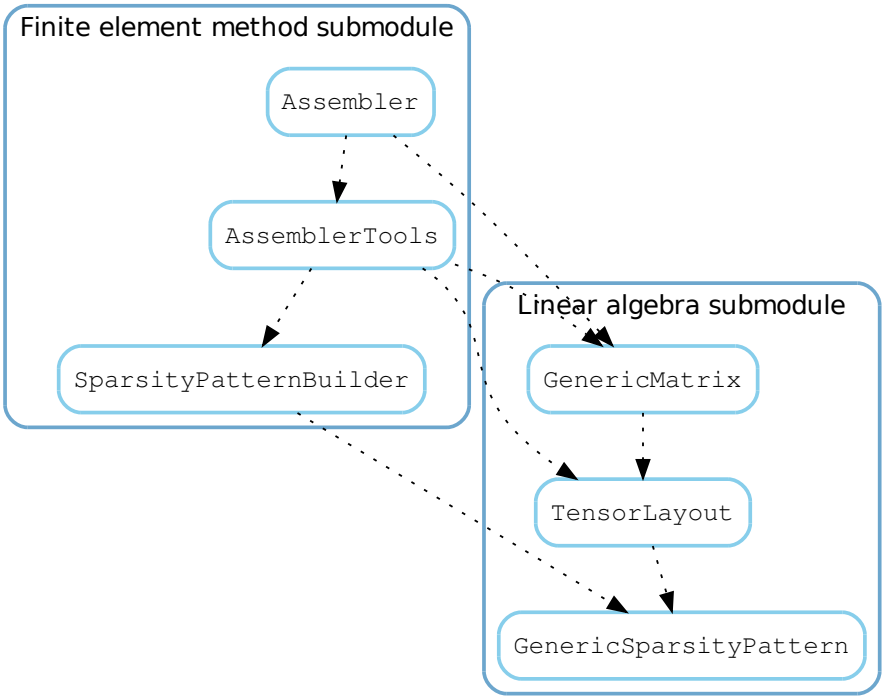


Figure 8.1: Outline of the classes that take part in the assembly process.

Poisson 3D			Standard		No insertion		No computation	
k	n	N	*Cells*	*Total*	*Cells*	*Total*	*Cells*	*Total*
3	32	912673	4.4457	16.6454	0.9505	12.9502	0.0787	12.5554
2	50	1030301	2.8925	11.6252	0.5072	9.3488	0.1684	9.4423
1	100	1030301	4.7569	9.8101	2.172	7.1895	0.6442	6.0943
1	70	357911	1.3972	3.2480	0.7490	2.6778	0.3048	2.0715

Table 8.1: Run times for several runs of matrix assembly for Poisson's equation. The variable k denotes the basis function polynomial degree, n denotes the number of cells in each direction for the unit cube mesh, and N denotes the size of the assembled $N \times N$ matrix.

to the total run time of the function.

All timings are performed using DOLFIN's Python interface. Using C++, the results are very similar, with the overhead using Python being only some tenths of a second. Since only potential performance bottlenecks are investigated here, this small overhead does not matter. The runs were performed on a computer with an Intel i3 2.1 GHz dual core CPU, with 4 GB of RAM.[1]

By comparing the numbers in the fourth and sixth columns of Table 8.1, we see that the local element tensor insertion accounts for a large part of the runtime when looking at assemble cells isolated. However, looking at the numbers in the fifth and seventh columns, to consider the assembly process as a whole, we observe that the significance of the insertion declines. On average for the four test cases, the insertion accounts for 22 percent of the total run time of the `assemble` function.

Using Amdahl's law, given in Eq. (3.5.2), it can be calculated how much speedup the assembly algorithm can be expected to achieve overall, when using a parallel local element matrix insertion routine. In our case, the parallelizable portion is 0.22 on average. The speedup of this portion is denoted S_p. When S_p approaches infinity, we get a maximum possible overall speedup factor of 1.27.

This is a crude over-estimate, as S_p will have much more moderate values in practice. A realistic estimate could be something in the range of 2 to 6. This, in turn, yields estimates for the overall speedup factor in the range of 1.12 to 1.22. While 10 to 20 percent speedup is not very much, it can definitely be valuable when performing time-consuming assembly of large systems. Therefore, it can be concluded that an attempt at accelerating the matrix insertion is worthwhile.

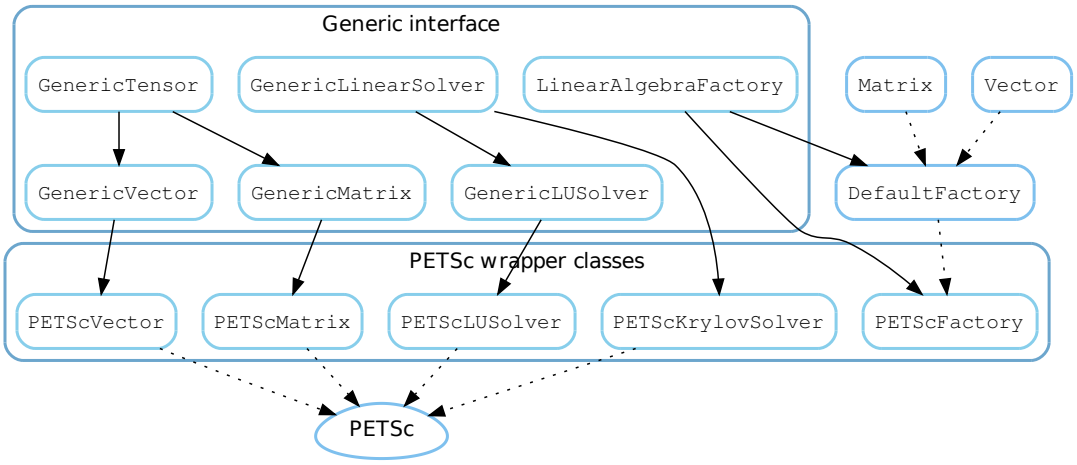


Figure 8.2: An overview of the generic linear algebra interface and the PETSc wrapper classes. Solid lines denote inheritance, dotted lines denote usage. Some of the less relevant PETSc classes are omitted for the sake of brevity. Note that also that the `Matrix` and `Vector` classes inherit from `GenericMatrix` and `GenericVector`, respectively, but these lines are also omitted to keep the diagram clear.

8.2 Structure of the PETSc wrapper layer

Sections 5.2.1 and 5.2.2 provided a brief discussion of DOLFIN's generic linear algebra interface, as well as some code examples for how to assemble and solve linear systems in DOLFIN. The integration of linear algebra backends in DOLFIN will now be discussed, with emphasis on the PETSc backend. This section presents the PETSc wrapper layer prior to the changes implemented as part of this project.

An overview of DOLFIN's linear algebra submodule can be seen in Figure 8.2. The diagram shows the abstract interface along with the PETSc wrappers that implement this interface. The solid arrows denote inheritance, while the dotted arrows denote dependencies. The classes for the other backend wrappers are omitted, as are some of the less relevant PETSc classes. A complete diagram of all PETSc classes can be seen in Appendix B. The implementation is based on the *factory method pattern*, where each backend has a corresponding factory class that takes care of instantiation of matrix and vector objects for that backend.

If DOLFIN is configured with PETSc, it is set as the default backend. The backend can be explicitly chosen by the following lines in Python:

1 The fact that this computer has relatively low performance does not matter in this case, as we are only interested in the ratios of run times of the different stages of the computation, not trying to achieve a low run time overall.

Python code

```
name = 'PETSc' # or 'uBLAS', 'MTL4', 'Epetra', ...
parameters.linear_algebra_backend = name
```

This ensures that all subsequent linear algebra operations are computed using the chosen backend. Matrices and vectors can be created directly by the user in the following way:

Python code

```
# Create an empty matrix
m = Matrix()
# Create an empty vector
u = Vector()
# Create a 100-long vector
v = Vector(100)
```

Here, the constructors of the `Matrix` and `Vector` classes use the `DefaultFactory` class to create instances. This factory checks the currently set linear algebra backend, and in turn invokes the factory corresponding to that backend. Alternatively, one can create instances of specific matrix and vector classes:

Python code

```
# Create vector with PETSc backend
u = PETScVector()
# Create vector with uBLAS backend
v = uBLASVector()
# Create matrix with MTL4 backend
m = MTL4Matrix()
```

By only using the generic `Matrix` and `Vector` classes, a program can be written to be backend-independent. That way, the program can be run using different backends by simply setting the `linear_algebra_backend`-parameter in the beginning of the program.

If a program is run in parallel using MPI, the factories automatically creates distributed vector and matrix objects. This is only supported when using backends that support shared memory parallelism via MPI, such as PETSc and Trilinos.

The polymorphism of the interface is used for instance in the assembly algorithm. The algorithm accepts a `GenericTensor A` as input, and iteratively calls the method `A.add()` to insert a block of values into the tensor. The method call is in turn resolved in the class of which `A` is an instance, for example `PETScMatrix`.

Now follows a discussion of the extensions provided to integrate GPU-based matrices and vectors.

8.3 Integration of GPU-based matrices and vectors

In order to be able to instantiate matrices and vectors of the GPU-enabled types, the parameter list for `parameters.linear_algebra_backend` was extended with a new option, called `PETScCusp`. Unlike the other possible choices for this parameter, the `PETScCusp` option does not actually invoke a separate backend library, but rather a new factory, called `PETScCuspFactory`. This, in turn, calls the constructors of `PETScMatrix` and `PETScVector` with an added parameter, which indicates that the tensors should be instantiated with a PETSc type that is GPU-enabled. The default value for this parameter leads to the tensors being instantiated to be of CPU type. Hence, the constructors can be called without passing the parameter, and the existing `PETScFactory` does not need to be modified. This approach allows for reuse of all the existing code in the PETSc matrix and vector classes, but also allows for switching between GPU and CPU at a high level in the user interface.

As we saw in Section 7.4.3, the Krylov solvers in PETSc run on the GPU as long as the linear system is defined using a matrix and a vector that are both GPU-based. Hence, these changes are all that is necessary to have the whole linear solve process run on the GPU.

Since the GPU-based tensors are designed to work in parallel via MPI, adding support for this should in theory be straightforward. However, the implementation of the PETSc library itself has some limitations in this regard. The distributed vectors in DOLFIN use the concept of *ghosted vectors*, a special kind of distributed vector in PETSc. These vectors are created via a shorthand function called `VecCreateGhost`, which automatically creates the vector and sets the length and type. This function sets the vector type to VECMPI, the standard distributed PETSc vector type. For vectors created with `VecCreateGhost`, it is not possible to change this to another type. Specifically, it cannot be changed to VECMPICUSP, which is what we need in this case.

Hence, in order to integrate MPI-enabled GPU vectors in DOLFIN, one would either have to rewrite the DOLFIN wrappers to not use ghosted vectors, or apply patches to PETSc to support GPU-based ghosted vectors. Neither of these options were pursued because of time constraints, and are hence the subject of future work.

8.4 Implementing batch assembly

As we saw in Section 8.1, the local element matrix insertion step accounts for a significant part of the run time of the `assemble` function. Hence, accelerating it may yield some performance improvements. This section describes the preliminary integration of the batch assembly function in DOLFIN. This function, called `MatSetValuesBatch`, was introduced in detail in Section 7.4.3. In this attempt at integrating it, only batch insertion of the local *cell matrices* was implemented. Bilinear forms including facet

integrals are hence not supported. Therefore, in order to measure the performance of the batch assembly, the assembly of bilinear forms without facet integrals must be benchmarked.

The batch assembly function makes it possible to insert all local element matrices with a single function call. This function is only concerned with the insertion of the matrices, not the computation. GPU-based computation of the local matrices has not been considered in this work, so they will still be computed in serial. Thus, it is fine to keep much of the serial assembly algorithm as it is, especially the iteration over mesh entities.

However, since the matrices are computed one by one in a loop, they must somehow be saved, or cached. Fortunately, the existing call to `A.add()` for each cell can be utilized to perform that caching. By turning this call into a low-latency operation that only caches the matrices, the actual assembly of the matrix can take place during the finalization phase. By only changing the methods of the corresponding PETSc classes, we see that more or less the entire assembly algorithm can be kept as it is. Specifically, the generic nature of the algorithm is kept by adding all batch assembly logic to the `PETScMatrix` class. The only necessary changes to the assembler class is the passing of additional metadata to `PETScMatrix`. This will be described further in the following section.

8.4.1 Adding a class for caching of local element matrices

The incremental caching of the matrices by turning `A.add()` into a low-latency call is achieved by introducing a new, lightweight matrix class, called `UnassembledMatrix`. The member variables of this class is a flat array of double precision floating point values, as well as an array of indices on the format that `MatSetValuesBatch` accepts. The value array holds the entries of the matrix as flattened blocks of values, conforming to the interface of `MatSetValuesBatch`. In addition, the class holds the number of blocks and the number of rows (and columns) of each block.

This class is thus not able to store arbitrary matrices, only those that can be decomposed into a number of square, equally sized blocks. Since the blocks are simply saved in arrays, one can implement a fast `add` method that inserts a block of values into the matrix. The `add` method defined in the `GenericTensor` interface already accepts a block of values as a flattened array, hence the insertion into the unassembled matrix is a simple memory copy from one array to another.

The disadvantage of this approach is that the number of blocks and their size must be known when the unassembled matrix is initialized, to allocate storage for the arrays. The good news is that this information is readily available at the start of the assembly algorithm. Thus, if there is a way to pass that information to the unassembled matrix before entering the cell iteration, it can be initialized in time for the iterative caching of the local element matrices.

This information is calculated and stored in an instance of the `TensorLayout` class,

which is passed on to the matrix class when initializing the matrix at the beginning of the assembly algorithm. The solution is to let `PETScMatrix` hold a reference to an instance of `UnassembledMatrix`. When the `TensorLayout` instance is passed to the `init` method of the PETSc matrix, the relevant data can be passed along to the unassembled matrix such that it can be initialized accordingly. Here, it can also be checked whether the tensor layout indicates that the blocks are of different sizes, or not square. If so, the blocks are incompatible with the `MatSetValuesBatch` function. Thus, a warning is issued to the user, the unassembled matrix is not initialized and we proceed with the regular incremental assembly instead.

The `add` method of `PETScMatrix` is modified such that the added block is forwarded on to the `add` method of `UnassembledMatrix` for caching. The `MatSetValuesBatch` function is called during finalization of the matrix. This happens in the `apply` method, which is called when the iteration of mesh entities is complete. The `apply` method of `PETScMatrix` usually calls `MatAssemblyBegin/End`, but in this case calls the batch assembly function instead, passing it the values held by the unassembled matrix.

The final class layout is outlined in Figure 8.3. This is a modified version of Figure 8.1, only showing the case of matrix assembly with PETSc. As seen from the figure, the `UnassembledMatrix` class is used only by the `PETScMatrix` class. None of the other classes are aware of the existence of this helper class.

8.5 Automated compilation and installation

The components of the FEniCS Project are distributed in a number of ways.[2] Precompiled binary packages are provided for Windows, Mac OS X, Debian and Ubuntu. In addition, all source code is freely available and may be downloaded and compiled manually.

Dorsal [61] is a tool that automates the process of building the components from source. It is a convenient tool for those who use platforms for which binary packages does not exist, or those who wants to build from source for other reasons. Dorsal functions as a very simplified package manager, which automatically downloads, compiles and installs all components. The structure of Dorsal makes it easy to modify and tweak compilation parameters, change version numbers and add support for new packages.

I have provided extensions to Dorsal that makes it possible to automatically build PETSc with GPU support. These extensions download and unpack the Cusp package and configure PETSc with support for Cusp and Thrust. Thrust is shipped with the CUDA Toolkit [58], which both Cusp and Thrust depend on. Unfortunately, there is no way to automatically install the toolkit on any of the platforms supported by Dorsal. This is due to the fact that NVIDIA only provides command-line based installation utilities for the toolkit. In addition, the toolkit requires the NVIDIA developer graphics drivers to be installed, which must also be installed manually via a command-line

2 See the Download-section at the FEniCS Project web pages: `http://fenicsproject.org/download/`.

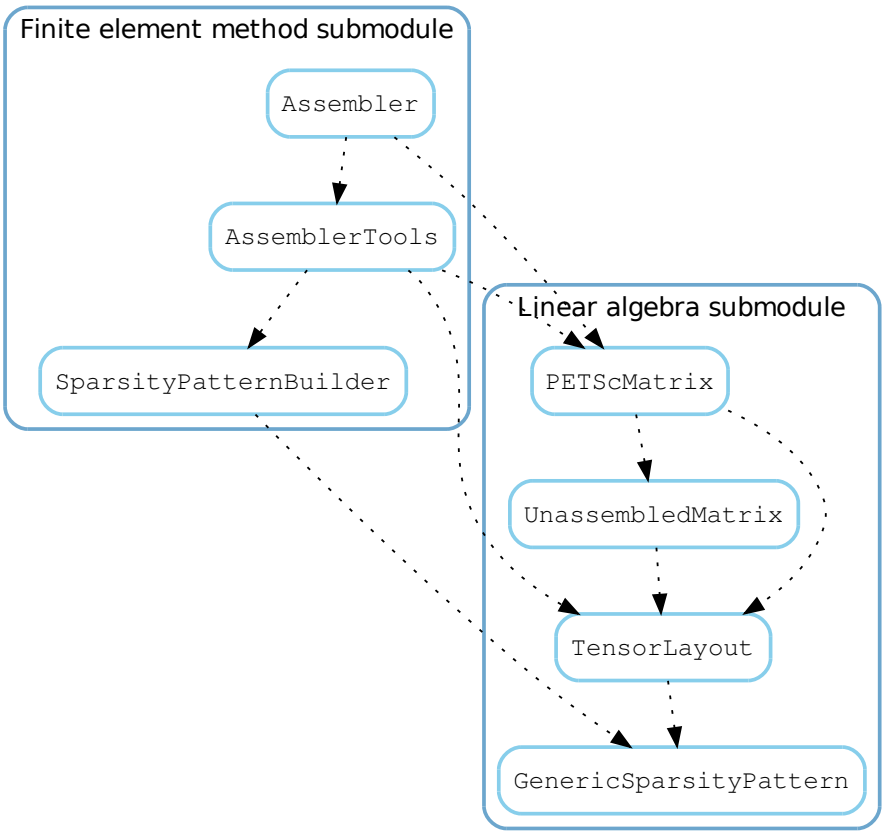


Figure 8.3: A modified version of Figure 8.1 that outline the classes that take part in the batch assembly process.

interface. These command-line-based utilities could in theory be run automatically by Dorsal, but an undertaking to implement this is outside the scope of this project.

Therefore, Dorsal assumes that the user has successfully managed to set up a working CUDA environment, and automates the remaining parts of the installation process.

9 Performance analysis

In this chapter, a performance study of the implementation is presented. Both the assembly and solve implementations are benchmarked for several test problems.

9.1 Methodology

When it comes to testing and benchmarking the implementation, both *correctness* and *performance* will be evaluated. Although attaining good performance is the main goal, it is crucial that the solutions computed by the GPU-based linear algebra backend are equal (to within a tolerance) to the ones computed by the CPU-based backend. Correctness will therefore always be verified before the performance is evaluated.

The accuracy of the computed solutions depend on the Krylov solver *tolerances*. The *relative tolerance* of the Krylov solver is a tolerance for the relative decrease in the norm of the residual, used as the stopping criterion of the Conjugate Gradient method, listed in Algorithm 4.2. The default value for the relative tolerance in DOLFIN is 1×10^{-6}. The *absolute tolerance* is a tolerance for the absolute size of the residual norm. This is not included in the stopping criterion of Algorithm 4.2, but it is used as an additional stopping criterion in PETSc[1] and DOLFIN: if the absolute norm of the residual gets less than the absolute tolerance, the iteration stops. The default value in DOLFIN for the absolute tolerance is 1×10^{-15}. Both tolerances will be left to their default values throughout the experiments.

The correctness is verified by computing the l_2-norm of the difference of the solution vectors produced by the two linear algebra backends. If this error is small, the solutions are interpreted to be equal and it is concluded that the GPU-based implementation produces the correct solution. In this context, an error comparable to the relative tolerance of the Krylov solver is labeled as small.

Each of the test problems discussed in Section 2.3 will be solved and the performance of the GPU-based backend will be evaluated for each run. Among these problems, Poisson's equation and the linear elasticity problem will be thoroughly benchmarked. These two problems are divided into four groups that will be benchmarked separately:

- Poisson's equation in 2D on the unit square, with element degree $k = 1, 2, 3$.

1 See the PETSc documentation: `http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/KSP/KSPDefaultConverged.html`.

- Poisson's equation in 3D on the unit cube, with element degree $k = 1, 2, 3$.
- Linear elasticity in 2D on the unit square, with element degree $k = 1, 2, 3$.
- Linear elasticity in 3D on the unit cube, with element degree $k = 1, 2, 3$.

The different values of k result in matrices with different density and structure, which affects how computationally demanding it is to solve the corresponding linear systems. For each of these four groups, the following metrics will be measured:

Run time
: The relevant calls to DOLFIN functions, specifically `assemble` and `solve`, will be isolated and timed. The raw run time of these functions for different problems and meshes can then be presented. By experimenting, mesh resolutions have been found that yield roughly the same number of unknowns in the linear system for each value of k for each equation. This makes it convenient to see how the run times vary with k for each equation. By increasing the mesh resolution, it is possible to analyze how the run time increases with the number of unknowns. For each problem, the largest mesh yields around 5×10^5 unknowns. This is a system size for which most of the problems are possible to solve under the memory constraints imposed by the GPU, with only one exception. Hence, a good basis for comparison of performance is established.

Speedup
: By dividing the run time for the CPU-based backend by the run time for the GPU-based backend, the *speedup* is obtained, as described in Section 3.5.1. In this case, only the speedups of the `assemble` and `solve` functions are evaluated, without measuring the speedup for the complete simulation.

  However, the solve step often accounts for a significant fraction of the total run time, and this fraction is measured for all problems.

Flops
: In the case of linear solve, the number of floating point operations performed per second by the Krylov solver will be measured. The flop rates will be measured and compared for both the CPU and the GPU. These numbers are obtained using the built in profiling tool of PETSc [34, Ch. 11]. This tool reports aggregated run times and flop rates for all PETSc functions invoked during the entire life of a program. Hence, to isolate the flop rates for the linear solve for each of the problems, each problem must be run in a separate program instance. This is achieved by having one Python program control the execution of several smaller Python programs.

These metrics will be measured for all of the test problems, not only Poisson and linear elasticity. However, an extensive performance study with analysis of the metrics

for different problem sizes will not be conducted for the other problems. Rather, the correctness of the computed solutions will be checked, and it will be verified that the performance of the GPU-based implementation is better than that of the CPU-based implementation.

We remark that the GPU-based implementation is benchmarked against code that is run in serial, on one processor core only. In other words, OpenMP or MPI for multicore or parallel execution is not used. We would expect less of a speedup if comparing against multicore or parallel CPU runs. However, it is known that running a program in parallel on an n-core processor would give a speedup factor of at most n. This can be kept in mind when evaluating the timing and speedup results, but will not be commented upon any further in the following performance analysis.

Throughout this chapter, the existing implementation will interchangeably be referred to as "PETSc", "the CPU-based linear algebra backend" or similar, and the new implementation as "PETScCusp", "the GPU-based linear algebra backend" or similar. All benchmarks are run in double precision arithmetic on a desktop computer with an Intel Xeon 6-core W3680 3.33 GHz processor, 24 GB of memory and an NVIDIA GeForce GTX 590 GPU with 3072 MB of global memory.[2]

9.2 Linear solve

9.2.1 Choice of linear solver

PETSc provides a vast family of Krylov methods, and a subset of these are integrated in DOLFIN. The same is the case for preconditioners. As mentioned in Section 7.4.3, all of the Krylov methods are GPU-accelerated when one uses the GPU-based matrix and vector classes. Among the preconditioners (PCs) from PETSc that are integrated in DOLFIN, only the following three run on the GPU at the time of writing:[3]

- Jacobi (diagonal) preconditioner (see Section 4.1.2).
- Block Jacobi preconditioner.
- Additive-Schwarz preconditioner.

Before performing the actual benchmarking, the Krylov method and preconditioner to use to solve each problem must be chosen. This was decided by numerical experiments, in which all of the problems were solved using all combinations of Krylov methods and preconditioners.

2 The GTX 590 GPU is a dual-chip GPU, which means it is essentially two GeForce GTX 580 GPUs in one unit. Hence, to utilize all the resources, it must be programmed as if one is programming several GPUs at once. As this current implementation does not support multiple GPUs, only one of the two chips are utilized during the benchmarking.

3 See the section about support for NVIDIA GPUs on PETSc's web page: `http://www.mcs.anl.gov/petsc/features/gpus.html`.

These experiments showed that the method of Conjugate Gradients (CG) was the most effective Krylov method for the symmetric problems, namely Poisson and linear elasticity. When it comes to the DG formulation of the advection equation, which gives a nonsymmetric system, the Generalized Minimal Residual method (GMRES) was the best choice. For GMRES with the block Jacobi and Additive-Schwarz preconditioners, the CPU and GPU solutions differed by only a factor of 10^{-14}. The GPU backend failed to compute the correct solution with the Jacobi preconditioner, giving only a zero solution. Among the two preconditioners that work for both backends, block Jacobi was the fastest one. The GMRES method with block Jacobi preconditioning was therefore chosen for this problem.

Regarding preconditioners for Poisson and linear elasticity, the Jacobi and block Jacobi preconditioners were the fastest. For the CPU backend, block Jacobi performed better, whereas the Jacobi preconditioner performed better for the GPU backend. A fair comparison would be to measure the metrics for solving the problems using the best-performing pair of method and preconditioner for each backend and problem. In addition to this, the problems will be solved using the Jacobi preconditioner for both backends to see what results this yields.

Note that the systems solved in the following sections are assembled using the regular assembly algorithm, not the batch assembly version presented in the previous chapter.

9.2.2 Poisson's equation

The first linear systems we investigate are those resulting from the discretization of Poisson's equation, as given in Eqs. (2.3.24) to (2.3.27). The two and three dimensional cases are considered separately below.

2D

To verify the correctness of the GPU-based linear solve, we compare the solutions computed by the GPU- and CPU-backends. In particular, the l_2-norm of the difference of the two solutions is examined. This norm is examined for the solutions for the various mesh sizes and element degrees. With a relative Krylov solver tolerance of 1×10^{-6}, the simulation data show that the norms are of order 1×10^{-9} or less for Jacobi preconditioning and of order 1×10^{-5} or less when using block Jacobi for CPU and Jacobi for GPU. In general, the solutions differ more when using different preconditioners for the different backends, but the norm of the difference is still not much larger than the relative Krylov solver tolerance. Further numerical experiments reveal that the norm of the difference decreases as the Krylov solver tolerance is reduced. We conclude that the PETScCusp backend computes correct results for this problem.

The factors of the total run time spent in the `solve` function using the original CPU-based implementation can be seen in Table 9.1 on the following page. The intervals in

	$k = 1$	$k = 2$	$k = 3$
Poisson 2D	$[0.34, 0.67]$	$[0.61, 0.91]$	$[0.77, 0.96]$
Poisson 3D	$[0.05, 0.09]$	$[0.28, 0.42]$	$[0.51, 0.70]$
Elasticity 2D	$[0.69, 0.89]$	$[0.81, 0.94]$	$[0.79, 0.93]$
Elasticity 3D	$[0.21, 0.35]$	$[0.19, 0.34]$	$[0.12, 0.23]$

Table 9.1: Ranges for the factors of time spent in the `solve` function for PETSc for the different problems. For each interval, the lower limit is the factor for the smallest mesh, while the upper limit is the factor for the largest mesh.

this table give the smallest and largest factor of time spent in `solve` for that problem for the various meshes. This factor increases with system size for all problems: the lower limit is achieved for the smallest mesh and the upper limit is achieved for the largest mesh. For this problem, we see that the factor of the total run time spent in the `solve` function varies between 0.34 and 0.96.

We now evaluate the performance of the GPU-based implementation. Figure 9.1 on page 84 shows run times for linear solve of Poisson's equation discretized on the unit square, using meshes of increasing size. Figure 9.1a shows results using Jacobi preconditioning for both backends, whereas Figure 9.1b shows results using block Jacobi preconditioning for CPU and Jacobi preconditioning for GPU. We first discuss the case $k = 1$. For PETSc, the run time is lower using block Jacobi than Jacobi preconditioning as previously mentioned. Specifically, the PETSc solve runs roughly twice as fast with block Jacobi preconditioning for all meshes. For both backends and both preconditioning approaches, the run time seems to depend almost linearly on the size of the linear system. We observe that PETScCusp runs significantly faster than PETSc for both preconditioning approaches and all mesh sizes.

The speedups are plotted in Figure 9.2 on page 85. The speedup factors vary with the size of the linear systems: they lie between 2.8 and 6.2 for Jacobi preconditioning and between 1.6 and 3 for block Jacobi preconditioning. For both preconditioning approaches, the speedups start low and increase with the size of the system. Still looking at $k = 1$, it seems as if the speedups would continue to increase for larger systems. Comparing Figure 9.2a with 9.2b, we see that the speedup is lower when using block Jacobi for PETSc than when using the same preconditioner for both backends. This is due to the fact that block Jacobi runs faster than Jacobi for PETSc, as we have generally observed.

The flop numbers are plotted in Figure 9.3 on page 86. We see that PETScCusp yields flop rates several times higher than PETSc. The flop rates for PETScCusp start low and increase with the size of the linear system, as for speedups. It also seems as if they would continue to rise for larger systems. The rates for PETSc, however, seem to decrease with the system size. This behavior is most evident for the Jacobi preconditioning scheme. The flop rates for PETScCusp are in the range of 4000 to

8500 megaflops, or 4.0 to 8.5 gigaflops. PETSc, on the other hand, shows flop rates in the range of 1100 to 1600 for Jacobi preconditioning. When looking at the flop rates achieved for the largest system, PETScCusp yields numbers that are 6.4 times higher than those of PETSc. We observe that this is very comparable to the maximum speedup numbers observed for this problem.

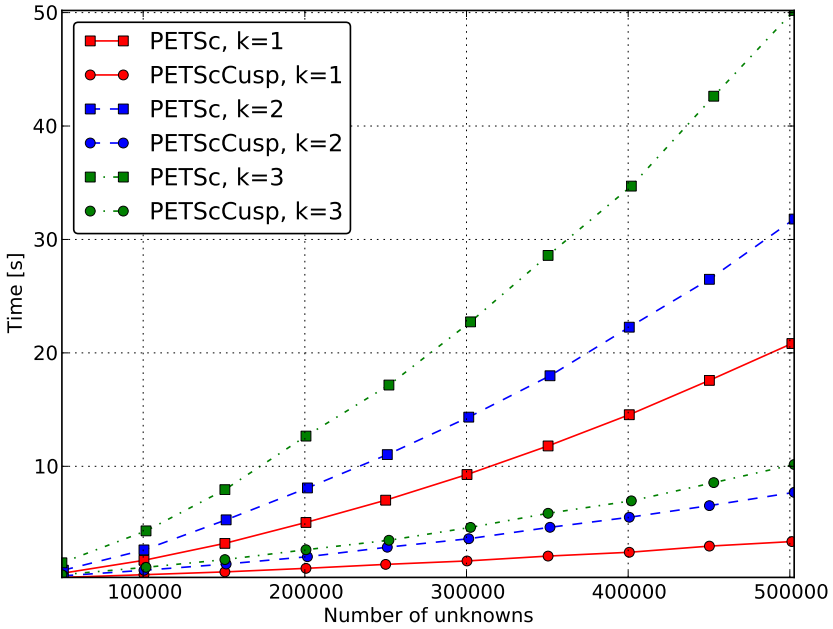
An important detail to note is that for PETSc, block Jacobi preconditioning seems to yield lower flop rates than Jacobi. This is the opposite of what we would expect, as block Jacobi runs much faster than Jacobi. This behavior stems from the fact that the flop rates reported by PETSc are only for the Krylov solve itself, without including the computations related to preconditioning. Hence, the actual flop rates for block Jacobi preconditioned solve are most probably higher than shown in the plot.

Another thing to note is that the flop rates are far from the theoretical peak flop rate of the GPU. The peak flop rate of the GTX 580 GPU is around 200 gigaflops,[4] which is over an order of magnitude more than what is obtained in practice. This is probably due to the fact that Krylov solvers are usually memory bandwidth limited [43], which makes full utilization of floating point performance difficult. This is a property of Krylov solvers in general and the PETSc implementations in particular, and is hence not an artifact of the integration of PETSc in FEniCS. Thus, the same behavior would be seen in pure C programs using PETSc's Krylov solvers.

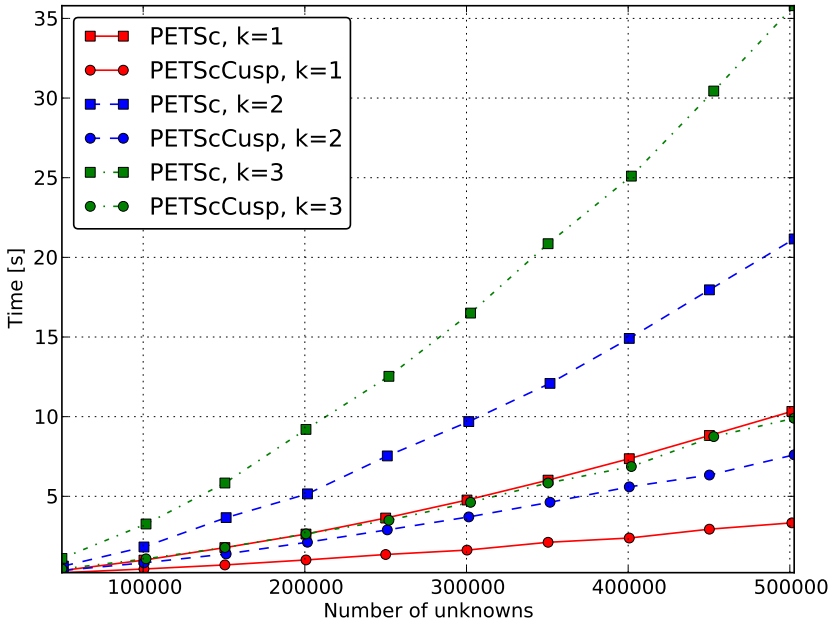
We now continue to the case $k = 2$. Regarding run time, we see the same general behavior as for $k = 1$. PETScCusp is faster than PETSc for each system size, and the run time exhibits almost linear dependence on the size of the system. The difference in run time between the backends is significant, but it gets lower when using block Jacobi preconditioning for PETSc. We see that this case runs slower than $k = 1$. The speedups are similar to $k = 1$ in that they start low and increase with the system size. However, they seem to flatten out and stabilize pretty fast. They also reach a lower maximum speedup factor than for $k = 1$, stabilizing at a factor of around 4 for the Jacobi case and between 2.5 and 3 for the block Jacobi case. Also, similar behavior as for $k = 1$ is seen for the flops. The flop rate for PETScCusp starts low and increases with the system size, but quickly flattens out and stabilizes at a value significantly below that of $k = 1$. Still, the attained flop rate is much higher than that of PETSc.

Last, we look at $k = 3$. We once again see similar behavior for run time as for $k = 1, 2$. This case runs even slower than $k = 2$, but PETScCusp is still faster than PETSc. The speedup increases with system size as for $k = 1, 2$. It stabilizes at factors 5 and 3.5 for Jacobi and block Jacobi preconditioning, respectively. In the block Jacobi case, it actually exhibits more speedup than both $k = 1$ and $k = 2$. The flop rates for PETScCusp for $k = 3$ stabilize at around 7000 megaflops for both preconditioning approaches, which is around halfway in between the values attained for $k = 1, 2$.

4 NVIDIA doesn't report the exact peak flop rate of the GTX 580/590 GPUs. The single precision performance is estimated to be around 1500 gigaflops: `http://en.wikipedia.org/wiki/GeForce_500`. As pointed out in Section 3.3, the GeForce GPUs have an $8\times$ performance penalty for double precision calculations over single precision. This places the GTX 580 at around 200 gigaflops in double precision.

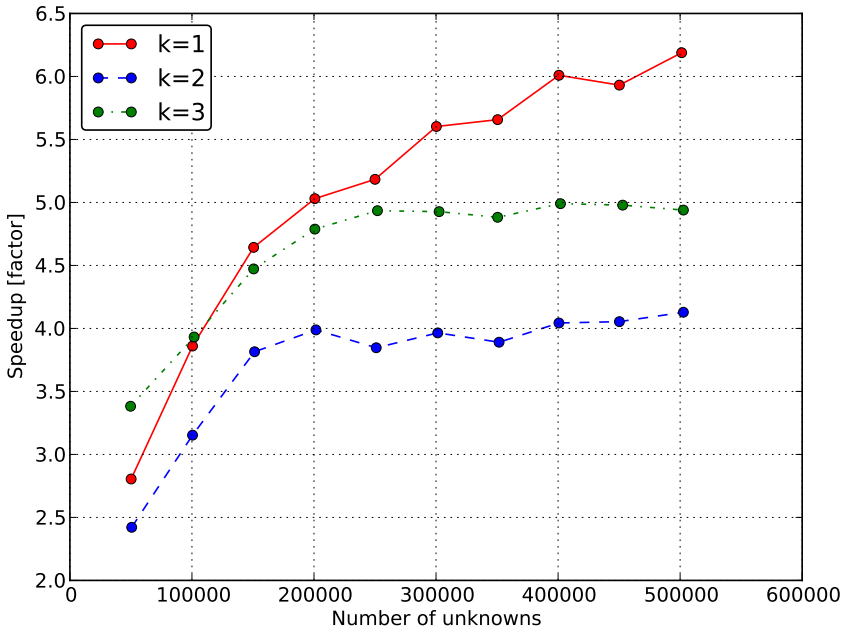


(a) Jacobi PC for both backends

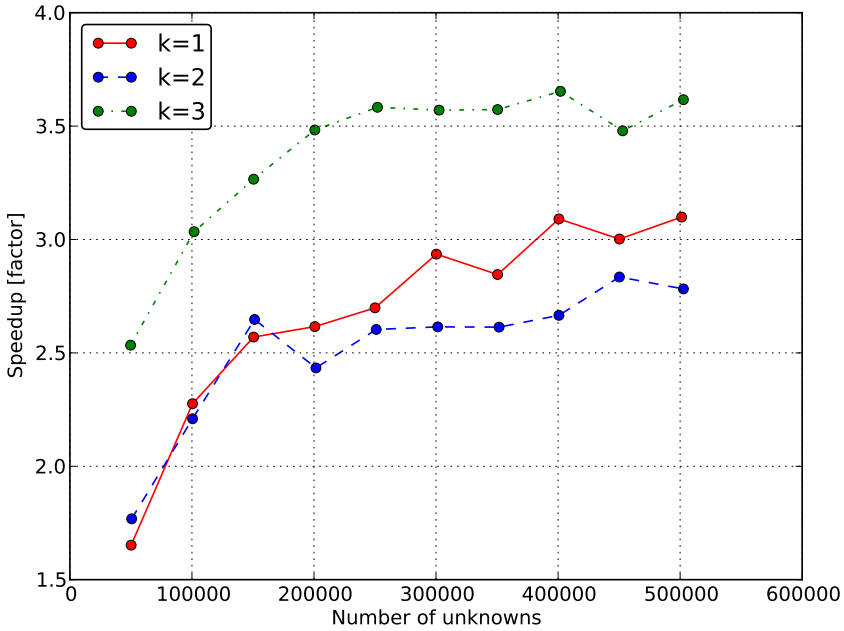


(b) Block Jacobi PC for CPU and Jacobi PC for GPU

Figure 9.1: Run times for linear solve of Poisson's equation on the unit square (2D).

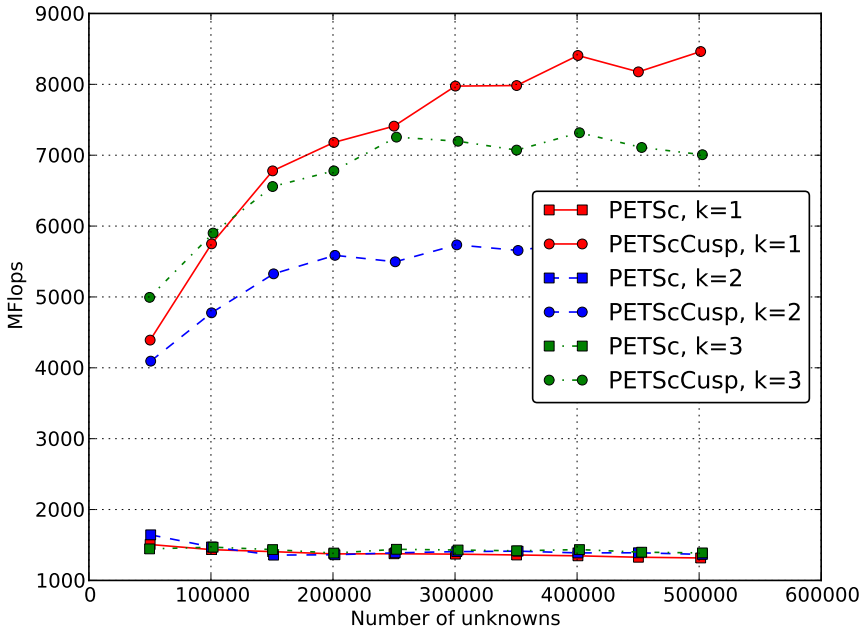


(a) Jacobi PC for both backends

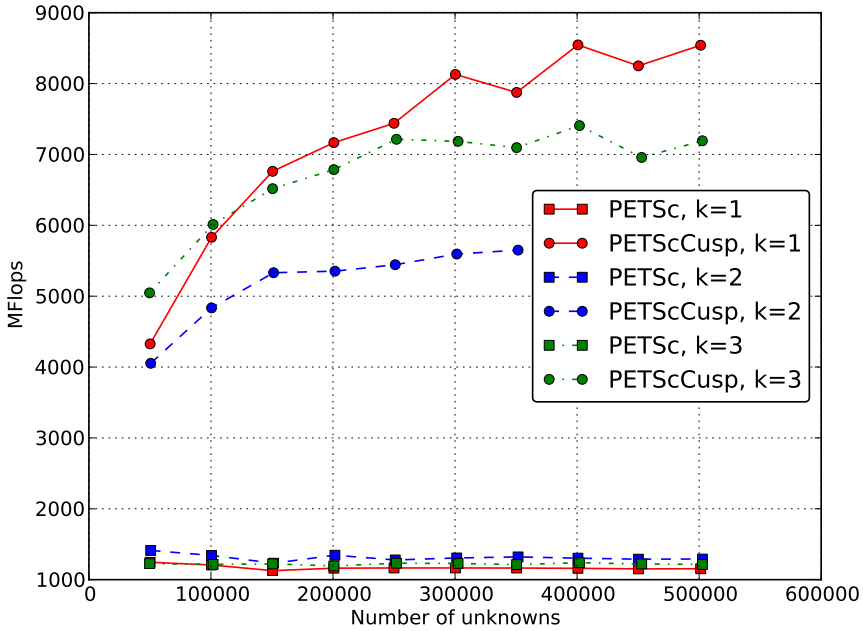


(b) Block Jacobi PC for CPU and Jacobi PC for GPU

Figure 9.2: Speedup for PETScCusp over PETSc for linear solve of Poisson's equation on the unit square (2D).



(a) Jacobi PC for both backends



(b) Block Jacobi PC for CPU and Jacobi PC for GPU

Figure 9.3: Flops for linear solve of Poisson's equation on the unit square (2D).

3D

For the three dimensional case, we will only look at numbers measured when using block Jacobi preconditioning for CPU and Jacobi preconditioning for GPU; that is, we compare the best-performing preconditioners for each linear algebra backend. Hence, we will obtain more conservative measurements of speedup than if we had used Jacobi preconditioning for both backends. We will use this approach for linear elasticity as well.

Regarding correctness, the l_2-norm of the differences between the solutions computed by the two backends are all of order 1×10^{-5}. Again, this is consistent with the relative Krylov solver tolerance. The fractions of time spent in the `solve` function for the PETSc backend, seen in Table 9.1, are much lower than in the 2D case for all values of k.

When studying the performance in the three dimensional case, we will discuss the results for all k at the same time rather than discussing them separately as in the 2D case. A plot of the run time can be seen in Figure 9.4. For each k, we see again PETScCusp is faster than PETSc. We also see that the runtime increases with k; that is, $k = 1$ is faster than $k = 2$, which in turn is faster than $k = 3$. A notable difference from the two dimensional case is that the run times are much lower overall. We also note that the curves look more linear than in the two dimensional case.

The speedups, seen in Figure 9.5, do not start low and increase with the system size as in the 2D case. Rather, they stay roughly constant, only varying with about ± 1. This nearly constant behavior is consistent with the fact that the run time curves are almost completely linear. When comparing the speedups with those of the block Jacobi case in 2D, shown in Figure 9.2b, we see two similarities. First, we note that the maximal speedup is once again attained for $k = 3$. Second, it looks as if $k = 1, 2$ are about to cross, such that the ordering of the curves becomes the same as in the 2D case. It might also be that the two curves are stabilizing at roughly the same value. A notable difference from the 2D case is that the curve for $k = 2$ is decreasing with the system size.

The flop rates are plotted in Figure 9.6 on page 90. For PETScCusp, they are in the range of 4000 to 7000 megaflops. Once again, the lowest flop rate is achieved for $k = 2$. The curve for $k = 3$ lies somewhat higher than $k = 1$ in this case, which is the opposite of what we saw in the two dimensional case. Another difference is that we do not see the steep growth in flop rates with system size that we saw in 2D. Also, it does not seem as if the flop rates for $k = 1$ continue to increase with the system size, as was the case in 2D.

9.2.3 Static, linear elasticity

The next linear systems we study are the ones arising from the discretization of the static, linear elasticity problem, as given in Eqs. (2.3.28) to (2.3.31). The two and three

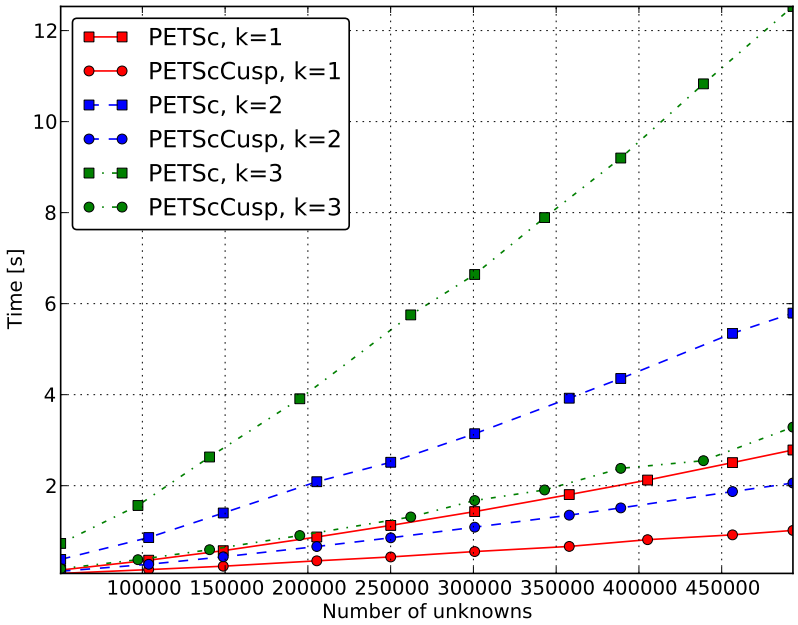


Figure 9.4: Run time for linear solve of Poisson's equation on the unit cube (3D).

dimensional cases are again considered separately.

2D

We verify correctness first. By taking the norm of the difference of the solutions computed by the two backends, we get results of order 1×10^{-7} to 1×10^{-8}. This is consistent with the relative Krylov solver tolerance. Table 9.1 shows that this is a relatively solver-heavy problem; that is, a significant amount of the run time is spent in the `solve` function.

The results for run time, speedup and flops can be seen in Figures 9.7, 9.8 and 9.9, respectively. The run times for linear solve follow the same general pattern as observed for Poisson's equation. PETScCusp is faster than PETSc for every system size. This is the case for all values of k. Also, the problems with lower k values run faster than the ones with higher k values. The run time depends roughly linearly on the system size. We observe that the run times are significantly higher in this case than for the two dimensional Poisson problem.

The speedups increase a little with the system size, but to a lesser extent that what we saw for 2D Poisson. We once again have the most speedup for $k = 3$, with the speedups for $k = 1, 2$ being roughly equal. The speedup factors are overall a little lower than for Poisson, with a maximum factor of 3 attained for $k = 3$.

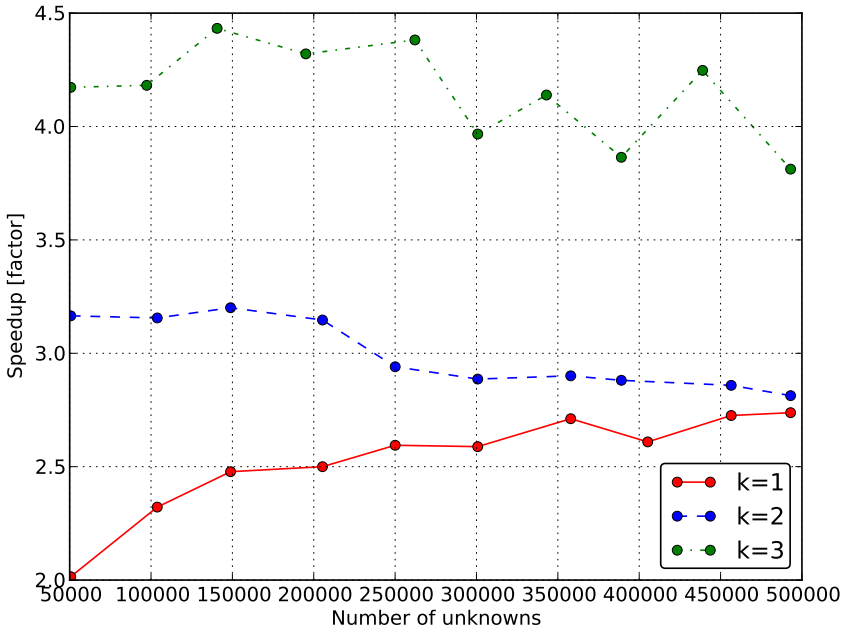


Figure 9.5: Speedup for PETScCusp of PETSc for linear solve of Poisson's equation on the unit cube (3D).

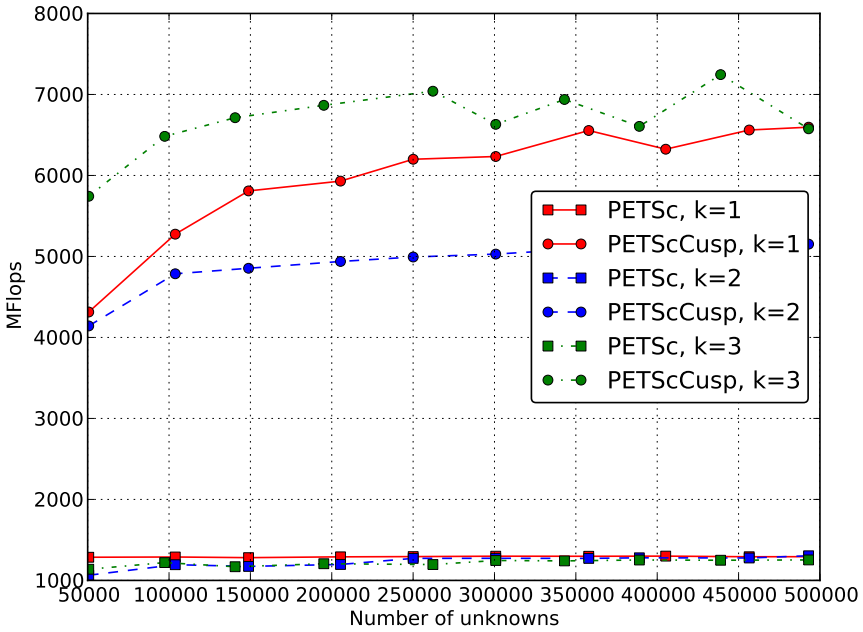


Figure 9.6: Flops for linear solve of Poisson's equation on the unit cube (3D).

Last, we look at the flop rates. PETScCusp yields flop rates much higher than PETSc for this problem as well. For PETScCusp, the values are in the range of 4000 to 7000. The curves look very similar to what we saw for Poisson's equation in both 2D and 3D.

3D

In three dimensions, the norm of the differences of the solutions are of order 1×10^{-5} to 1×10^{-6}. We can once again conclude that PETScCusp computes the correct solution. The fractions of run time spent in the solve step for PETSc, seen in Table 9.1, are significantly lower than in two dimensions. This is the same behavior as observed for Poisson's equation.

The run times, shown in Figure 9.10 on page 94, are again similar in nature to what we have seen in all the previous problems. PETScCusp is faster than PETSc for all system sizes for each value of k. The run time increases with k, and it depends linearly on the system size. We observe that the run time is lower than for the two dimensional elasticity problem. It is also noted that for $k = 3$, no data exists for systems of more than 350 000 unknowns. This is because the linear solve function breaks down for larger systems for this element degree.

The speedup numbers are shown in Figure 9.11 on page 95. They are fairly constant,

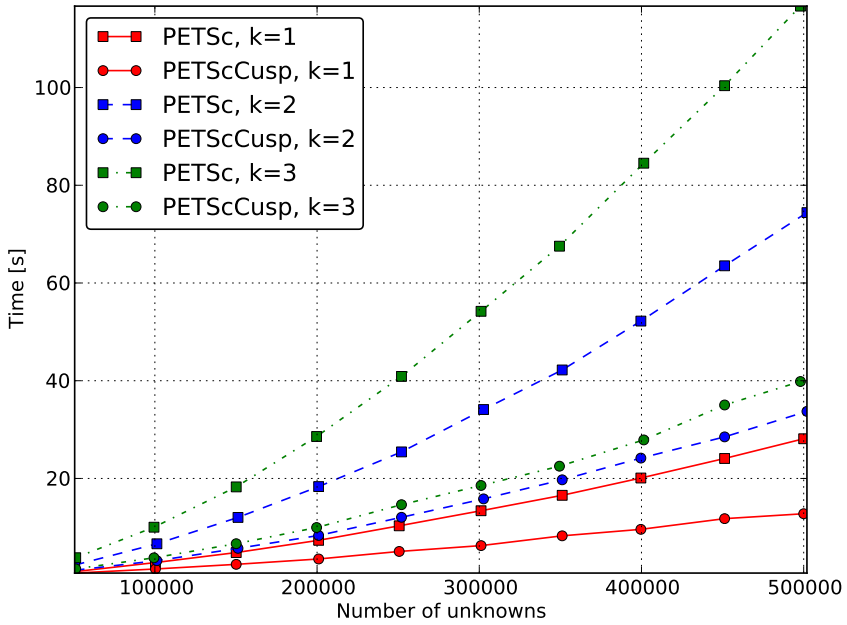


Figure 9.7: Run time for linear solve of the static, linear elasticity problem on the unit square (2D).

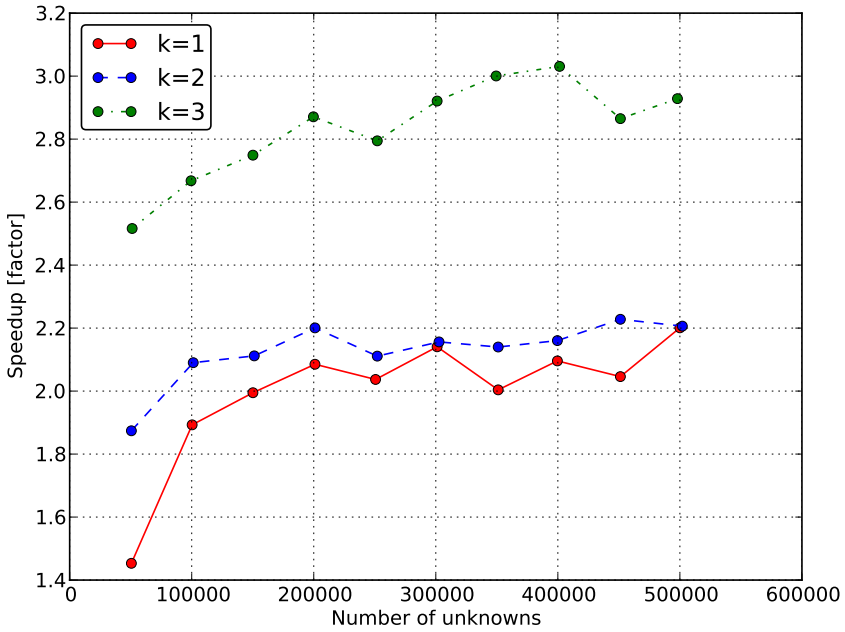


Figure 9.8: Speedup for PETScCusp of PETSc for linear solve of the static, linear elasticity problem on the unit square (2D).

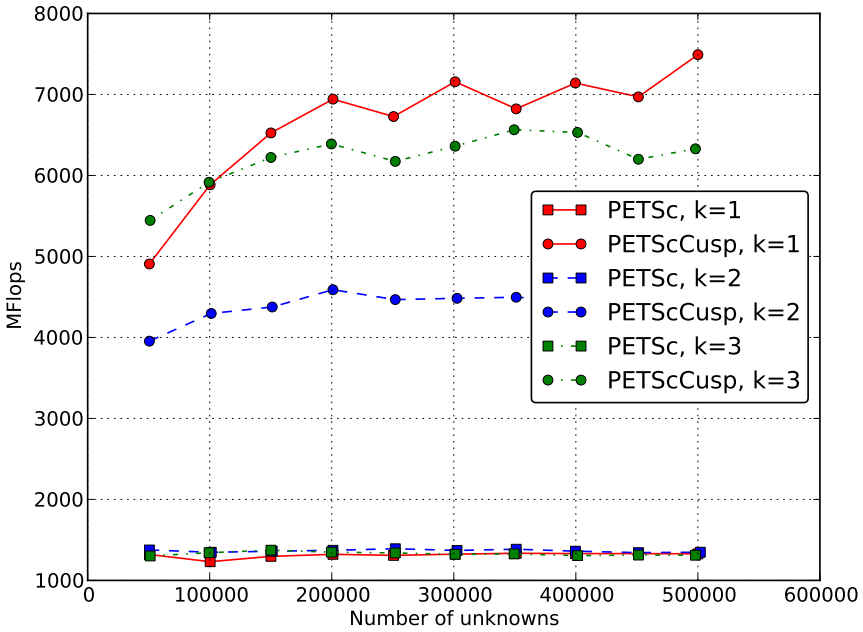


Figure 9.9: Flops for linear solve of the static, linear elasticity problem on the unit square (2D).

similar to what we had for Poisson's equation in 3D. One difference is that $k = 1, 2, 3$ all have different values, with none of them stabilizing around the same value. Also similar to Poisson's equation in 3D is that the curve for $k = 2$ is decreasing with system size. The speedup factors lie in the range of 2 to 4, which is also similar to the three dimensional Poisson problem.

Last, we look at the flop rates, seen in Figure 9.12 on page 96. The numbers lie in the range of 5000 to 8000 megaflops. We note that the lower limit is 1000 megaflops higher than what we have observed for all the other problems. We also see from the plot that the numbers decline slightly as the system size grows. This is also the first case where $k = 2$ does not yield the lowest flop rates. Here, it is instead $k = 1$ that yields the lowest rates.

9.2.4 Summary of results for Poisson's equation and linear elasticity

We summarize our observations:

Correctness

For all the problems, the l_2-norm of the difference of the solutions computed by the two backends is of order 1×10^{-5} or smaller. We conclude that the new, GPU-based implementation computes the correct solution for all the problems under study. To experiment, all the benchmarks were also run with a relative

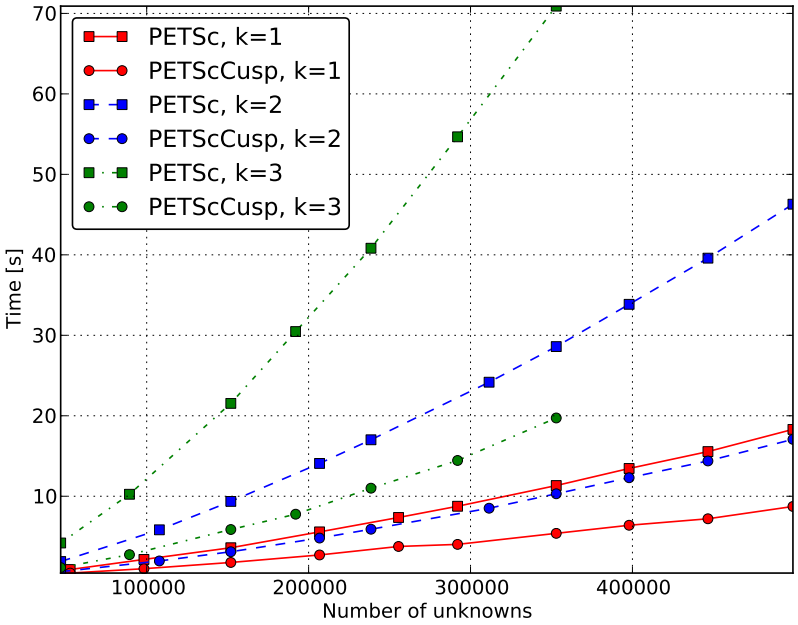


Figure 9.10: Run time for linear solve of the static, linear elasticity problem on the unit cube (3D).

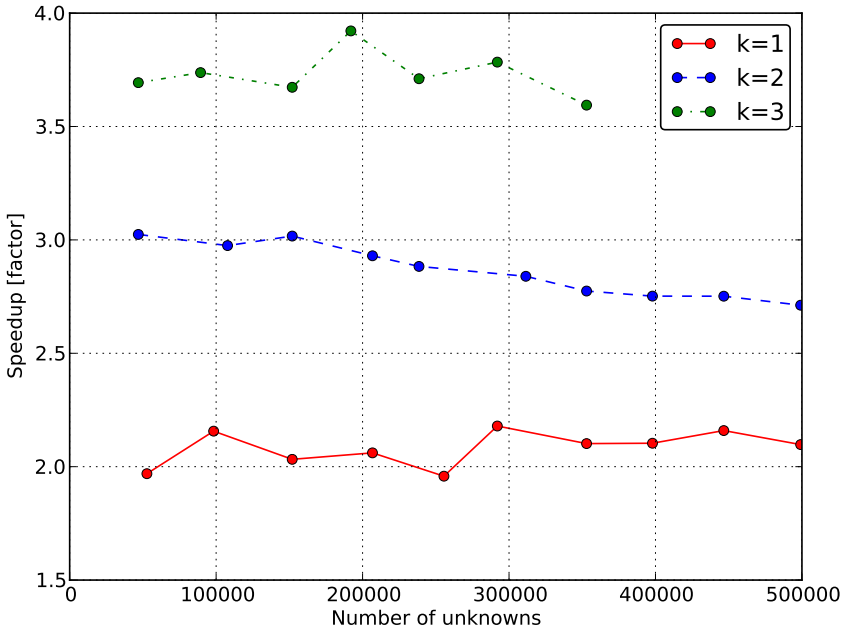


Figure 9.11: Speedup for PETScCusp of PETSc for linear solve of the static, linear elasticity problem on the unit cube (3D).

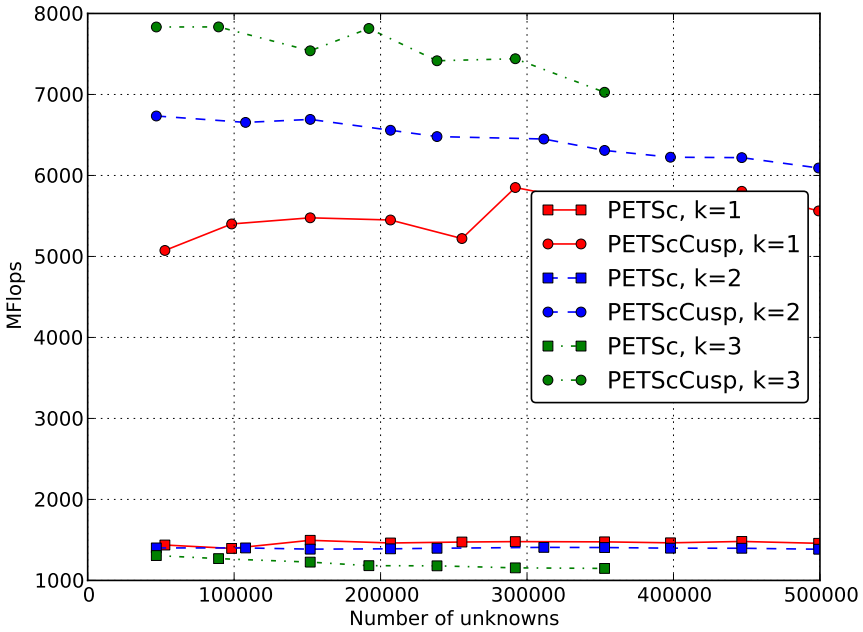


Figure 9.12: Flops for linear solve of the static, linear elasticity problem on the unit cube (3D).

Krylov solver tolerance of 1×10^{-12}. This decreased the norm of the error, and also led to more Krylov solver iterations and longer run times.

Run time

We have seen that PETScCusp runs faster than PETSc for all the problems. We have also observed that the lower order problems always run faster than the higher order problems. Also, the run time has increased linearly or slightly above linearly with the system size. For both equations, the three dimensional problems have run faster than the two dimensional ones. Also, the factor of time spent in the `solve` function has been lower for the three dimensional problems.

Speedup

We observe speedup factors of above 1 for all the problems, consistent with the observation that PETScCusp is faster than PETSc in all cases. In general, the three dimensional problems achieve somewhat higher speedup factors than the two dimensional ones. We also observe that the higher order problems achieve more speedup than the lower order ones. In all cases, the highest speedup is attained for $k = 3$. In all cases except the two dimensional Poisson problem, the lowest speedup is attained for $k = 1$.

Flops

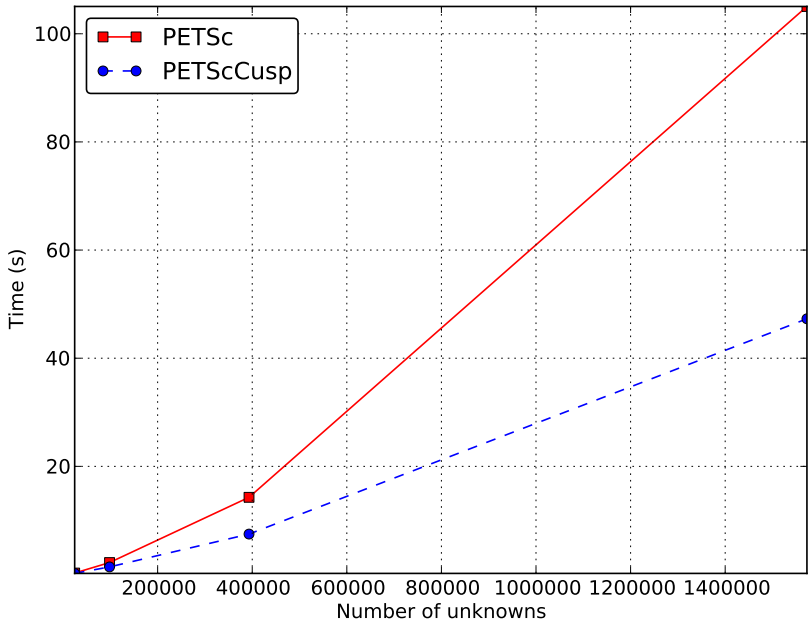


Figure 9.13: Run time for linear solve of the DG advection problem in 2D.

The flop rates achieved with PETScCusp are consistently several times higher than the rates achieved with PETSc. For PETScCusp, we have seen flop rates in the range from 4000 to 8500 megaflops. While PETSc has shown more or less constant rates for all problems and all mesh sizes, the rates for PETScCusp has differed with the problems and system sizes. The case $k = 2$ has the lowest flop rate in all cases except for the three dimensional elasticity problem.

9.2.5 DG advection

Now, the linear systems arising from discontinuous Galerkin discretization of pure advection (or simply "DG advection"), as given in Eqs. (2.3.32) to (2.3.35), will be solved. A benchmark is implemented based on a demo distributed with the DOLFIN source code. This demo reads a mesh and a velocity field from files. The velocity field is defined relative to the mesh. To increase the mesh size as in the previous problems, the mesh cannot simply be redefined to have larger resolution. Instead, a uniform mesh refinement is performed using the DOLFIN function `refine(mesh)`, which adds a new vertex per edge in the original mesh. The way the test problems are defined, one vertex in the mesh corresponds to one unknown in the linear system. Hence, the system size increases very rapidly after only a few refinements. Thus, it is only possible to perform three refinements before the system becomes too large to solve on

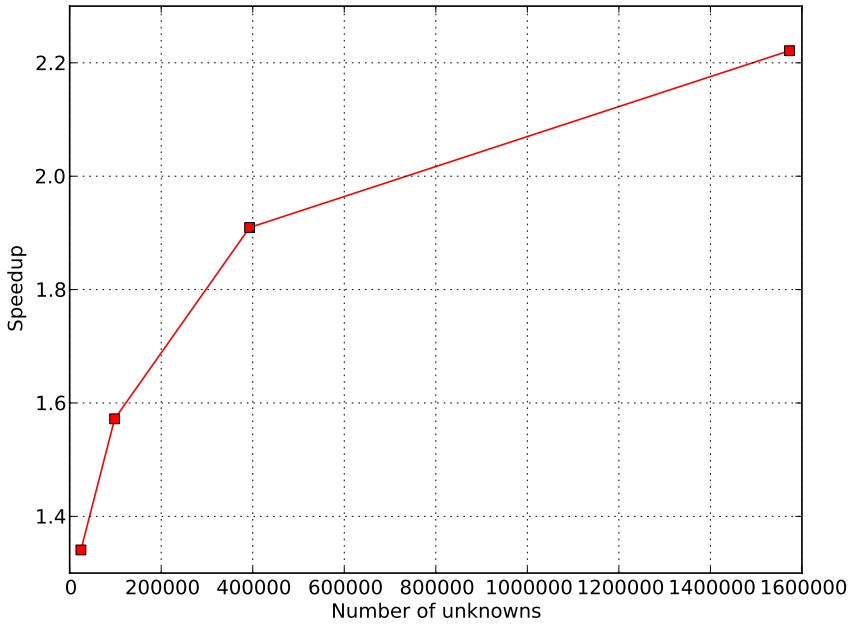


Figure 9.14: Speedup for PETScCusp over PETSc for linear solve of the DG advection problem in 2D.

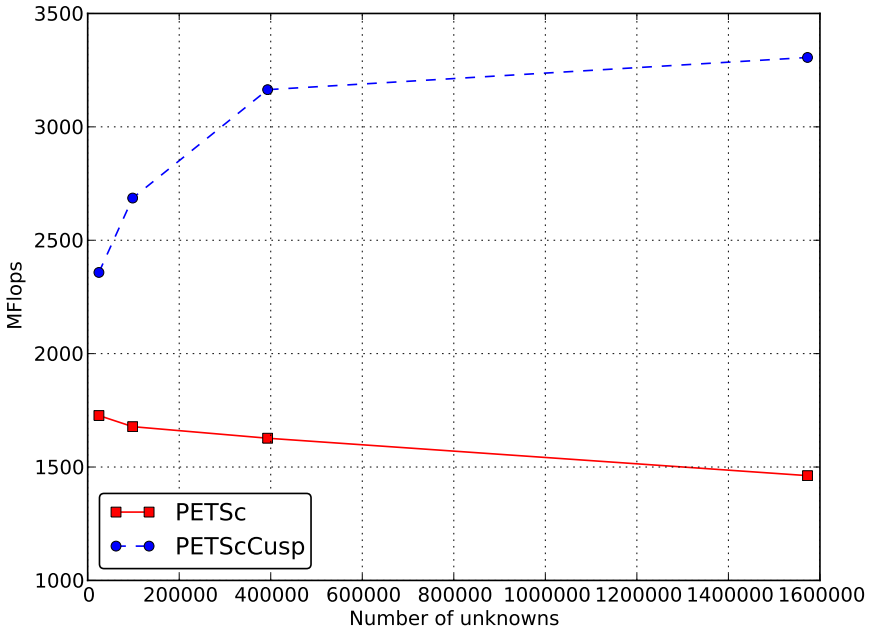


Figure 9.15: Flops for linear solve of the DG advection problem in 2D.

the test machine.

The three refinements lead to a total of four solves. The run times for these can be seen in Figure 9.13. The run time increases almost linearly with the number of unknowns, just as in the previously investigated cases. The run times becomes significantly higher than in the previous problems, but we also see that the system sizes are very much larger than what we have previously studied. Hence, the large run times are expected. For this problem, the solve step accounts for 30 to 40 percent of the total run time.

The speedups, seen in Figure 9.14 on the preceding page, also have the same characteristics as the previous problems. However, the speedup factors attained are lower than for the other problems. This is due to the fact that this problem is solved using block Jacobi preconditioning for both backends. For PETScCusp, Jacobi preconditioned solve did not produce correct results, so the slower block Jacobi preconditioner had to be used. Thus, the difference between run times for the two backends, and hence speedup, is lower for this problem than for the previous ones.

The flop rates for DG advection, plotted in Figure 9.15, are also much lower than for the other problems. For the largest system, only around 3250 megaflops is attained. There are two reasons for this, both of which have already been touched upon earlier: the flop rates reported for block Jacobi preconditioned solve are less than for Jacobi, since the rates cited from PETSc are only for the solve itself, not including all

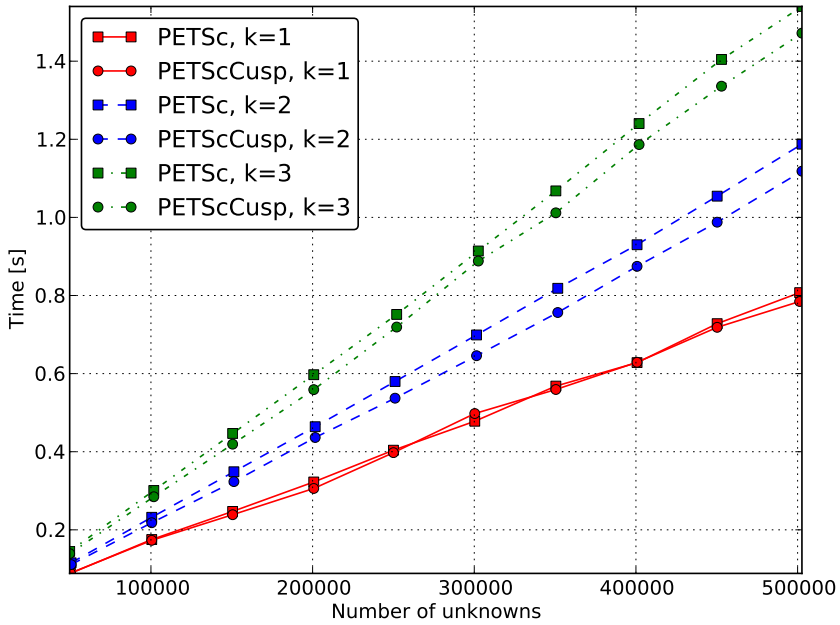


Figure 9.16: Run times for assembly of Poisson on the unit square (2D).

computations related to preconditioning. In addition, the block Jacobi actually runs slower on the GPU, which also indicate lower flop rates, probably due to poor memory bandwidth utilization for this preconditioner on the GPU.

9.3 Assembly

For assembly, the run time and speedup for Poisson's equation is presented. First, the two dimensional case is considered. The run times and speedup for assembling the linear systems can be seen in Figures 9.16 and 9.17, respectively. For $k = 2, 3$, PETScCusp runs faster than PETSc for all system sizes. As seen in the plot, the difference in run time is not very large, corresponding to speedup factors between 1.03 and 1.08. In other words, the speedup is below 10 percent in all cases. For $k = 1$, the results are less stable. PETSc and PETScCusp alternate between being the fastest for the various system sizes. As a result, we see speedup factors both above and below 1.

The run times for the three dimensional case can be seen in Figure 9.18 on page 103. The results are more stable in this case, with PETScCusp being consistently faster than PETSc. The speedups, seen in Figure 9.19 on page 103, lie in the range of 1.13 to 1.25. This is consistent with the estimates that was calculated in Section 8.1. The most speedup is attained for $k = 2$, and the least is attained for $k = 1$. In the three

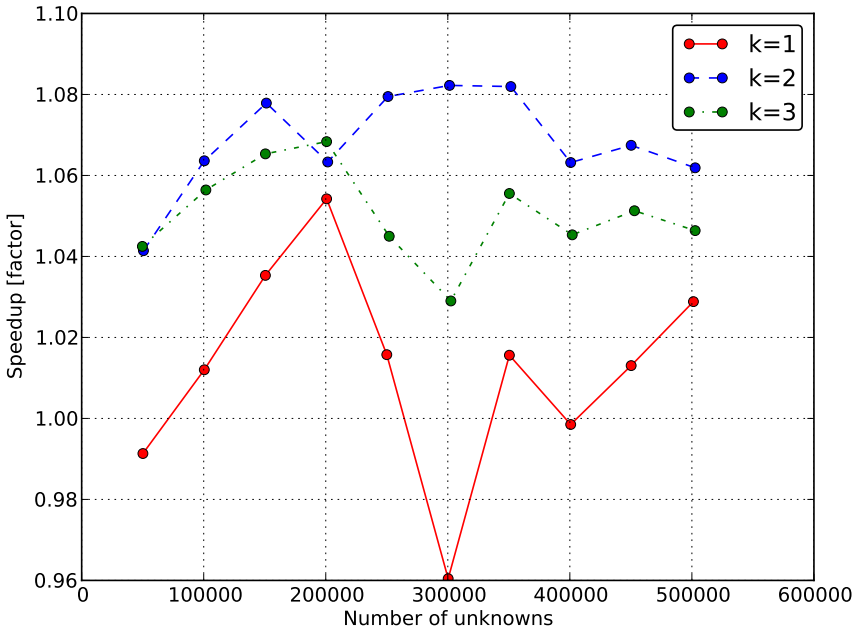


Figure 9.17: Speedup for assembly of Poisson on the unit square (2D).

dimensional case, however, a major weakness of the method becomes evident: the batch assembly function is unable to assemble large systems. A seen in Figures 9.18 and 9.19, the largest systems that the method is able to assemble in the 3D case are systems with around 2.5×10^5 unknowns. For three dimensional systems larger than this, the `MatSetValuesBatch` function fails and throws the memory-related exception `std::bad_alloc`. The reason it is able to assemble larger systems in 2D than in 3D is probably due to the fact that the 3D systems on average have more nonzero values, requiring more memory for storing a matrix with the same dimensions.

However, the standard assembly algorithm is able to assemble larger systems. This was seen when benchmarking the solution of larger systems in the previous section, where all systems were assembled using the standard assembly algorithm. The exact reasons for why the batch assembly function has these size limitations are not known. However, it is certain that the limitations are related to the handling and allocation of memory in the functions that copy matrix values from the host to the GPU.

In conclusion, three shortcomings of the batch assembly routine have been identified:

Unstable performance

The speedup factors seen are in the expected range, similar to what was estimated using Amdahl's law in Section 8.1. They are, however, pretty unstable, sometimes going below 1 in the two dimensional case.

Unable to assemble large systems

Systems with a certain amount of nonzero values are not possible to assemble due to memory constraints. This is a serious problem, as much of the point with a faster assembly routine vanishes when it is unable to assemble larger systems. As seen from the run time plots in Figures 9.16 and 9.18, the total run times for assembly are only a few seconds. Hence, one must have rather long running assembly function calls in order for the time saved to have any significance.

The local element tensor computation happens on the CPU

In the current implementation, the computation of the local element tensors is not implemented on the GPU. This is intentional, and is not really an observed weakness as the ones described above. However, it would be an advantage for a GPU-based assembly implementation to be completely implemented on the GPU. As seen from the study of related work in Chapter 6, much study has been undertaken into computing the local element tensors on the GPU.

Bringing this preliminary integration of the `MatSetValuesBatch` function up to a level of production-ready quality would take a significant effort. First of all, support for facet integrals would have to be implemented. Furthermore, the implementation introduces extra complexity to the linear algebra layer, as seen in the class diagrams presented in Chapter 8. Thorough testing for all cases would have to be conducted, and the resulting implementation would have to be maintained with future versions of DOLFIN. In light of the mediocre performance and weaknesses of the method, this was not considered to be worthwhile at this time. Instead, it is proposed that a design is made for a more stable method that also includes computation of the local element tensors on the GPU. This is discussed further in the next chapter.

9.4 A nonlinear example: Hyperelasticity

In this last example, it is demonstrated that the implementation can solve nonlinear problems correctly and efficiently. Nonlinear problems in DOLFIN are solved using another version of the `solve` function that repeatedly assembles and solves the system.[5] This version of the `solve` function takes a definition of a nonlinear variational problem as input, rather than a linear system. A run time measurement of this function would include both repeated assembles and solves. As we just saw, the assembly step did not run very efficiently on the GPU. However, the solve step runs efficiently on the GPU. Hence, the nonlinear `solve` function should perform somewhat better on the GPU compared to the CPU.

The chosen nonlinear problem is the hyperelasticity problem presented in Section 2.3.2. It was solved on a mesh of an *atrium*, which is one of the chambers of a heart. It

5 See the section on nonlinear problems in the FEniCS tutorial: `http://fenicsproject.org/documentation/tutorial/nonlinear.html`.

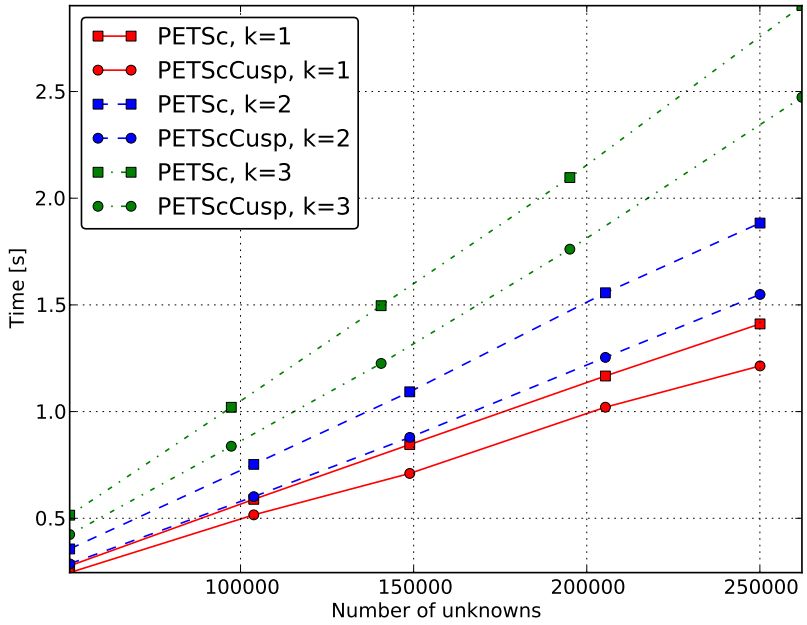


Figure 9.18: Run times for assembly of Poisson on the unit cube (3D)

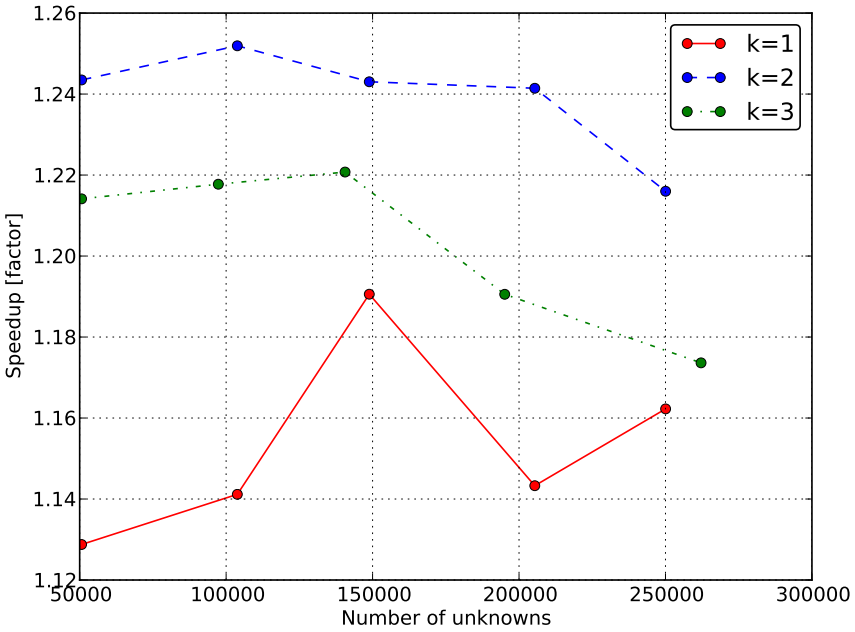


Figure 9.19: Speedup for assembly of Poisson on the unit cube (3D)

	Run time	Flops
PETSc	6.90 s	1677
PETScCusp	4.61 s	2500

Table 9.2: Performance statistics for the hyperelasticity problem.

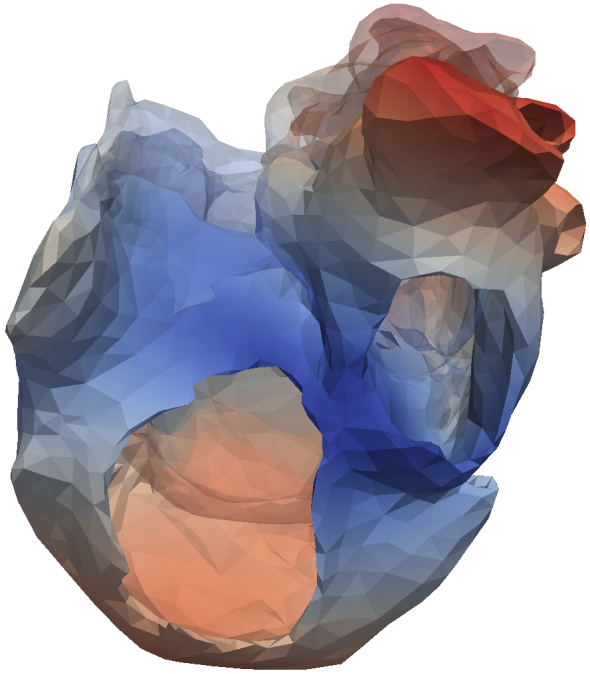


Figure 9.20: Plot of the deformed atrium. The transparent part of the figure shows the original shape of the atrium, and the solid-colored part shows the shape after deformation.

was solved using both linear algebra backends, generating the data seen in Table 9.2. These run times correspond to a total speedup factor of 1.5 for PETScCusp over PETSc.

The flop rate for PETScCusp is only 2500, which is much less than what was observed for the previous problems. This relatively low rate probably stems from the fact that the mesh, and hence the linear system, is quite small. Hence, the startup costs related to solving the system (including memory transfer) dominate, such that a high flop rate is not achieved.

A plot of the deformed atrium can be seen in Figure 9.20. The l_2-norm of the difference of the solutions computed by the two backends is 2.18×10^{-12}. Therefore, we conclude that the GPU-based linear algebra backend computes the correct solution for this problem.

9.5 Summary of benchmarks

This chapter demonstrates that the new implementation efficiently and accurately solves the linear systems arising from the finite element method discretization of a

wide range of test problems. For certain choices of Krylov method and preconditioner for certain problems, the GPU-based implementation fails to compute the correct solution. This behavior was seen for the GMRES method and Jacobi preconditioner for the DG advection problem. Therefore, one must always carefully verify the correctness of the computed results.

Given that a suitable pair of Krylov method and preconditioner is chosen, the GPU-based linear algebra backend is able to correctly solve all the linear systems faster than the CPU-based linear algebra backend. The performance varies, with speedup factors ranging from 1.5 to 6, largely depending on the structure and size of the linear system.

Throughout the performance evaluation, some interesting behavior is seen. This demands some further explanation:

Poisson's is solved much faster than the elasticity problem

Poisson's equation is solved much faster than the linear elasticity problem for systems of equal size. This is the case when comparing both the two dimensional cases and the three dimensional cases of the two problems. To find the reason for this, we consider two systems of roughly equal size corresponding to the 2D cases of the two equations.

Studying a linear system for Poisson's equation with 50176 unknowns, we find that it contains roughly 0.015% nonzero values. A linear system for the elasticity problem with 50562 unknowns, on the other hand, contains about 0.023% nonzero values. Hence, the elasticity system has over twice as many nonzero values as the system for Poisson's equation.

This increased matrix density leads to a more costly computation of the matrix-vector product due to the larger amount of floating point operations performed. In addition to this, the number of Krylov solver iterations needed to converge is much lower for Poisson's equation than for the elasticity problem. In the present example, the solver converges after 664 iterations for Poisson's equation and after 1516 iterations for the elasticity problem. To find the reason why more iterations are needed for the elasticity problem, the properties of the Krylov method and preconditioner have to be studied together with the structures and properties of the matrices.

In summary, the higher number of iterations as well as the more costly matrix-vector product leads to a larger run time for the elasticity problem.

3D problems run much faster than 2D problems

If we again compare matrices of similar size arising from different equations, we observe that the 3D problems in general gives denser matrices than the 2D problems. This leads to a more costly matrix-vector product in the 3D case.

However, experiments show that the 3D problems in general require fewer iterations in the Krylov solver to converge than the 2D problems. This is related to

the size of the meshes and the condition number of the matrices. For a given linear system size, the mesh size h of a 3D problem is larger than the mesh size for the corresponding 2D problem. The condition number of the corresponding matrices can be expected to grow as h decreases. Since the number of iterations in the Conjugate Gradient method scales with the condition number [39, p. 137], the convergence is faster for the 3D problems than for the 2D problems when comparing linear systems of equal size.

This leads to faster overall run times for the linear solve step in 3D than in 2D, even though each iteration is more costly in 3D.

Very low percentage of run time spent in solve Poisson 3D, $k = 1$

For 3D problems in general, mush of the run time is spent computing mesh entities and mesh connectivity. Matrix assembly is also time consuming. Hence, the setup and assembly stages are slower in 3D than in 2D.

However, as we just saw, the solve stage is generally faster in 3D than in 2D. As a result, the percentage of run time spent in solve is much lower in 3D than in 2D. This becomes particularly evident in the first order case of Poisson's equation in 3D, but it is seen for all the other 3D problems as well, see Table 9.1.

Elasticity 3D breaks down for larger systems

Discretizing the three dimensional elasticity problem on larger meshes leads to systems that are not possible to solve. This is simply due to the fact that the systems contain so many nonzero values that the memory requirements for solving them become too high. The `solve` function fails with an `std::bad_alloc` exception; it is not possible to allocate the necessary amount of memory on the GPU.

The flop rate for elasticity 3D declines with system size

Whenever the flop rate of a program is below the peak flop rate of the GPU, the program is memory bandwidth limited (see Section 3.2.4). Hence, when the flop rate declines as in this case, it is because the memory bus utilization is declining.

The exact reason for this decline in memory bus utilization is not known, but a possible reason is that the algorithm loads the data in a less efficient way when the size of the system increases.

The lowest flop rate is achieved for $k = 2$ in three of four cases

The reason for this is not known at this time. A further study of the sparsity pattern and block structure of the matrices, as well as a study of the properties of the Krylov methods and preconditioners used is needed in order to reach a full understanding of this behavior.

The results computed on the CPU and the GPU differ

From what we have seen so far, we might believe that the results from the CPU

and the GPU should be identical. In particular, as we saw in Section 7.4.3, the same host-based Krylov solver algorithm is used in PETSc for both CPU- and GPU-based matrices and vectors. The only difference is whether the vector operations and sparse matrix-vector product are computed on the CPU or the GPU.

However, as seen in Section 3.3.1, the floating point computations on the CPU and GPU may in fact be performed differently, possibly yielding somewhat higher accuracy on the GPU. In addition, performing floating point operations in parallel may give different results than performing the same operations in serial, due to the fact that floating point addition is not commutative [10, Ch. 7].

Together, these two reasons lead to slightly different results being computed on the GPU compared to the CPU.

10 Summary and conclusions

10.1 Summary

A library-based approach to accelerate the FEniCS Project using GPUs has been presented. By using libraries, the computational power of modern graphics processors has been leveraged with a moderate programming effort. The observed performance improvements are significant. One alternative to using libraries is hand-writing an implementation in CUDA C or another GPU programming language. Several advantages of the library approach have been emphasized:

- The resulting implementation is easier to maintain.
- The implementation benefits from improved hardware and updates to the library code without any changes to the actual software code. In other words, some work is outsourced to the developers of the external library.
- The user interface can remain clean and transparent, with the logic of the GPU acceleration hidden from the user.

10.2 Contributions

The solution of linear systems of equations is a fundamental step in solving (partial) differential equations by the finite element method. The focus of this work has been to solve the linear systems using GPUs. Specifically, GPU-based data structures for matrices and vectors from the PETSc library are integrated into DOLFIN, the main user interface of the FEniCS Project. These data structures facilitate faster, GPU-based computations of vector operations and the sparse matrix-vector product. These operations, in particular the sparse matrix-vector product, are important building blocks of Krylov methods. Therefore, the whole solution process runs faster when these operations are accelerated.

Speedup factors of up to over 6× has been observed for the linear solve step. The flop rates attained on the GPU have been up to over 6 times higher than those attained on the CPU.

It has been demonstrated that the new implementation is able to correctly and efficiently solve a wide range of problems. For certain problems, however, it may fail to compute the correct solution for a given pair of Krylov method and preconditioner.

The Krylov method and preconditioner must therefore be carefully selected, and the computed result must always be verified.

An attempt was also made at performing one part of the assembly of the linear systems on the GPU. Namely, a function for batch insertion of local element matrices was integrated. This attempt yielded moderate performance improvements, in the same ranges as what was estimated using Amdahl's law. However, the performance was unstable, and the routine broke down for larger systems. This approach was therefore not pursued further.

The build system for the FEniCS Project, named Dorsal, has been extended to automatically build all necessary dependencies with GPU support. This simplifies distribution and compilation for users of the FEniCS Project.

The contributions to both DOLFIN and Dorsal will be included in upcoming releases.

10.3 Conclusions

I conclude that using libraries is a viable approach to introduce GPU-accelerated computing to an existing software system. Even if the system is not designed with massively parallel computations in mind, parts of the system may still be amenable for outsourcing to a GPU-accelerated library.

To get further performance improvements, a specialized implementation using GPU computing programming languages may be necessary. This will most probably involve more dramatic changes to the source code, with redesigns of parts of the system. Through this project, we have seen that the assembly algorithm is one example of a computation that cannot simply be accelerated by outsourcing to external libraries. To implement this algorithm using GPUs, it must be redesigned to fit the programming paradigm of GPUs.

Developers will always have to assess whether the translation of a code to a new architecture is an appropriate approach to increase the performance. Porting a software project to use GPUs may be a significant undertaking, and is not necessarily always the right path to take. Using libraries is thus a golden mean; you can get clean code written in a familiar programming language and programming paradigm, and at the same time get some of the performance benefits that GPU computing can provide.

10.4 Limitations and future work

The implementation is subject to some limitations, some of which are fitting candidates for future projects.

Distributed GPU vectors and matrices

The relatively low amount of memory on a GPU limits the size of the problems it

can solve. Therefore, it is of interest to be able to distribute matrices and vectors over clusters of GPU-enabled compute nodes. This is not supported yet due to limitations in PETSc. This could be implemented either by adding the necessary extensions to PETSc, or by rewriting DOLFIN's wrapper layer for PETSc vectors.

GPU-based assembly
Assembling linear systems directly on the GPU is not supported. It is therefore proposed that a complete design is made for an assembly routine that performs both computation and insertion of local element matrices directly on the GPU. This would entail parallel processing of cell tensors for several cells at once, possibly yielding significant speedups. A starting point for such a project could be the previous work undertaken in this area, described in Chapter 6.

Add support for OpenCL
The provided implementation has the limitation that it can only be used with NVIDIA GPUs. Implementing support for platform-independent GPU computing APIs, such as OpenCL, would therefore make the implementation accessible to more users. To this end, one could integrate a linear algebra backend that utilizes OpenCL as the interface to the GPU, such as ViennaCL [81].

In addition, the following topics are possible paths for future work:

Adding support for more Krylov methods and preconditioners
The implementation is limited to the Krylov methods and preconditioners provided by PETSc. In particular, it is not possible to use user-specified preconditioning matrices on the GPU. Integrating more functionality in this direction would be beneficial.

Integration with other GPU-accelerated linear algebra libraries
It would be of interest to integrate other GPU-accelerated linear algebra libraries, to see if it is possible to achieve better performance than what has been observed with PETSc. Some of the libraries surveyed in Chapter 7 could be possible candidates. In addition, more GPU-accelerated libraries will probably be available in the future.

GPU-accelerating other parts of DOLFIN
Inspired by the work by Rathgeber [48], one could investigate the possibility of GPU-accelerating other parts of DOLFIN, possibly by re-implementing parts of DOLFIN in CUDA C. Special care must then be taken with regard to completeness and to make the backend changes transparent to the user.

Further streamlining of distribution and compilation
The possibility of providing pre-built binaries with GPU-support could be investigated, to further ease the installation for novice users.

Appendix A

CUDA installation guidelines

This appendix gives some rough guidelines and hints for installing CUDA under Ubuntu. While NVIDIA provides documentation for how to do this, one often encounters problems that are not covered in the official documentation. This appendix goes through some common problems that I have encountered when installing CUDA on various machines.

A.1 Basic guidelines

The general guidelines for installation are as follows:

1. Download the NVIDIA developer drivers, CUDA toolkit and GPU computing SDK from `http://developer.nvidia.com/cuda-downloads`.
2. Download the Getting Started Guide for your platform, found at NVIDIA's web pages: `http://developer.nvidia.com/nvidia-gpu-computing-documentation`. Follow the installation instructions, step by step. Remember to verify your installation by compiling and running some of the example code provided with the *GPU Computing SDK* [66].

A.2 Developer drivers vs. drivers from the package manager

Under Ubuntu, NVIDIA drivers are provided via the package `nvidia-current`. Although NVIDIA tells you to download and install the so-called "Developer drivers" from their website, this is often not necessary. Installing the `nvidia-current` package is often sufficient.

To install the drivers this way, you should first add an unstable repository to your list of repositories, to get the latest version of the drivers. This is especially important if your Ubuntu version is not the most recent. This is done by running the following line in a terminal:

Bash code

```
sudo add-apt-repository ppa:ubuntu-x-swat/x-updates
```

To install the drivers, run

Bash code

```
sudo apt-get update
sudo apt-get install nvidia-current
```

By installing the drivers this way, one may encounter a problem when compiling the sample code from the SDK. The compiler may complain that it "`cannot find -lcuda`". This is because the package manager installs the drivers to a different directory than where the compiler expects to find them. To fix this, add symbolic links to point to the location of the drivers:

Bash code

```
    sudo ln -s /usr/lib/nvidia-current/libcuda.so
       /usr/lib/libcuda.so
    sudo ln -s /usr/lib/nvidia-current/libcuda.so.1
       /usr/lib/libcuda.so.1
```

A.3 Hybrid setups

Some recent laptops are built with two GPUs; one on-board chip as well as a discrete NVIDIA GPU. This is not supported out of the box under Ubuntu, but is supported through a third-party tool called *Bumblebee*. The tool provides a terminal command called `optirun`. This command explicitly enables the NVIDIA GPU for programs that need it. Hence, when running CUDA programs via the command line on a laptop with a hybrid hardware configuration, they should always be run through `optirun`:

Bash code

```
optirun ./your-cuda-program
```

To install Bumblebee, run the following:

Bash code

```
sudo add-apt-repository ppa:bumblebee/stable
sudo apt-get update
sudo apt-get install bumblebee bumblebee-nvidia
sudo usermod -a -G bumblebee $USER
```

Reboot your computer after these steps. After installing, make sure that all your installed packages are up to date. Be aware that the `bumblebee-nvidia` package depends on `nvidia-current`, so it will automatically be installed as well. Hence, it may cause trouble if you have also installed the NVIDIA developer drivers manually. It is recommended to always let the package manager handle the installation of drivers when using Bumblebee. Remember to add the symbolic links to the libraries as described in the previous section.

To verify the Bumblebee installation, run

Bash code

```
glxspheres
optirun glxspheres
```

The first line should run a graphics demo rendered with the on-chip GPU, while the second runs the graphics demo rendered with the NVIDIA GPU. You can verify this by looking at the terminal output the commands give.

A.4 Install CUDA 4.1 under Ubuntu 11.04 on a hybrid setup

This section provides a step-by-step guide for installing CUDA 4.1 under Ubuntu 11.04 on a laptop with a hybrid setup. The procedure will be similar for other Ubuntu and CUDA versions, but you may encounter errors not covered here. If you do, Google is your friend.

Start by adding the unstable graphics driver repositories:

Bash code

```
sudo add-apt-repository ppa:ubuntu-x-swat/x-updates
sudo apt-get update
```

Then, install Bumblebee as described in Section A.3. You do not need to explicitly install the NVIDIA drivers as they are installed by Bumbleblee. After installing Bumblebee and rebooting, install some packages needed for running CUDA and OpenGL programs:

Bash code

```
sudo apt-get install freeglut3-dev build-essential libx11-dev
    libxmu-dev libxi-dev libgl1-mesa-glx libglu1-mesa
    libglu1-mesa-dev
```

Next, download the CUDA Toolkit and GPU Computing SDK from NVIDIA's webpages: `http://developer.nvidia.com/cuda-toolkit-41`. Install by doing the following:

- Run `sh cudatoolkit_***.run` to install the CUDA toolkit. If you want to install to the default location, which is `/usr/local`, the command must be run as root. If you install to a non-default location, you must set the environment variable `CUDA_INSTALL_PATH` after installing to point to your installation directory.
- Install the SDK by running `sh gpucomputingsdk_***.run`.

The installation is verified by the following commands:

Bash code

```
cd <path-to-sdk-install>/C
make
cd bin/linux/release
./deviceQuery
optirun ./particles
```

The latter two commands are for running some of the example code. See the *CUDA Getting Started Guide for Linux* [45] for examples of correct output from the `deviceQuery` program.

Appendix B

Complete PETSc wrapper layer diagram

A complete diagram of the PETSc wrapper layer in DOLFIN's linear algebra submodule can be seen in Figure B.1 on the following page. This diagram does not include the extra class introduced as part of the batch assembly implementation discussed in Section 8.4.

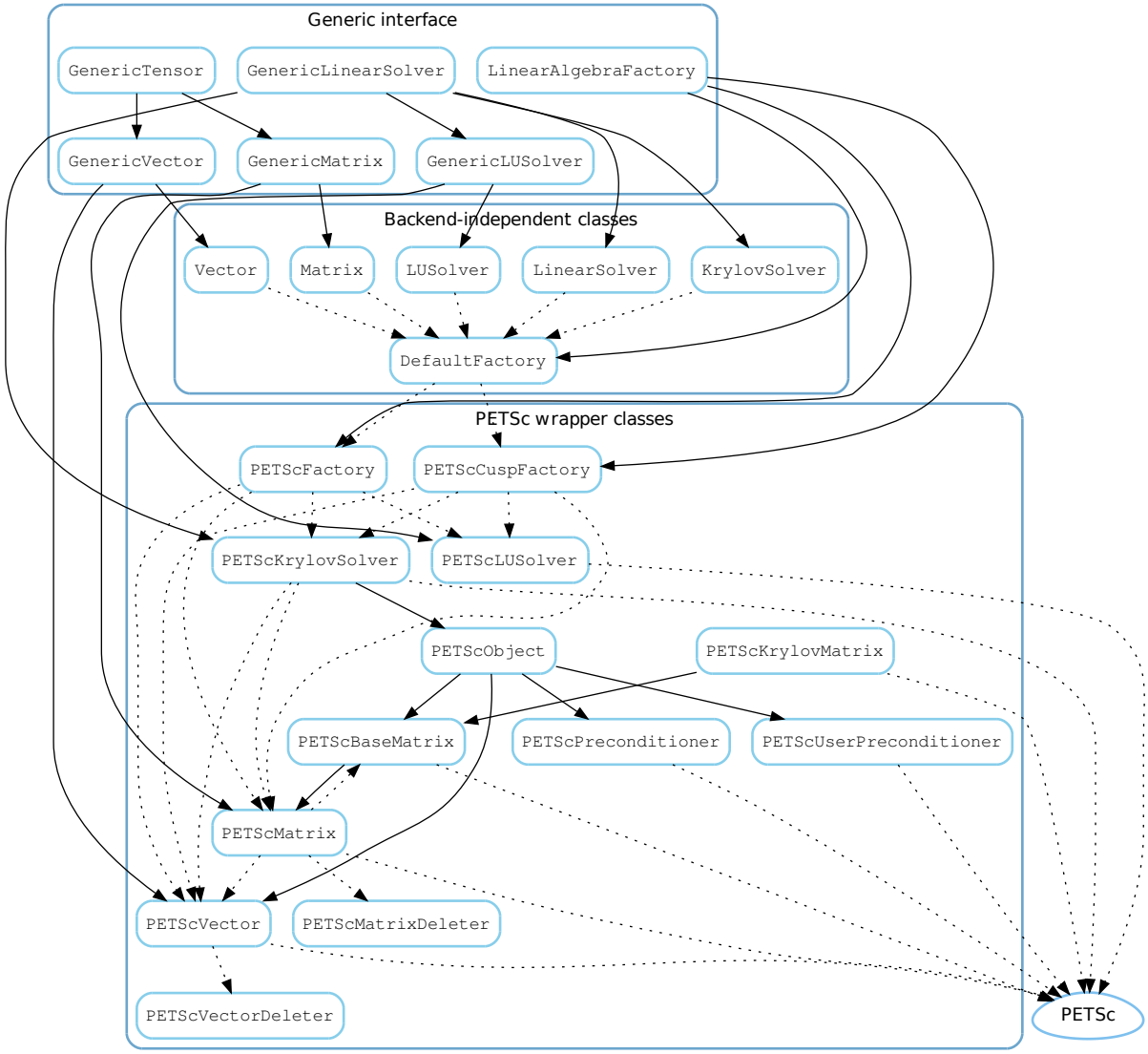


Figure B.1: The PETSc wrappers.

Bibliography

Books and book chapters

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering*. 3rd ed. A. K. Peters, Ltd., 2008.

[2] Martin Sandve Alnæs. "UFL: a finite element form language". In: *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012.

[3] Martin Sandve Alnæs, Anders Logg, and Kent-Andre Mardal. "UFC: a finite element code generation interface". In: *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012.

[4] Susanne C. Brenner and L. Ridgway Scott. *The Mathemathical Theory of Finite Element Methods*. 3rd ed. Springer, 2008.

[5] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[6] Kenneth Eriksson et al. *Computational Differential Equations*. Studentlitteratur, 1996.

[7] Rafael C. Gonzales and Richard E. Woods. *Digital Image Processing*. 3rd ed. Pearson Education, Inc., 2008.

[8] Robert C. Kirby. "FIAT: numerical construction of finite element basis functions". In: *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012.

[9] Robert C. Kirby and Anders Logg. "The finite element method". In: *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012.

[10] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann, 2010.

[11] Hans Petter Langtangen. *Computational Partial Differential Equations. Numerical Methods and Diffpack Programming*. 2nd ed. Springer, 2003.

[12] Anders Logg, Kent-Andre Mardal, and Garth N. Wells, eds. *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012. URL: `http://fenicsproject.org/book/`.

[13] Anders Logg, Kent-Andre Mardal, and Garth N. Wells. "Finite element assembly". In: *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012.

[14] Anders Logg, Garth N. Wells, and Johan Hake. "DOLFIN: a C++/Python finite element library". In: *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012.

[15] Anders Logg et al. "FFC: the FEniCS form compiler". In: *Automated Solution of Differential Equations by the Finite Element Method. The FEniCS Book*. Springer, 2012.

[16] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. SIAM, 2003.

[18] Jason Sanders and Edward Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

Journal papers

[19] Martin Sandve Alnæs et al. "Unified Framework for Finite Element Assembly". In: *International Journal of Computational Science and Engineering* 4.4 (Nov. 2009), pp. 231–244.

[20] Luc Buatois, Guillaume Caumon, and Bruno Lévy. "Concurrent number cruncher: a GPU implementation of a general sparse linear solver". In: *International Journal of Parallel, Emergent and Distributed Systems* 24.3 (2009), pp. 205–223.

[21] Magnus R. Hestenes and Eduard Stiefel. "Methods of Conjugate Gradients for Solving Linear Systems". In: *Journal of Research of the National Bureau of Standards* 49.6 (1952), pp. 409–436.

[22] Ilse C. F. Ipsen and Carl D. Meyer. "The Idea Behind Krylov Methods". In: *American Mathematical Monthly* 105.10 (1998), pp. 889–899.

[23] Robert C. Kirby. "Algorithm 839: FIAT, A New Paradigm for Computing Finite Element Basis Functions". In: *ACM Transactions on Mathematical Software* 30.4 (Dec. 2004), pp. 502–516.

[24] Robert C. Kirby and Anders Logg. "A Compiler for Variational Forms". In: *ACM Transactions on Mathematical Software* 32.3 (Sept. 2006), pp. 417–444.

[25] Matthew G. Knepley and Andy R. Terrel. "Finite Element Integration on GPUs". In: *ACM Transactions on Mathematical Software* (). Forthcoming.

[26] Erik Lindholm et al. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55.

[27] Anders Logg and Garth N. Wells. "DOLFIN: Automated Finite Element Computing". In: *ACM Transactions on Mathematical Software* 37.2 (Apr. 2010), 20:1–20:28.

[28] Graham R. Markall, David A. Ham, and Paul H. J. Kelly. "Towards generating optimised finite element solvers for GPUs from high-level specifications". In: *Procedia Computer Science* 1.1 (2010), pp. 1815–1823.

[29] Gordon E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (Apr. 1965).

[30] Kristian B. Ølgaard, Anders Logg, and Garth N. Wells. "Automated Code Generation for Discontinuous Galerkin Methods". In: *SIAM Journal on Scientific Computing* 31.2 (2008), pp. 849–864.

[31] Youcef Saad and Martin H. Schultz. "GMRES: A generalized minimal residual method for solving nonsymmetric linear systems". In: *SIAM Journal on Scientific Computing* 7.3 (1986), pp. 856–896.

[32] John E. Stone et al. "Accelerating molecular modeling applications with graphics processors". In: *Journal of Computational Chemistry* 28.16 (2007), pp. 2618–2640.

[33] Taras P. Usyk, Ian J. LeGrice, and Andrew D. McCulloch. "Computational model of three-dimensional cardiac electromechanics". In: *Computing and Visualization in Science* 4.4 (2002), pp. 249–257.

Other written references

[34] Satish Balay et al. *PETSc Users Manual*. Version 3.2. Argonne National Laboratory. Sept. 2011. URL: `http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf` (visited on 02/28/2012).

[35] Muthu Manikandan Baskaran and Rajesh Bordawekar. *Optimizing Sparse Matrix-Vector Multiplication on GPUs*. Tech. rep. IBM, Dec. 2008. URL: `http://domino.watson.ibm.com/library/Cyberdig.nsf/papers/1D32F6D23B99F7898525752200618339/$File/rc24704.pdf` (visited on 04/18/2012).

[36] Nathan Bell and Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. Tech. rep. NVR-2008-004. NVIDIA, Dec. 2008. URL: `http://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf` (visited on 04/18/2012).

[37] *CUDA C Best Practices Guide*. Version 4.1. NVIDIA. 2012. URL: `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf` (visited on 02/17/2012).

[38] Martin H. Gutknecht. "A Brief Introduction to Krylov Space Methods for Solving Linear Systems". In: *Proceedings of the International Symposium on Frontiers of Computational Science 2005*. Springer, 2007, pp. 53–62.

[39] Tom Lyche. "Lecture Notes for Inf-Mat 4350". 2010. URL: http://www.uio.no/studier/emner/matnat/ifi/INF-MAT4350/h10/book2010.pdf (visited on 05/15/2011).

[40] Graham Markall. *Accelerating Unstructured Mesh Computational Fluid Dynamics on the NVidia Tesla GPU Architecture*. Independent Study Option (ISO) report. 2009. URL: http://www.doc.ic.ac.uk/~grm08/gmarkall_iso1.pdf (visited on 02/16/2012).

[41] Graham Markall. "Generatively Programming Galerkin Projections on General Purpose Graphics Processing Units". Master's thesis. Imperial College London - Department of Computing, Sept. 2009. URL: http://www.doc.ic.ac.uk/~grm08/graham_markall_msc_report.pdf (visited on 01/17/2012).

[42] Paulius Micikevicius. "Fundamental Optimizations". Slides for talk at GTC 2010. URL: http://www.nvidia.com/content/GTC-2010/pdfs/2011_GTC2010.pdf (visited on 01/18/2012).

[43] Victor Minden, Barry Smith, and Matthew G. Knepley. "Preliminary Implementation of PETSc using GPUs". In: *Proceedings of the 2010 International Workshop of GPU solutions to Multiscale Problems in Science and Engineering*. 2010. URL: http://www.mcs.anl.gov/petsc/petsc-as/features/gpus.pdf (visited on 01/16/2012).

[44] *NVIDIA CUDA C Programming Guide*. Version 4.2. NVIDIA. 2012. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (visited on 05/01/2012).

[45] *NVIDIA CUDA Getting Started Guide for Linux*. Mar. 2011. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Getting_Started_Linux.pdf (visited on 04/12/2012).

[46] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. White paper. NVIDIA, 2009. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (visited on 01/18/2012).

[47] *NVIDIA Tesla C2075 Companion Processor*. Datasheet. NVIDIA, 2011. URL: http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf (visited on 01/18/2012).

[48] Florian Rathgeber. "Automated Finite Element Computations in the FEniCS Framework using General Purpose Graphics Processing Units. On the road towards interactive simulation". Master's thesis. KTH Computer Science and Communication, 2010. URL: www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2010/rapporter10/rathgeber_florian_10106.pdf (visited on 01/18/2012).

[49] Jonathan Richard Shewchuk. "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain". Version $1\frac{1}{4}$. 1994. URL: http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf (visited on 02/13/2012).

[50] *SpeedIT Extreme Library Programmers Guide*. Version 1.2. Vratis Ltd. URL: http://speedit.vratis.com/licenses/SpeedIT_Extreme_Library_Programmers_Guide12.pdf (visited on 01/16/2012).

[51] Nathan Whitehead and Alex Fit-Florea. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. White paper. NVIDIA, 2011. URL: http://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus (visited on 05/01/2012).

Online references

[52] Satish Balay et al. *PETSc web page*. 2012. URL: http://www.mcs.anl.gov/petsc (visited on 01/19/2012).

[53] *Basic Linear Algebra Subprograms (BLAS)*. URL: http://www.netlib.org/blas/ (visited on 03/30/2012).

[54] Nathan Bell and Michael Garland. *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*. URL: http://cusp-library.googlecode.com (visited on 01/12/2012).

[55] *CGAL*. URL: http://www.cgal.org (visited on 01/19/2012).

[56] *cuBLAS (NVIDIA CUDA Basic Linear Algebra Subroutines)*. NVIDIA. URL: http://developer.nvidia.com/cublas (visited on 03/30/2012).

[57] *CUDA. Parallel Programming and Computing Platform*. NVIDIA. URL: http://www.nvidia.com/object/cuda_home_new.html (visited on 01/16/2012).

[58] *CUDA Toolkit*. NVIDIA. URL: http://developer.nvidia.com/cuda-toolkit (visited on 04/02/2012).

[59] *CULA tools*. Version S1. EM Photonics. URL: www.culatools.com (visited on 01/12/2012).

[60] *cuSPARSE*. NVIDIA. URL: developer.nvidia.com/cusparse (visited on 01/12/2012).

[61] *Dorsal*. URL: https://launchpad.net/dorsal (visited on 04/02/2012).

[62] *Fermi website*. NVIDIA. URL: http://www.nvidia.com/object/fermi_architecture.html (visited on 01/18/2012).

[63] *Fluidity*. Applied Modelling and Computation Group, Imperial College. URL: http://amcg.ese.ic.ac.uk/index.php?title=Fluidity (visited on 01/31/2012).

[64] *GCC, the GNU Compiler Collection*. Free Software Foundation, Inc. URL: http://gcc.gnu.org (visited on 04/28/2012).

[65] *GNU General Public License*. Version 3. Free Software Foundation, Inc. 2007. URL: http://www.gnu.org/licenses/gpl.html (visited on 04/21/2012).

[66] *GPU Computing SDK*. NVIDIA. URL: http://developer.nvidia.com/gpu-computing-sdk (visited on 02/14/2012).

[67] Jared Hoberock and Nathan Bell. *Thrust: A Parallel Template Library*. URL: http://www.meganewtons.com/ (visited on 01/16/2012).

[68] *MATLAB*. The MathWorks, Inc. URL: http://www.mathworks.se/products/matlab/ (visited on 04/22/2012).

[69] *Message Passing Interface Forum*. URL: http://www.mpi-forum.org/ (visited on 01/19/2012).

[70] *MTL4*. URL: http://mtl4.org (visited on 03/29/2012).

[71] *OpenCL*. Khronos Group. URL: http://www.khronos.org/opencl/ (visited on 01/16/2012).

[72] Francois Pellegrini. *SCOTCH*. URL: http://www.labri.fr/perso/pelegrin/scotch (visited on 01/19/2012).

[73] *Performance of SpMV in CUSPARSE, CUSP and SpeedIT*. Vratis Ltd. URL: http://vratis.com/blog/?p=1 (visited on 02/01/2012).

[74] *SpeedIT*. Vratis Ltd. URL: http://speedit.vratis.com (visited on 01/12/2012).

[75] *SWIG*. URL: http://www.swig.org (visited on 01/19/2012).

[76] *The FEniCS Project*. URL: www.fenicsproject.org (visited on 02/22/2012).

[77] *The OpenMP API Specification for parallel programming*. URL: http://openmp.org (visited on 03/27/2012).

[78] *Top500*. URL: http://top500.org (visited on 04/21/2012).

[79] *Trilinos*. URL: http://trilinos.sandia.gov/ (visited on 01/19/2012).

[80] *uBLAS*. URL: http://www.boost.org/libs/numeric/ublas/doc/ (visited on 01/19/2012).

[81] *ViennaCL*. Institute for Microelectronics, TU Wien. URL: http://viennacl.sourceforge.net/ (visited on 01/12/2012).

[82] *ViennaCL Benchmarks*. Institute for Microelectronics, TU Wien. URL: http://viennacl.sourceforge.net/viennacl-benchmarks.html (visited on 01/16/2012).