Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA INFORMATYKI

PRACA MAGISTERSKA

KRZYSZTOF KUŹNIK

RÓWNOLEGŁY SOLWER WIELOFRONTALNY DLA SYMULACJI IZOGEOMETRYCZNYCH NA GPU BAZUJĄCY NA GRAMATYCE GRAFOWEJ

PROMOTOR:
dr hab. Maciej Paszyński

Kraków 2012

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

.....

PODPIS

AGH

University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF SCIENCE THESIS

KRZYSZTOF KUŹNIK

GRAPH GRAMMAR BASED PARALLEL MULTIFRONTAL SOLVER FOR ISOGEOMETRIC SIMULATIONS ON GPU

SUPERVISOR:
Maciej Paszyński Ph.D., D.Sc.

Kraków 2012

I would like to thank my supervisor **Maciej Paszyński**, Ph.D., D.Sc. and **Victor Calo**, Ph.D. from King Abdullah University of Science and Technology for the guidance and support I received from them during the creation of this thesis.

Contents

This page is intentionally left blank.

1. Thesis and its Motivation

The main goal of this work is to prove the usefulness of graph grammars in the modeling and analysis of complex parallel algorithms. This methodology is applied to create an effective algorithm for a problem from a still little appreciated field of isogeometric analysis. The utilization of a graphics processing unit (GPU) for this purpose allows to verify the theoretical model in a real application.

Isogeometric analysis is the approach to integrate the process of the computer aided design (CAD) and finite element analysis (FEA). A huge number of engineering systems require various methods of computational analysis and simulations including the simulations of heat transfer, fluid dynamics, structural mechanics, etc. All of those problems are somehow connected with solving partial differential equations (PDEs) using the finite element method (FEM). On the other hand, there is CAD software used for modeling. To represent surfaces it uses NURBS curves because of their flexibility and precision. However, models prepared in such way are not instantly ready for analysis. Creation of the geometry appropriate for FEA is a non-trivial task and often takes up to 80% of the entire analysis time [CHB09]. Hence the idea of using a single geometric model for both design and analysis (*isogeometric*). The main problem with this approach is that the computational cost of analysis is considerably higher than for ordinary FEA. Therefore there is a need for new effective algorithms which will be able to face this obstacle. The parallel multi-frontal solver for PDEs described in this work is a perfect example of such algorithm.

Recent rapid development of GPUs had lead to the situation where not only are they used for the purpose of graphics but they can act as a multi core computer for extensive calculations. Almost every modern GPU supports thousands of active threads which can process data much more efficiently than ordinary CPUs. Moreover, under some conditions a GPU can be used to simulate a situation where there is an infinite number of cores to use, which is very important for the sake of this work.

The main thesis of this work is stated as follows:

It is possible to implement a parallel multifrontal solver for the isogeometric final element method, which time of execution grows as $O(log\ N)$ ***for the number of elements in the computational domain denoted by*** N**.**

Although the subject might be unclear for the reader, the next chapter (chapter 2) aims to provide the basic knowledge required to entirely understand the discussed problem and its solution.

The solver algorithm presented in this work is expressed as a sequence of the graph grammar productions. Those productions are grouped into sets of independent tasks which can run concurrently. The graph grammar allows to easily inspect the algorithm flow and its concurrency. This is crucial from the point of view of the computational cost estimation. Chapter 3 provides full definition of the problem together with the explanation of the utilized graph grammar productions.

Chapter 4 is a description of the solver implementation. Implementation of each graph grammar production is analyzed in terms of the time cost, memory usage and concurrency. The chapter contains also specification of all applied optimizations.

The final results presented in chapter 5 contain algorithm analysis and a comparison against existing software. Ultimate goal is to prove the theoretical model i.e. that the solver execution time grows logarithmically with a growing number of elements in FEM. Moreover, the GPU solver is expected to run much faster than its CPU counterpart.

2. Introduction

This chapter contains a theoretical introduction needed to fully understand the problem which is solved. The introduction contains most important details about differential equations, B-spline functions, methods of solving linear systems of equations, graph grammars and the GPU programming model.

2.1. Differential equations

Simulations of physical processes require to solve wide range of various **differential equations**. Problems from domains of mechanics, fluid flow, heat transfer are all expressed as a set of differential equations. Some simple cases can be solved manually but for most engineering problems people use highly sophisticated software to achieve good solutions in a considerably short time.

2.1.1. Ordinary differential equations (ODEs)

$$F(x, u, u', u'', ..., u^{(n)}) = 0 \tag{2.1}$$

Formula 2.1 is called **ordinary differential equation (ODE)** when u is a function of one independent variable x. u' and u'' are respectively first and second derivative of u with respect to x. ODE is considered to be n^{th} order if operator F depends on n^{th} derivative of u in a non-trivial way. u which satisfies (2.1) is called the solution of a differential equation [Zwi98].

2.1.2. Partial differential equations (PDEs)

$$F(x, u, Du, D^2u, ...) = 0 \tag{2.2}$$

$$D^k u(x) := \left\{ D^\alpha u(x) = \frac{\partial^{|\alpha|} u(x)}{\partial x_1^{\alpha_1} \dots \partial x_n^{\alpha_n}} : |\alpha| = k \right\} \tag{2.3}$$

When u is a function of more than one independent variable and F depends on partial derivatives of u the equation is called **partial differential equation (PDE)**. Accordingly to the differentiation operator D (Formula 2.3) we can define a PDE order k [Eva98].

2.1.3. Linear and non-linear DEs

The differential equation is called **linear** if the unknown function does not appear within powers or any complicated functions i.e. the differentiation operator F is a linear operator [Ger98]. A linear DE can be written as follows:

$$Fu = f(x) \tag{2.4}$$

$$F(u) \equiv \frac{d^n u}{dx^n} + A_1(x)\frac{d^{n-1}u}{dx^{n-1}} + \cdots + A_{n-1}(x)\frac{du}{dx} + A_n(x)u \tag{2.5}$$

f in formula 2.4 is called a *forcing term*. When $f(x) \equiv 0$ the equation is called *homogeneous*, otherwise it is *inhomogeneous*.

If a differential equation cannot be stated in the form 2.4 it is **non-linear**.

2.1.4. Examples of PDEs

Below there is a set of well known partial differential equations with example applications.

Poisson's equation

Poisson's equation is a second order PDE of a form:

$$\triangle u = f \tag{2.6}$$

Where $\triangle$ is a Laplace operator defined as follows:

$$\triangle = \nabla^2 = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + ... + \frac{\partial^2}{\partial x_n^2} \tag{2.7}$$

f is a forcing term, u is the unknown function – the solution of the Poisson's equation. For a one dimensional case a Poisson's equation becomes a second order ODE.

$$u''(x) = f(x) \tag{2.8}$$

For $f \equiv 0$ the formula 2.6 becomes **Laplace's equation**:

$$\triangle u = 0 \tag{2.9}$$

The Poisson's equation is utilized in electrostatics, gravity and heat transfer. It is a fairly simple PDE so it is often used as a benchmark problem for PDE solvers.

Navier–Stokes equations

Navier–Stokes equations are used for analysis of fluids flow[PW07]. Usually they find application in modeling weather, air flow around aircraft wings, flow of the water in a pipe, etc. General form of **Navier-Stokes equations** looks as follows:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \nabla\cdot\mathbf{T} + \mathbf{f} \tag{2.10}$$

Where ρ is the fluid density, $\mathbf{v}$ is the flow velocity, p is the pressure, $\mathbf{T}$ is the stress tensor and $\mathbf{f}$ represents body forces acting on the fluid.

The variation of **Navier–Stokes equations** are **shallow water equations** also known as **Saint Venant equations**. They are somehow simpler version of **Navier–Stokes equations** in the case where the horizontal length scale is much greater than the vertical length scale. They are often used to model floods.

Cahn–Hilliard equations

The **Cahn–Hilliard equation**[CH58] is an equation which describes the process of phase separation – for example separation of oil and water shaken in a test-tube. In its general form it looks as follows:

$$\frac{\partial c}{\partial t} = D\nabla^2 \left(c^3 - c - \gamma\nabla^2 c\right) \tag{2.11}$$

Where $c = \pm 1$ is the concentration of the fluid, D is a diffusion coefficient, γ is the length of the transition region between fluids.

2.2. Methods for solving DEs

There are two basic approaches to solving differential equations:

- **analytical** – This is an exact, but also difficult method. For most engineering problems it is usually impossible to obtain analytical solution.
- **numerical** – This method usually provides approximate solution, but is better suited for machine computations. Two of the most known numerical methods are the finite difference method (FDM) and the finite element method (FEM).

2.2.1. Analytical solution

This is a strictly mathematical approach to solving differential equations. It is based on analytical transformations of a given equation. Despite that the method assures the best quality of the solution it is practically useless for machine computations. Although there are plenty of analytical methods (which are suited only for certain classes of differential equations), for most of the cases engineers are forced to use numerical methods.

On the one hand, there is a big improvement in the domain of solving differential equations analytically. The process is no longer required to be performed manually. Mathematical software such as Mathematica[mat12a], Wolfram Alpha[wol12] or Matlab[mat12b] provide wide range of mechanisms for such computations.

On the other hand, there are multiple types of differential equations which cannot be solved analytically because of the size and complexity of the problem. For them the numerical methods provide sufficient approximation of the analytical solution and sometimes are the last resort.

2.2.2. Finite element method

Finite element method[SF08] is a numerical method widely used in plenty areas such as aeronautical engineering, fluid dynamics or even weather prediction. Whole method consists of several steps:

1. **Strong formulation** – this is the ordinary form of the differential equation. For the sake of this explanation simple Laplace's equation can be used with Dirichlet boundary condition:

$$\begin{cases} -\frac{d^2u}{dx^2} &= f \\ u &\in V = \left\{v \in H^1(\Omega \subset \mathbb{R}) : v(\partial\Omega) = 0\right\} \\ f &\in C_0(\Omega) \end{cases} \tag{2.12}$$

2. **Weak formulation** – a weak formulation is obtained from a strong formulation by taking L_2 scalar product of equation 2.12 with a test function $v \in V$. L_2 product is defined as follows:

$$L_2(u, v) = \int_\Omega uv \, d\Omega \tag{2.13}$$

After taking L_2 product ...

$$-\int_\Omega \frac{d^2u}{dx^2} v \, dx = \int_\Omega fv \, dx \tag{2.14}$$

... and integration by parts and applying boundary conditions weak formulation looks as follows:

$$\int_\Omega \frac{du}{dx}\frac{dv}{dx} \, dx = \int_\Omega fv \, dx \tag{2.15}$$

Which for readability can be written like this:

$$b(u, v) = l(v) \quad \forall v \in V \tag{2.16}$$

3. **Discretization** – this is the part from which the method takes its name. Note that V is infinite dimensional space and therefore cannot be used in any machine computations. In this step the computational domain is divided into *finite elements*, which implies taking finite dimensional subspace of V (Let it be V_N and let $dim\ V_N = N$). Problem is now redefined to its discrete counterpart.

u_N will be an approximation for u. It will be a linear combination of chosen basis functions:

$$u \approx u_N = \sum_{i=1}^{N} u_i \phi_i \tag{2.17}$$

For a simplicity the basis functions ϕ_i can be used as test functions. This produces a new (**discrete**) form of the weak formulation which is also a system of linear equations:

$$\sum_{i=1}^{N} u_i \int_\Omega \frac{d\phi_i}{dx}\frac{d\phi_j}{dx} \, dx = \int_\Omega f\phi_j \, dx \qquad \text{for } j = 1, ..., N \tag{2.18}$$

It can be also written in matrix form:

$$\mathbf{B}u = l \tag{2.19}$$

Where:

$$\mathbf{B}_{i,j} = \int_{\Omega} \frac{d\phi_i}{dx}\frac{d\phi_j}{dx}\,dx \tag{2.20}$$

$$l_j = \int_{\Omega} f\phi_j\,dx \tag{2.21}$$

$\mathbf{B}$ here is called the *stiffness matrix* and f is called the *forcing vector*. Solution to this linear system will be coefficients for the linear combination presented in formula 2.17. u_N from this formula is the approximate solution for the differential equation.

One of the most important steps in the procedure shown above is the choice of basis functions. Bad choice may significantly influence the size of the matrix, the execution time and the quality of the solution. The following section explains why B-splines are a good choice for basis functions.

2.3. B-Splines

B-spline is a short for *basis spline*. B-splines are functions widely used in CAD software because of their simplicity, higher continuity and numerical stability.

B-spline functions are defined on a knot vector $[\xi_1, ..., \xi_{m-1}]$ where ξ_i are called knots and they meet following condition:

$$\xi_1 \leq \xi_2 \leq ... \leq \xi_{m-1} \tag{2.22}$$

$m - p - 1$ basis B-splines of degree p can be defined on the provided knot vector. $N_{k,p}$ denotes k^{th} basis B-spline of degree p.

$$N_{i,0}(\xi) = I_{[\xi_i,\xi_{i+1}]} \tag{2.23}$$

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{x_{i+p} - \xi_i}N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{x_{i+p+1} - \xi_{i+1}}N_{i+1,p-1}(\xi) \tag{2.24}$$

Recursive formula 2.23 - 2.24 is called **Cox-de-Boor formula**[dB01]. Figure 2.1 presents a visualization of this formula presenting how to obtain a quadratic B-spline.

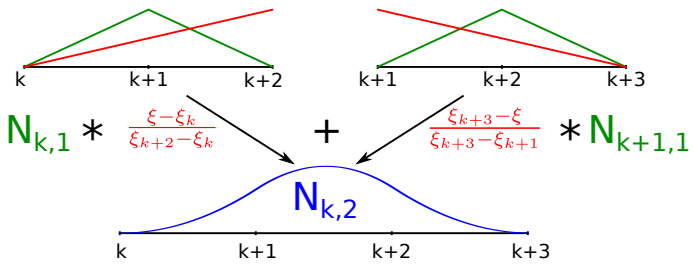

FIGURE 2.1: VISUALIZATION OF COX-DE-BOOR FORMULA FOR QUADRATIC B-SPLINE

Advantages of B-spline basis functions over other kinds of basis functions include:

– **higher global continuity of the solution** – almost all types of basis functions assure solution that is C^∞ continuous on the elements but only C^0 on the element boundaries. The situation is much better with B-splines – they provide C^∞ continuity on the elements and (depending on B-spline degree p) C^{p-1} on the element boundaries. This can be seen on the figure 2.2. If a solution is expressed as a linear combination of second order Lagrange basis (red) it does not provide a continuous derivative. On the other hand, if B-spline basis of degree $p = 2$ is used (blue) solution will be C^1 continuous in the entire domain.

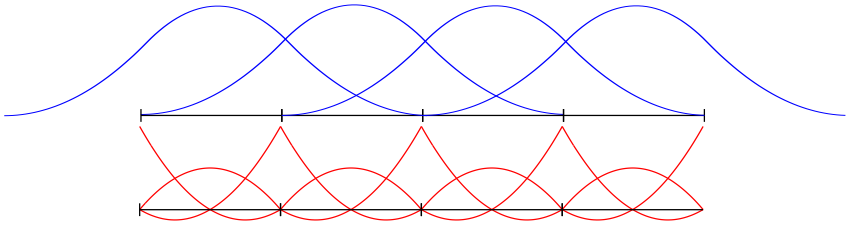

FIGURE 2.2: B-SPLINE (TOP) AND LAGRANGE (BOTTOM) BASIS FUNCTIONS OF SECOND ORDER

– **application in isogeometric analysis** – B-splines are basis functions for NURBS curves used for models in CAD software. Once defined geometry is applicable for both design and analysis, which is not a case for other basis functions.

– **application in more complex problems** – higher global continuity of the solution allows usage of B-spline basis functions for higher order PDEs (such as Cahn-Hilliard)[GCH09].

Among all their advantages B-splines have also some downsides. The biggest one is that B-spline basis functions have wider support than Lagrange polynomials or Demkowicz hierarchical functions[DKP$^+$07]. This causes more non-zero entries in the stiffness matrix which in turn causes longer and more expensive computations.

2.4. Linear system of equations

Solving the final system of equations is the most expensive step of FEM. Choice of the appropriate solver has a great influence on both the quality of the solution and the solver execution time.

Generally there are two approaches to solving linear systems:

– **direct** – where solution is obtained by making direct modifications to the matrix

– **iterative** – where solution is obtained by iterative improvement of initial (usually bad) solution

Note that stiffness matrix is a sparse matrix and it might be tempting to use an iterative solver. They usually are faster and easier to implement. However, iterative solvers provide only approximate solution of the linear system. This adds another error for final solution of the PDE. What is

more, iterative solvers do not always converge to the final solution or they converge really slowly. This restricts the domain of problems that can be solved.

The fact that the stiffness matrix is sparse excludes also direct use of well known libraries for linear algebra such as LAPACK because they are designed for dense problems. It would be impossible to solve any bigger problem because of enormous memory consumption. More flexible solution is needed.

2.4.1. Multifrontal solvers

A **multifrontal solver**[DR83] is a direct solver for linear systems of equations with sparse matrices. The main idea behind it is to partition a single big sparse matrix into multiple small dense matrices. These small matrices are called fronts. During a processing of the frontal matrices some degrees of freedom are eliminated, then new frontal matrices are chosen. The procedure is repeated until all degrees of freedom are eliminated. This kind of processing has several advantages in comparison to ordinary direct methods:

1. Frontal matrices are dense so it is possible to use well known methods such as Gaussian elimination or use libraries for dense matrices such as LAPACK. There is no overhead on processing or storing zero elements. Computations are much more efficient.

2. Frontal matrices are constructed for separate degrees of freedom so they can be processed concurrently. This allows significant speed-up of computations if multi-core machine is used.

3. Matrix is not stored explicitly in the memory. This causes big savings in the storage area and therefore allows solving bigger problems.

One significant disadvantage of the multifrontal solver is the time required for a matrix preprocessing. Solver algorithm has to analyze whole matrix to predict what is the best partitioning. As it is simple for banded matrices, in general case it is a non-trivial task which requires sophisticated mechanisms. Luckily when using a multifrontal solver for matrices resulting from FEM some of this work can be simplified by a special frontal matrices assembly.

2.5. Graph grammars

Graph grammar[Roz97] is a method of structural processing of graphs. It allows to precisely describe the process of a graph transformation by defining a set of grammar productions. By analogy to formal languages graph grammar productions modify graphs starting with a single node (starting symbol) until they reach a state where all nodes cannot be processed any more (terminal symbols). Example applications of graph grammars include image generation, algorithm design and software engineering. Graph grammars have been also successfully applied in a design of the PDE solvers [PS10, Pas09]. Further chapters of this thesis contain full definition of a graph grammar for a differential equation solver.

2.6. GPU programming model (CUDA)

A recent rapid development of GPUs had lead to the situation where not only are they used for a purpose of the graphics but they can act as a multi-core computer for extensive calculations. Almost every modern GPU supports thousands of active threads which can process data much more efficiently than the ordinary CPU. Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by NVIDIA which allows to use a GPU for computations. Below are some details on GPU programming model introduced by CUDA and details on parts of the main interest – memory and GPU itself.

2.6.1. GPU specification

In the early days of their existence the GPUs were designed only for preparing images in a memory and sending them to a display. Later on, people started to see the GPU potential for a parallel data processing which lead to the situation when nowadays there are GPU devices created only for computations. To classify their processors NVIDIA introduced the notion of **compute capability**, which is used to describe GPU predispositions for performing calculations. The earliest models of CUDA GPUs had compute capability **1.0** and the newest have the compute capability of **3.0**. With a growing compute capability the power of GPUs increases:

- Support double precision computations instead of single precision.
- Support for more threads
- Bigger shared memory size
- Bigger number of registers per multiprocessor
- Support for 3D thread structures

Generally the GPU consists of several multiprocessors. Each multiprocessor consists of tens or hundreds of cores. Each multiprocessor has a shared memory which is shared among all its cores.

2.6.2. Memory model

There are four basic kinds of memory available on GPU:

1. **Global memory** – off-chip big ($\sim$ 1GB) and slow memory
2. **Shared memory** – on-chip low latency memory, although of a limited size (48 KB)
3. **Constant memory** – off-chip cached memory initialized by host
4. **Texture memory** – off-chip read-only memory optimized for 2D spatial locality

Usually only two former kinds are used for computations.

There are two important things to remember:

1. Global memory access is transactional. No matter how much data is needed to be transferred from the global memory to GPU registers it is always at least 32B (for 32bit data it is 128B). To optimize global memory access consecutive threads should access consecutive addresses in the global memory to fully utilize the bandwidth.

2. Shared memory is used by all threads on a multiprocessor. If there is too much shared memory needed by a user function it may restrict the number of thread blocks run on a single processor and therefore significantly decrease the performance.

2.6.3. Threads, blocks and grid

The CUDA programming model assumes that software is independent of a device to allow a high scalability and a compatibility with different hardware. Hence the notion of a *block* (of threads). A block is a set of threads which run on a single multiprocessor and which use a common shared memory. Several blocks might be run on one multiprocessor but one block is never split between multiprocessors. To make programming easier blocks as well as threads inside a block can be organized into 2D or 3D grids which simplifies processing of matrices or volumes [NVI11]. Figure 2.3 shows the organization of threads on GPU for a two dimensional case.

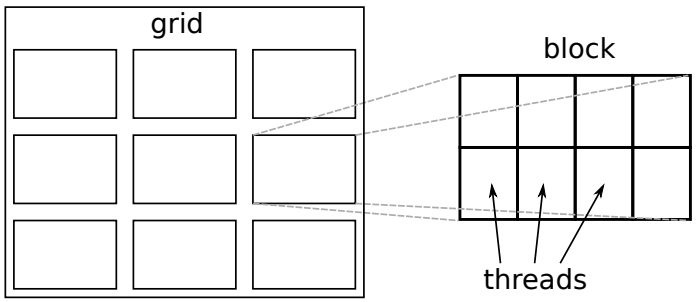

FIGURE 2.3: ORGANIZATION OF THREADS IN 2D

3. Problem formulation

The ultimate goal of this thesis is to prove the usefulness of the graph grammars in a design of modern parallel algorithms from the domain of isogeometric analysis. Theory presented in this chapter suggests that having enough processing units one is able to solve 1D differential equation using isogeometric FEM in time $O(log\ N)$ where N is the number of elements. In this thesis it is proven that this theoretical model is correct and it is possible to implement such a solver.

3.1. Solved DE

3.1.1. Strong formulation

As an example application for the solver the following differential equation is used:

$$\text{Find } u \in C^2(\Omega) \text{ such that:}$$
$$\begin{cases} -\frac{d}{dx}\left(a(x)\frac{du(x)}{dx}\right) + b(x)\frac{du(x)}{dx} + c(x)u(x) &= f(x) \\ u(0) &= 0 \\ a(l)\frac{du}{dx}(l) + \beta u(l) &= \gamma \end{cases} \tag{3.1}$$

Where:

$$\Omega = [0, l] \tag{3.2}$$
$$a \in C^1(0, l) \tag{3.3}$$
$$b, c, f \in C^0(0, l) \tag{3.4}$$
$$\beta, \gamma \in \mathbb{R} \tag{3.5}$$

In a general case it is impossible to solve this equation analytically. However, using the Finite Element Method it is possible to provide a good approximation of the solution for arbitrary parameters conforming to the provided conditions.

Without loss of generality for the rest of this thesis it is assumed that $l = 1$

3.1.2. Weak formulation

Weak formulation of this equation is obtained by taking L^2 inner product with test function v that satisfies condition $v(0) = 0$

$$-\frac{d}{dx}\left(a\frac{du}{dx}\right)+b\frac{du}{dx}+cu=f \tag{3.6}$$

$$\int_\Omega\left(-\frac{d}{dx}\left(a\frac{du}{dx}\right)+b\frac{du}{dx}+cu\right)v\,dx=\int_\Omega fv\,dx \tag{3.7}$$

$$\int_\Omega -v\frac{d}{dx}\left(a\frac{du}{dx}\right)\,dx+\int_\Omega bv\frac{du}{dx}\,dx+\int_\Omega cuv\,dx=\int_\Omega fvdx \tag{3.8}$$

Integrating by parts:

$$-\left[av\frac{du}{dx}\right]_\Omega+\int_\Omega a\frac{du}{dx}\frac{dv}{dx}\,dx+\int_\Omega bv\frac{du}{dx}\,dx+\int_\Omega cuv\,dx=\int_\Omega fvdx \tag{3.9}$$

$$\int_\Omega\left(a\frac{dv}{dx}\frac{du}{dx}\,dx+bv\frac{du}{dx}+cvu\right)\,dx-\left[av\frac{du}{dx}\right]_\Omega=\int_\Omega fvdx \tag{3.10}$$

Note that:

$$\left[av\frac{du}{dx}\right]_\Omega=a(l)v(l)\frac{du}{dx}(l)-a(0)v(0)\frac{du}{dx}(0) \tag{3.11}$$

$$\tag{3.12}$$

From equation conditions we have:

$$v(0)=0 \tag{3.13}$$

$$a(l)\frac{du}{dx}(l)=\gamma-\beta u(l) \tag{3.14}$$

Then:

$$\left[av\frac{du}{dx}\right]_\Omega=\gamma v(l)-\beta u(l)v(l) \tag{3.15}$$

And finally weak formulation is:

$$\int_\Omega\left(a\frac{dv}{dx}\frac{du}{dx}\,dx+bv\frac{du}{dx}+cvu\right)\,dx+\beta u(l)v(l)=\int_\Omega fvdx+\gamma v(l) \tag{3.16}$$

Problem is redefined to:

Find $u\in V=\{u\in H^1(0,1):u(0)=0\}$ such that:

$$b(u,v)=l(v),\forall v\in V \tag{3.17}$$

Where:

$$b(u,v)=\int_\Omega\left(a\frac{dv}{dx}\frac{du}{dx}\,dx+bv\frac{du}{dx}+cvu\right)\,dx+\beta u(l)v(l) \tag{3.18}$$

$$l(v)=\int_\Omega fvdx+\gamma v(l) \tag{3.19}$$

3.1.3. Discretization

Note that V is infinite dimensional space of functions and therefore weak formulation in this form is still useless for machine computations. To make it practical a finite dimensional subspace

of V is chosen. It is worth to mention that the choice of this subspace has a great influence on the final solution - bigger subspace usually results in a better approximation. However, a bigger subspace results also in a longer computation and greater memory consumption.

Having a finite dimensional space V_i solution to the previously defined differential equation can be approximated as a linear combination of the basis functions ($\mathbf{v_i}$) from this space.

$$u(x) \approx \sum_i \hat{u_i} = \sum_i u_i \, \mathbf{v_i}(x) \tag{3.20}$$

Now it is possible to define discretized weak formulation.

$$\sum_i b(\hat{u_i}(x), v_j(x)) = l(v_j(x)), \forall j \tag{3.21}$$

$$\sum_i b(u_i \, \mathbf{v_i}(x), \mathbf{v_j}(x)) = l(\mathbf{v_j}(x)), \forall j \tag{3.22}$$

b is bilinear form so final form can be written as:

$$\sum_i u_i \, b(\mathbf{v_i}(x), \mathbf{v_j}(x)) = l(\mathbf{v_j}(x)), \forall j \tag{3.23}$$

In isogeometric analysis B-splines are used for basis functions. Generally for B-spline basis of degree p weak formulation looks as follows:

$$\sum_i u_i \, b(N_{i,p}(x), N_{j,p}(x)) = l(N_{j,p}(x)), \forall j \tag{3.24}$$

3.2. Graph grammar

Graph grammar productions presented in this chapter represent all basic tasks that are needed to solve 1D differential equation using isogeometric finite element method and parallel multifrontal solver for a linear system of equations.

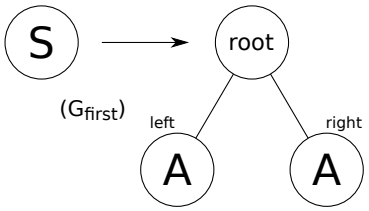

FIGURE 3.1: INITIAL PRODUCTION FOR GENERATION OF ELIMINATION TREE

3.2.1. Generation of elimination tree

Set of (G) productions is responsible for a generation of the elimination tree. The computational domain is partitioned into finite elements by applying:

– (G_{first}) - a starting production which generates $root$ of the elimination tree and two A nodes, each containing a direction attribute indicating its position in the domain. Here each of A nodes represents half of the domain. See figure 3.1

– $(G)^{left}$, (G) and $(G)^{right}$ - intermediate productions splitting each domain part into 2 new parts. Direction attributes propagate during the generation as shown in the figure 3.2.

– $(G_{last})_p$ - at the leaves level this is the production which produces final domain partitioning. It also has direction attributes. Moreover G_{last} production differs depending on B-spline degree (p) used for computations. As shown in the figure 3.3 the production $(G_{last})_p$ generates $p+1$ final nodes. $(G_{last})_1$ is used for linear B-splines, $(G_{last})_2$ for quadratic, etc.

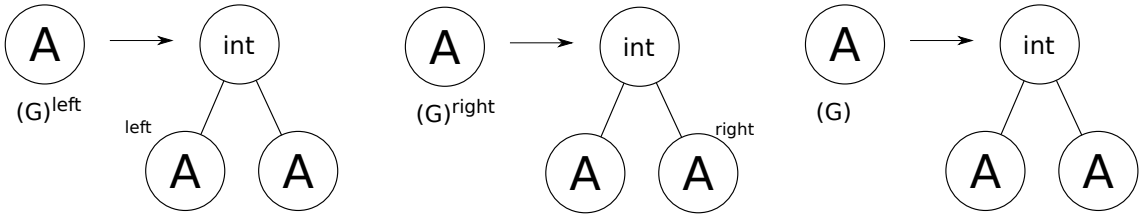

FIGURE 3.2: INTERMEDIATE PRODUCTIONS FOR GENERATION OF ELIMINATION TREE

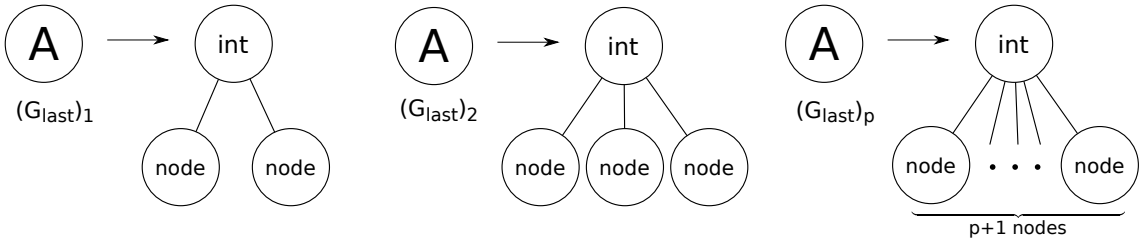

FIGURE 3.3: FINAL PRODUCTION FOR GENERATION OF ELIMINATION TREE

3.2.2. Generation of local stiffness matrices

After applying productions (G) the computational domain $\Omega = [0,1]$ is partitioned into N finite elements $\Omega = \cup_{k=1,\ldots,N}[\xi_k, \xi_{k+1}] = \cup_{k=1,\ldots,N}[\frac{k-1}{N}, \frac{k}{N}]$. Now it is time to assemble local to element stiffness matrices and force vectors. These matrices will be also frontal matrices for multifrontal solver. Each frontal matrix is generated by computing the value of $b\left(N_{j,p}\left(x\right), N_{i,p}\left(x\right)\right)$ for B-splines defined over the element e_k. Size of the frontal matrix depends on B-spline basis degree:

– For linear B-splines ($p = 1$), there are two basis functions $N_{k,1}$ and $N_{k+1,1}$ with support over an element e_k. This means that element frontal matrix size is $2 \times 2 = 4$, as it is illustrated in figure 3.4 and in formula 3.25 below.

$$\begin{bmatrix} b\left(N_{k,1}\left(x\right), N_{k,1}\left(x\right)\right) & b\left(N_{k,1}\left(x\right), N_{k+1,1}\left(x\right)\right) \\ b\left(N_{k+1,1}\left(x\right), N_{k,1}\left(x\right)\right) & b\left(N_{k+1,1}\left(x\right), N_{k+1,1}\left(x\right)\right) \end{bmatrix} \tag{3.25}$$

– For quadratic B-splines, $p = 2$, there are three functions: $N_{k,2}$, $N_{k+1,2}$ and $N_{k+2,2}$ with support over an element e_k. The element frontal matrix has $3 \times 3 = 9$ entries, see figure 3.5 and formula 3.26 below.

$$\begin{bmatrix} b\left(N_{k,2}\left(x\right), N_{k,2}\left(x\right)\right) & b\left(N_{k,2}\left(x\right), N_{k+1,2}\left(x\right)\right) & b\left(N_{k,2}\left(x\right), N_{k+2,2}\left(x\right)\right) \\ b\left(N_{k+1,2}\left(x\right), N_{k,2}\left(x\right)\right) & b\left(N_{k+1,2}\left(x\right), N_{k+1,2}\left(x\right)\right) & b\left(N_{k+1,2}\left(x\right), N_{k+2,2}\left(x\right)\right) \\ b\left(N_{k+2,2}\left(x\right), N_{k,2}\left(x\right)\right) & b\left(N_{k+2,2}\left(x\right), N_{k+1,2}\left(x\right)\right) & b\left(N_{k+2,2}\left(x\right), N_{k+2,2}\left(x\right)\right) \end{bmatrix} \tag{3.26}$$

– Generally, for B-splines of degree p there is $p + 1$ functions with support over element e_k, denoted $N_{k,p}$, ..., $N_{k+p,p}$ and the element frontal matrix has $(p + 1)^2$ entries.

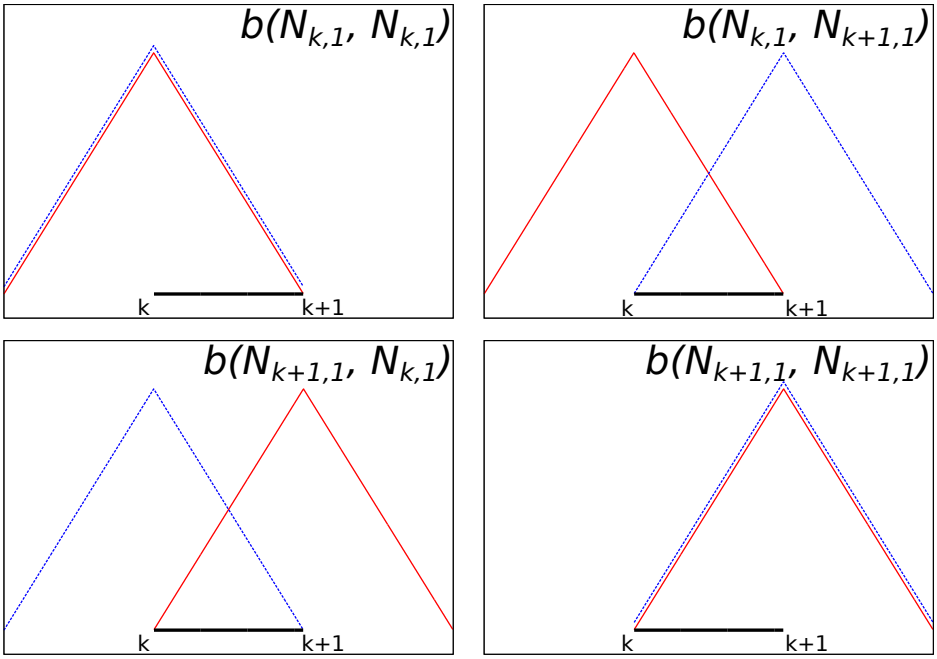

FIGURE 3.4: STRUCTURE OF FRONTAL MATRIX FOR LINEAR B-SPLINES

Productions $(A_k)_p$ are responsible for generation of local frontal matrices (LFM). They generate stiffness matrix and force vector for element e_k when B-splines of degree p are used (figure 3.6).

3.2.3. Merging of local stiffness matrices

Local stiffness matrices are also frontal matrices for the multifrontal solver. Productions (M) are responsible for merging the frontal matrices at defined level of the elimination tree. There are two types of this merge:

– $(M_{first})_p$ - a production for the initial merge depending on B-spline degree p. This is merging at leaf level of elimination tree - $p + 1$ matrices are merged for B-spline of degree p. Note that at leaf level $p + 1$ consecutive frontal matrices have only one common degree of freedom. See example for $(M_{first})_1$ on figure 3.7.

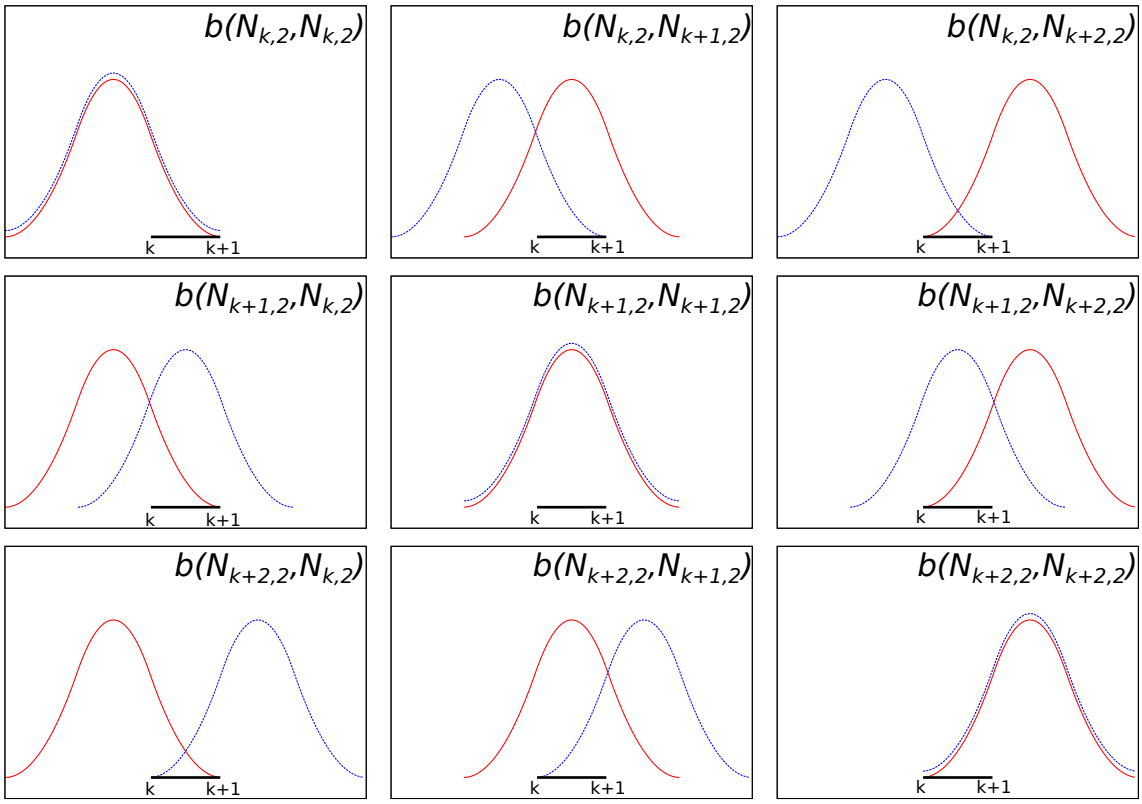

FIGURE 3.5: STRUCTURE OF FRONTAL MATRIX FOR QUADRATIC B-SPLINES

– $(M)_p$ - a production for merges up to the root of elimination tree. This production always merges 2 nodes representing consecutive parts of the domain. However, depending on B-spline degree there is a different number of rows and a different number of common degrees of freedom. For B-spline of degree p there are matrices of size $2p \times 2p$ with $2p$ common degrees of freedom. See figure 3.8.

3.2.4. Elimination of fully assembled rows

Productions (E) are responsible for eliminating fully assembled rows from merged frontal matrices (MFM). After eliminating fully assembled row(s) with Gaussian elimination, obtained Schur complement [Zha05] becomes new intermediate local frontal matrix (ILFM) ready for merging at the next level of the elimination tree. There are two different types of this elimination that can be distinguished:

– $(E_{first})_p$ - a production for eliminating exactly one fully assembled row obtained after the initial merge $(M_{first})_p$. Depending on the degree p of a B-spline basis used in FEM the row length is different. It is exactly $2p + 1$. See the example for $(E_{first})_1$ on the figure 3.9.

– $(E)_p$ - a production for eliminating fully assembled rows at intermediate levels of the elimination tree. For B-spline of degree p it eliminates exactly p rows. Each row length is exactly $3p$. See the example for $(E)_1$ on the figure 3.10.

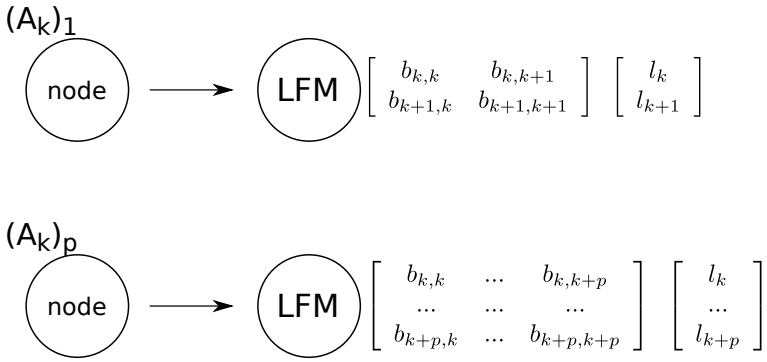

FIGURE 3.6: PRODUCTIONS FOR GENERATION OF LFM

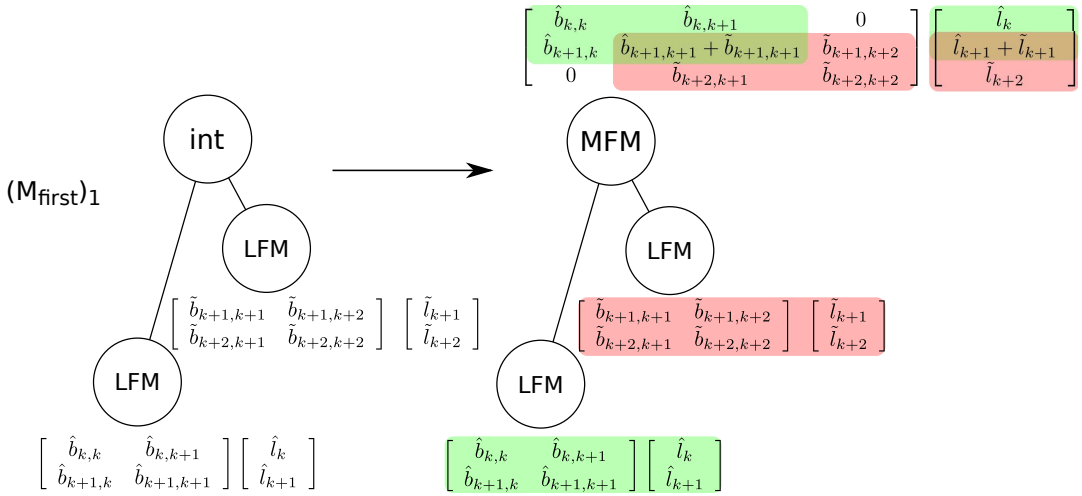

FIGURE 3.7: PRODUCTION (M_{first}) FOR MERGING OF **LFM** AT LEAF LEVEL (LINEAR B-SPLINES)

3.2.5. Root problem

After last two frontal matrices are merged there is a system of equations where all degrees of freedom are fully assembled. This is called the root problem and a production responsible for this is $(R)_p$. This system is again solved by the Gaussian elimination. Coefficients obtained from the solution of this system are also part of the final solution to the DE. They are respectively u_1, ... , u_p, $u_{N/2+1}$, ... , $u_{N/2+p}$, u_{N+1}, ... ,u_{N+p}. Having them it is possible to obtain the final solution of FEM by recursive backward substitutions. The figure 3.11 represents an example $(R)_1$ production for linear B-splines.

3.2.6. Backward substitution

Productions (B) are responsible for the recursive backward substitution. They work analogically to building elimination tree by productions (G). At this point each ILFM node contains p fully assembled rows which need $2p$ coefficients to be solved (to obtain new coefficients for those fully assembled degrees of freedom). At the leaf level there are nodes that contain exactly one row which also depends on $2p$ coefficients from the higher level of the elimination tree. Hence again

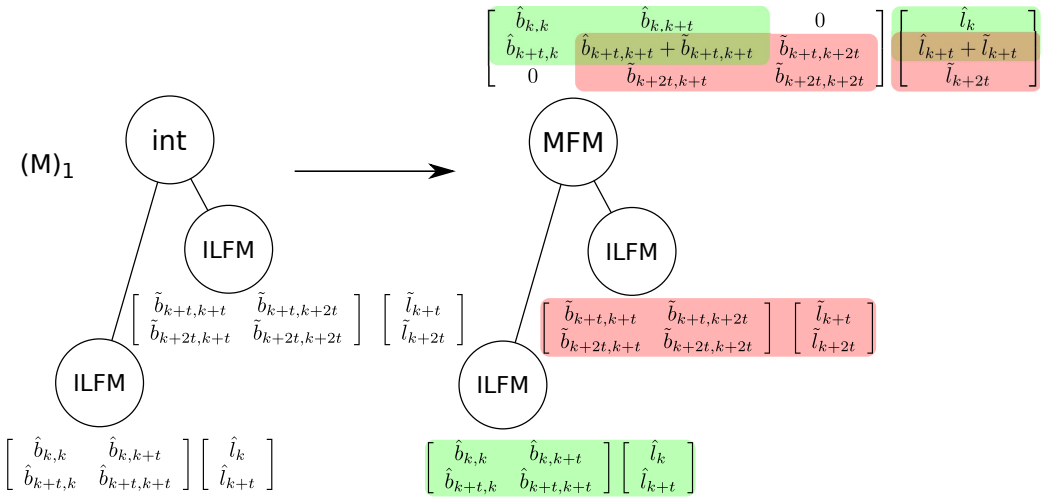

FIGURE 3.8: PRODUCTION (M) FOR MERGING OF **ILFM** AT INTERMEDIATE LEVEL (LINEAR B-SPLINES)

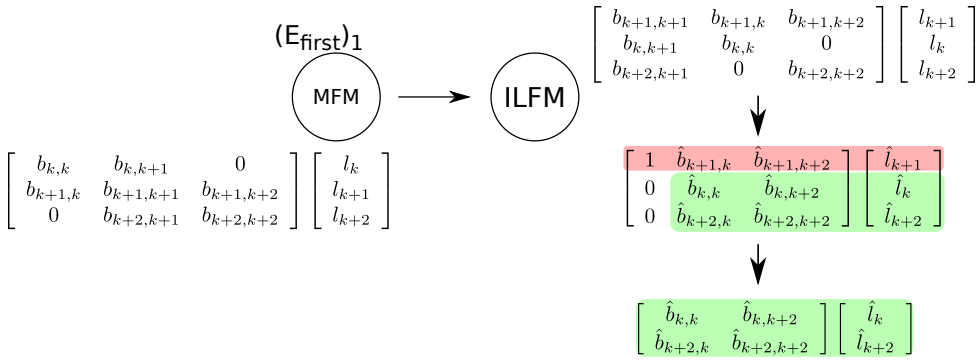

FIGURE 3.9: PRODUCTION (E_{first}) FOR ELIMINATING FULLY ASSEMBLED ROW FROM **MFM** AT LEAF LEVEL (LINEAR B-SPLINES)

there are two types of this production:

- $(B)_p$ - a production for calculating p coefficients at intermediate levels of the elimination tree. Evaluation is done with the usage of $2p$ coefficients calculated at higher levels of the tree. The figure 3.12 is an example of this production for linear B-splines.

- $(B_{last})_p$ - a production for calculating one coefficient at the leaf level of the elimination tree. The evaluation is done with the usage of $2p$ coefficients calculated at higher level of the tree. Figure 3.13 is an example of this production for linear B-splines.

3.2.7. Scheduling tasks for the solver execution

When all productions are defined it is possible to schedule all basics tasks needed for the solver execution. Figure 3.14 shows all solver steps for the linear B-spline basis functions expressed as graph grammar productions. In the figure degrees of freedom that are not yet fully assembled are denoted by underlined symbols (e.g. $\underline{\tilde{u}_2}$). A tilde sign means that degree of freedom is not computed yet.

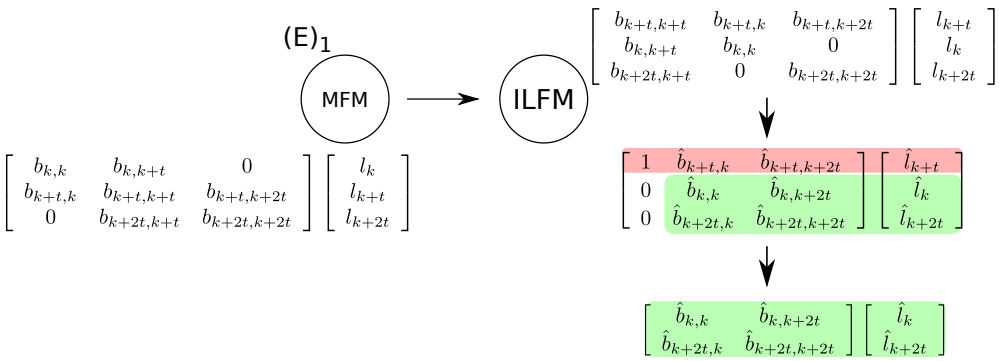

FIGURE 3.10: PRODUCTION (E) FOR ELIMINATING FULLY ASSEMBLED ROW FROM **MFM** (LINEAR B-SPLINES)

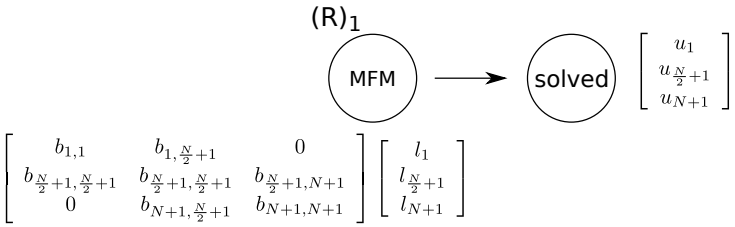

FIGURE 3.11: PRODUCTION (R) FOR SOLVING THE ROOT PROBLEM (LINEAR B-SPLINES)

At first, the domain is partitioned and the elimination tree is constructed with productions (G). When the construction is complete the frontal matrices are constructed at the leaves of the elimination tree with productions (A). Then with productions (M) and (E) applied alternately the frontal matrices are merged and fully assembled degrees of freedom are eliminated. In the end, the root problem is solved with production (R). This is followed by recursive backward substitutions (B) from the root down to the leaves.

Figure 3.15 shows analogical process for quadratic B-splines. Note that for quadratic B-splines production (M_{first}) merges three matrices and only one degree of freedom is fully assembled and ready to be eliminated. In the following steps two matrices are merged as with linear B-splines. However, after each merge two degrees of freedom are fully assembled and eliminated. Whole process can be easily generalized for higher order B-splines.

The crucial property of this algorithm is that productions (A), (M), (E) and (B) at each level of the elimination tree process independent sets of data – this allows them to be applied concurrently. Consider for example the situation presented on the figure 3.14. A sequential algorithm would firstly apply the production (G_{first}), then two instances of the production (G), then four instances of the production $(G_{last})_1$, then eight instances of the production $(A)_1$ and so on... The computational cost would grow at least as the power of 2 of input size. However, the situation is much better for the parallel algorithm – all copies of the same production are applied at once for a single level of the elimination tree. The height of the elimination tree is proportional to the logarithm of the number of elements in the computational domain. It means that having enough processing units it is possible to solve this problem in time $O(log\ N)$ where N is the number of elements.

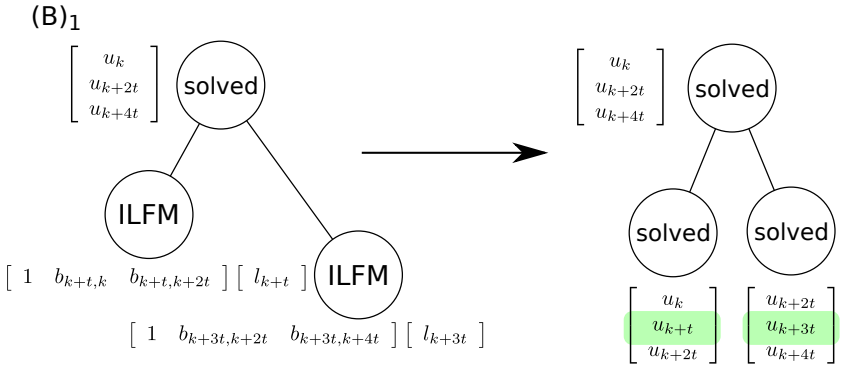

FIGURE 3.12: PRODUCTION (B) FOR BACKWARD SUBSTITUTION AT INTERMEDIATE LEVELS OF ELIMINATION TREE (LINEAR B-SPLINES)

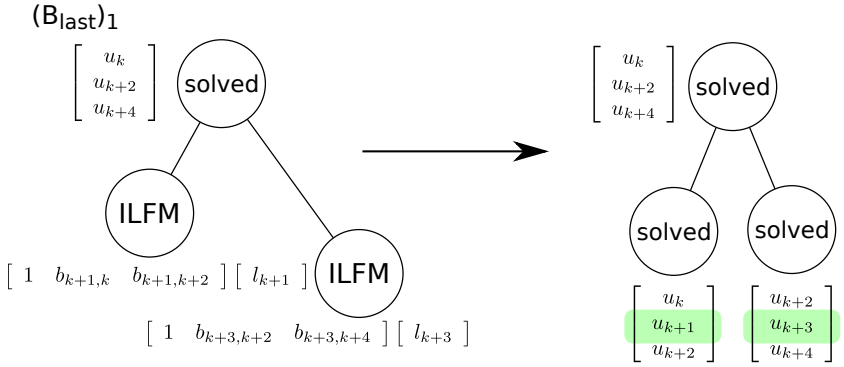

FIGURE 3.13: PRODUCTION (B_{last}) FOR BACKWARD SUBSTITUTION AT LEAF LEVEL OF ELIMINATION TREE (LINEAR B-SPLINES)

3.3. Logarithmic time cost

Total time cost of the solver execution can be estimated by the following sum (3.27):

$$T_{total} = T_{(G)} + T_{(A)} + T_{(M,E)} + T_{(R)} + T_{(B)} \tag{3.27}$$

– $T_{(G)}$ - the time of generation of the elimination tree. Note that instances of productions (G_{first}) and (G) divide single part of a domain into two new parts, and instances of production $(G_{last})_p$ into $p+1$ parts. For the target number of elements denoted by N the number of levels in the generation step is: 1 for (G_{first}), 1 for $(G_{last})_p$ and $log_2(\frac{N}{p+1}) - 1$ for (G). The time of a single production (G) or (G_{first}) is constant because it is just the bisection of the segment. Time of a single $(G_{last})_p$ production is a time of dividing a segment into $p+1$ parts so it is $O(p)$. Finally, the time for this step can be presented as it is done in formula 3.28:

$$T_{(G)} = O(1) + \sum_{i=1}^{log_2(\frac{N}{p+1})-1} O(1) + O(p) = O(p + log(N) - log(p)) \tag{3.28}$$

– $T_{(A)}$ - the time of the assembly of LFMs. The LFM size is $(p+1) \times (p+1)$ so the assembly time is $O(p^2)$ since all matrices are generated concurrently by separate threads.

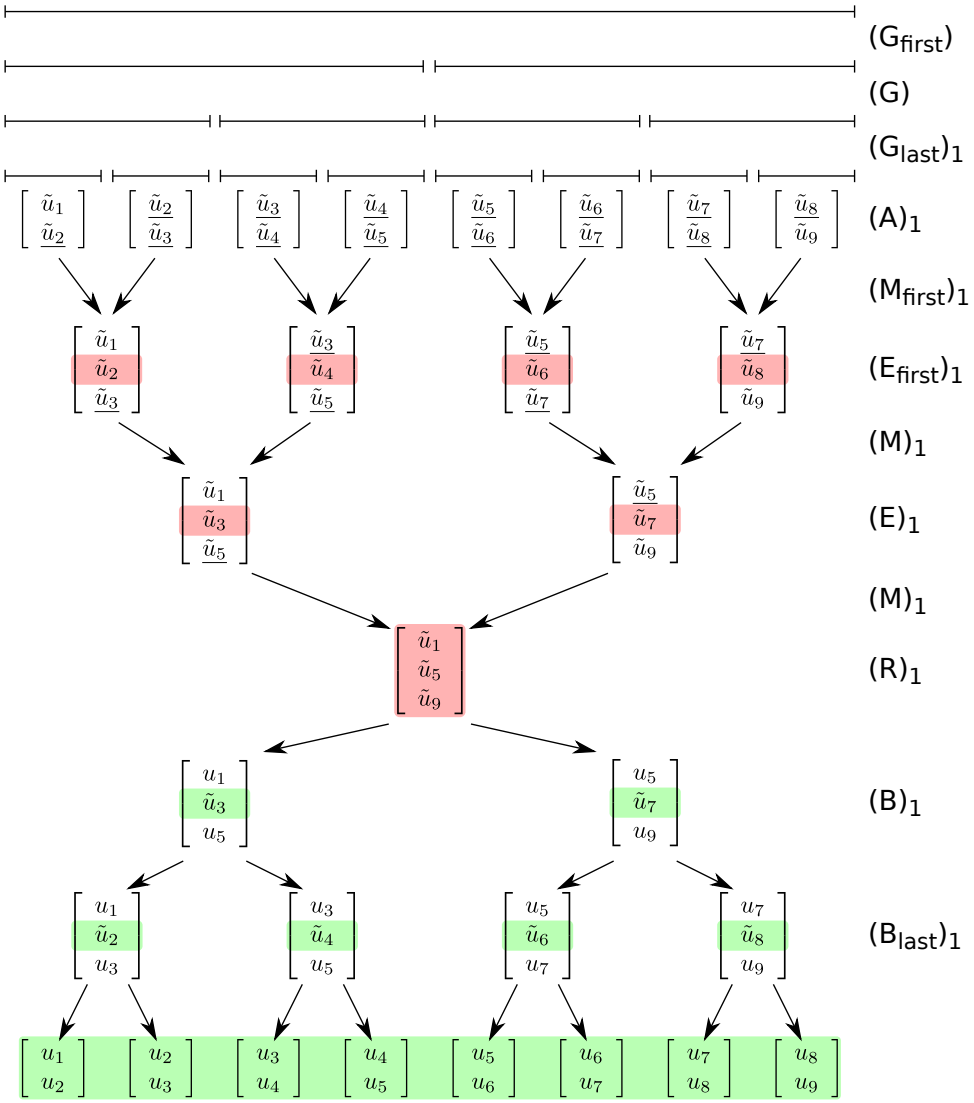

FIGURE 3.14: SOLVER EXECUTION FOR LINEAR B-SPLINES

– $T_{(M,E)}$ - the time of the stiffness matrix decomposition. The situation here is similar to the first step – there is a single first level and $log_2(\frac{N}{p+1}) - 1$ intermediate levels. The time cost at the first level is the time of merging $p+1$ LFMs and eliminating single fully assembled degree of freedom (3.29):

$$T_{(M,E)first\ level} = T_{merge} + T_{elimination} = (p+1)O(p^2) + 2pO(2p+1) = O(p^3) \quad (3.29)$$

... and at the intermediate levels it is the time of merging two ILFMs and eliminating p fully assembled degrees of freedom (3.30):

$$T_{(M,E)intermediate\ levels} = \sum_{i=1}^{log_2(\frac{N}{p+1})-1} \left(O(p^2) + pO(p^2)\right) = O(p^3(log(N) - log(p))) \quad (3.30)$$

– $T_{(R)}$ - the time of the single production in the root of the elimination tree. This is the time of the full Gaussian elimination: $O(p^3)$

– $T_{(B)}$ - the time of the backward substitution. At all $log_2(\frac{N}{p+1}) - 1$ intermediate levels there are p rows of ILFM calculated and at the last level there is a single row of LFM calculated (per each instance of the (B) production). So the time of this step is:

$$T_{(B)} = \sum_{i=1}^{log_2(\frac{N}{p+1})-1} pO(p) + O(p) = O(p^2(log(N) - log(p))) \tag{3.31}$$

Summing up, the computational cost can be estimated as follows (3.32):

$$T_{total} = O(p^3(log(N) - log(p))) \tag{3.32}$$

Which for the constant degree of the B-spline basis function is (3.33):

$$T_{total} = O(log\ N) \tag{3.33}$$

The graph grammar model does not assume anything about the number of processing units used for the solver execution. However, $O(log\ N)$ execution time can be achieved only when all graph grammar productions at each level can be executed concurrently. This means that the number of concurrent tasks and the number of elements in the computational domain must be of the same order of magnitude. This puts quite serious restrictions on the size of the problem that can be solved by real implementation.

Nowadays it is possible to use supercomputers with hundreds of thousands of cores which would allow to achieve high performance even for big problems. However, not everybody is able to gain access to such machines so more common solution is needed. GPUs are the best alternative. Hardware is affordable for everybody and capabilities in parallel processing of data can be compared to several-hundred-core computer. As shown in further chapters GPU can be used to simulate situation when there are several thousands of concurrently running threads, which is enough for the sake of this thesis. Moreover, when the number of threads is not big enough, by the means of the parallel processing the GPU implementation of the solver is still much faster than its ordinary CPU counterpart.

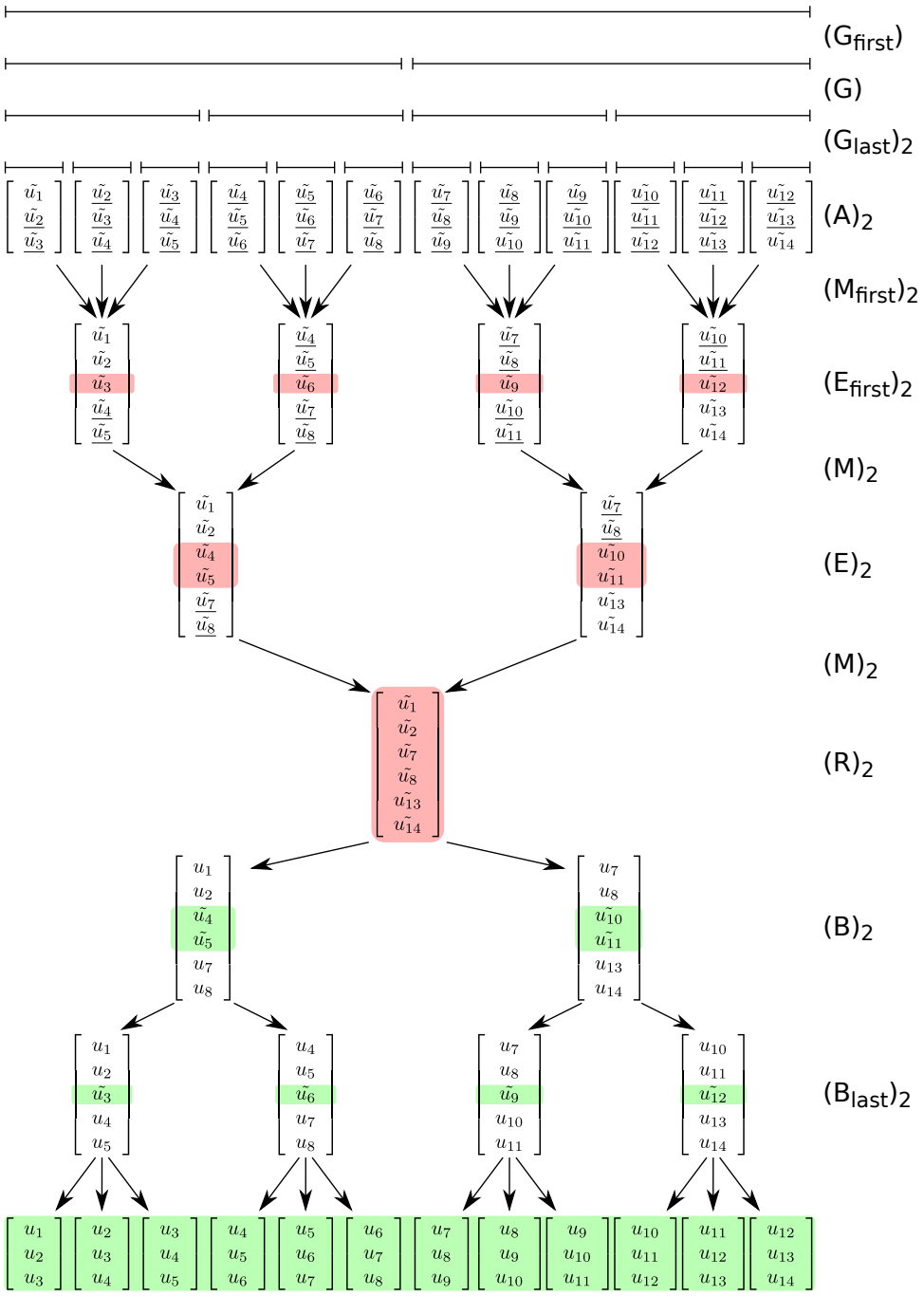

FIGURE 3.15: SOLVER EXECUTION FOR QUADRATIC B-SPLINES

4. Solution and its Implementation

This chapter contains step-by-step explanation of the graph grammar-based multifrontal solver implementation. To achieve the highest performance, the GPU programming model requires developer to carefully plan every single part of the software. The most important aspects are:

- data layout in global memory
- shared memory utilization
- GPU registers utilization

Everywhere where applicable it is explained how these aspects affected algorithm design.

4.1. Basis functions and derivatives values evaluation

Among their great advantages, B-splines basis functions have also some properties that make them quite complicated to use. Recursive Cox-de-Boor formula works perfectly for mathematicians. However, it is useless in machine computations when applied explicitly.

Figure 4.1 shows recursive evaluation for two consecutive basis functions of fifth order. Note that a single invocation of $N_{1,5}$ results in several invocations of lower order functions (e.g. two invocations of $N_{2,3}$, three invocations of $N_{2,2}$, etc.). This duplication results in a serious performance downgrade in a comparison to the other kinds of basis functions. What is more, when consecutive B-splines are evaluated they also share common lower order B-splines. Green area in the figure 4.1 highlights lower order functions which are common to these two basis functions. Situation gets even more complicated when it comes to derivatives. Figure 4.2 shows evaluation of cubic B-spline derivative. Also in this case **values** of B-splines are calculated again.

When Cox-de-Boor formula is applied explicitly the time cost of calculating $N_{k,p}(x)$ is approximately $O(2^p)$. This approach is obviously unacceptable and needs a few optimizations. Let us take a look at the weak formulation:

$$\sum_i a_i \, b\left(N_{i,p}\left(x\right), N_{j,p}\left(x\right)\right) = l\left(N_{j,p}\left(x\right)\right), \forall j \tag{4.1}$$

$$b(u,v) = \int_\Omega \left(a \frac{dv}{dx}\frac{du}{dx} + bv\frac{du}{dx} + cvu \right) \, dx + \beta u(l) v(l) \tag{4.2}$$

$$l(v) = \int_\Omega f v dx + \gamma v(l) \tag{4.3}$$

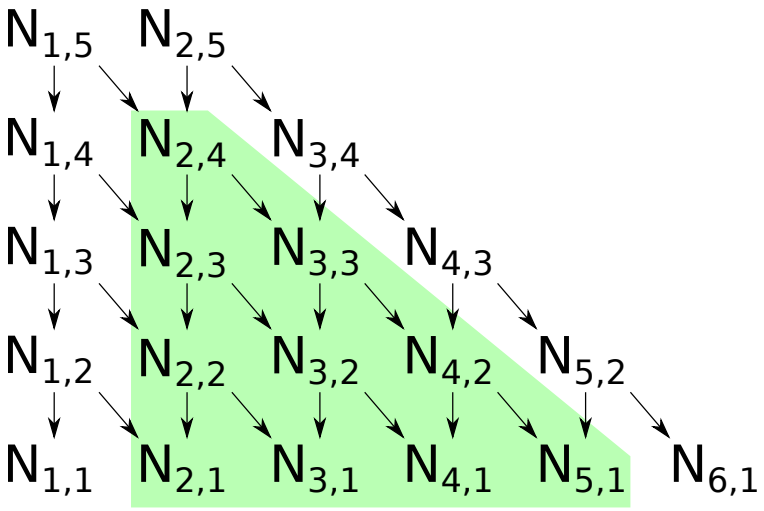

FIGURE 4.1: RECURSIVE B-SPLINE *(quintics)* EVALUATION

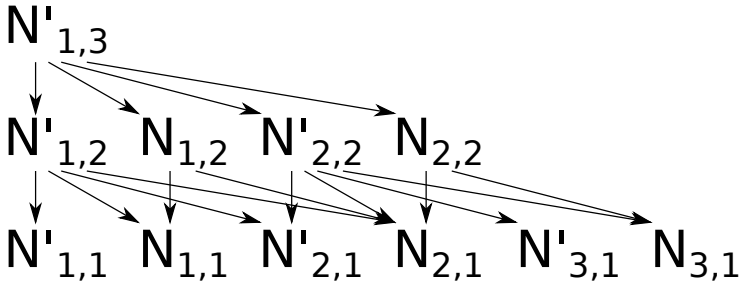

FIGURE 4.2: RECURSIVE B-SPLINE DERIVATIVE *(cubic)* EVALUATION

In formulae 4.2 and 4.3 Both b and l contain integrals over the entire domain. Those integrals are calculated using Gaussian quadratures. The integral becomes a weighted summation over the function values in the quadrature points:

$$b(u,v) = \sum_G \left(w_G a(x_G) \frac{dv}{dx}(x_G) \frac{du}{dx}(x_G) + b(x_G) v(x_G) \frac{du}{dx}(x_G) + c(x_G) v(x_G) u(x_G) \right) + \beta u(l) v(l) \tag{4.4}$$

$$l(v) = \sum_G w_G f(x_G) v(x_G) + \gamma v(l) \tag{4.5}$$

From the computational point of view there is no need to have a formula for basis functions or their derivatives. The only thing which is required are values of basis functions and derivatives in the quadrature points. This means that after choosing a domain partitioning and basis functions it is possible to calculate needed data prior to launching the multifrontal solver. This is the first optimization. Time cost of calculating $N_{k,p}(x_G)$ in formulae 4.4 and 4.5 is reduced from $O(2^p)$ to $O(1)$.

The problem that is still present is this initial evaluation of basis functions and their derivatives in the quadrature points. For n basis functions of order p and g quadrature points per element this

is time cost of $O(n^2 g 2^p)$. The second optimization is to note that B-spline basis of order p has non-zero values only over $(p+1)$ elements. The time cost is reduced to $O(ng(p+1)2^p)$ by evaluating B-splines only for those elements.

The third and the most valuable optimization is to eliminate this 2^p factor from the cost estimation. The idea is to calculate values of basis functions and their derivatives in the quadrature points in a step-by-step manner(or "degree-by-degree"). This means that values are calculated incrementally and stored in memory – first of all the values of all $N_{k,0}$ and $N'_{k,0}$, then using Cox-de-Boor formula higher degrees. When calculating $N_{k,p}(x_G)$ needed values of $N_{k,p-1}(x_G)$ and $N_{k+1,p-1}(x_G)$ do not need to be calculated recursively, they are just retrieved from the memory in time $O(1)$. This way n basis functions of order p and g quadrature points per element can be calculated in time $O(ng(p+1)p)$. Whole procedure is illustrated in pseudo-code 4.1.

Algorithm 4.1 Basis functions and derivatives (P-degree) calculation in quadrature points

Require: P – B-spline degree
Require: n – number of elements in domain
Require: g – number of quadrature points per element
  for $k = 1$ to $n + P + 1$ **do**
    for all x_G quadrature points in $N_{k,0}$ support **do**
      $N_{k,0}(x_G) \leftarrow 1$
      $N'_{k,0}(x_G) \leftarrow 0$
    end for
  end for
  for $p = 1$ to P **do**
    for $k = 1$ to $n + P + 1$ **do**
      for all x_G quadrature points in $N_{k,p}$ support **do**
        $N_{k,p}(x_G) \leftarrow$ Cox-de-Boor$(N_{k,p-1}(x_G), N_{k+1,p-1}(x_G))$
        $N'_{k,p}(x_G) \leftarrow$ Cox-de-Boor$'(N_{k,p-1}(x_G), N_{k+1,p-1}(x_G))$
      end for
    end for
  end for

GPU implementation

After applying the three improvements to basis functions and derivatives calculation it is time to take a look at the GPU implementation of this algorithm. It is worth to note that this algorithm can be easily parallelized. At each "level" p calculation of $N_{k,p}(x_G)$ and $N'_{k,p}(x_G)$ can be run independently for all values of k, and x_G. This means that running n threads it is possible to compute all needed values in time $O(g(p+1)p)$.

To achieve high performance and lower memory usage for memorized data there where two tricks applied:

– **"in-place" calculation** – when values of basis functions and derivatives are calculated with Cox-de-Boor formula at p level they use only values from $p-1$ level. This means that calculating values of $N_{k,2}$, values of $N_{k,0}$ are no longer needed and can be replaced be entries of $N_{k,2}$. Using this technique one is able to calculate values of arbitrary order p B-spline with buffer of constant size.

– **special memory layout** – as mentioned in earlier chapters global memory access should be coalesced for best GPU power utilization. Consecutive threads should access consecutive addresses in the global memory. This holds for both – reads and writes. In this implementation one thread is responsible for calculating all values for one basis and its derivative. To assure coalesced access, those values are grouped in memory by index of quadrature point (not by function index) as shown in figure 4.3. Each of n basis function is stored as $g \times (p+1)$ quadrature points x_{Gi} where $i = 0...(g(p+1)-1)$.

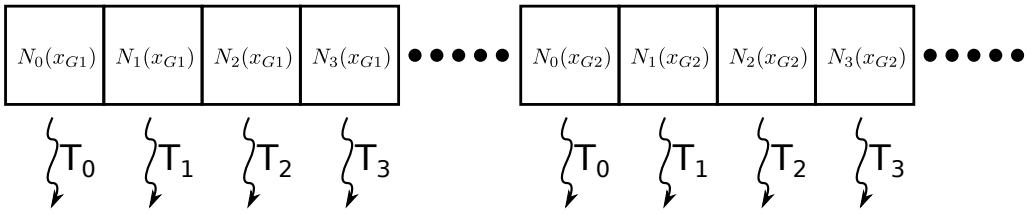

FIGURE 4.3: MEMORY LAYOUT FOR VALUES OF B-SPLINES BASIS FUNCTIONS

After completion of this step, global memory contains values for all basis functions and their derivatives in Gauss-Legendre quadrature points.

4.2. Generation of local frontal matrices

Generation of LFMs is basically applying formula 4.4 using data computed in previous step. Important thing to remember here is that Gauss-Legendre quadratures are defined for $(-1;1)$ interval, so when calculating b all quadrature weights have to be scaled for a given element length. Time cost of calculating b_{ij} can be estimated by $O(g(p+1))$ - this is the time of calculating the sum of $g \times (p+1)$ entries each of which takes time $O(1)$.

GPU implementation

In this implementation one thread is responsible for calculating one entry in LFM (b_{ij}). However, this is not simple procedure. Again coalesced memory access plays huge role here. To prepare LFMs for next steps (merging) data has to be put into memory in a way that will ensure the best performance. All matrices are divided into $p+1$ groups such that LFM_i goes to group m_t where $t = i \ mod \ (p+1)$. After completion of this step the global memory contains all entries of all LFMs grouped by:

1. group number m_t

2. inside groups by cell number $i = 0...(p+1)^2$

The layout is shown on the figure 4.4 (for linear B-splines) and on figure 4.5 (for quadratic B-splines). In the next section it will become clear why this layout is so efficient.

The generation step is executed by $(p+1) \times (p+1)^2$ groups of $\frac{n}{p+1}$ threads. Assuming that all threads run at once, time cost of this step is $O(g(p+1)^4)$.

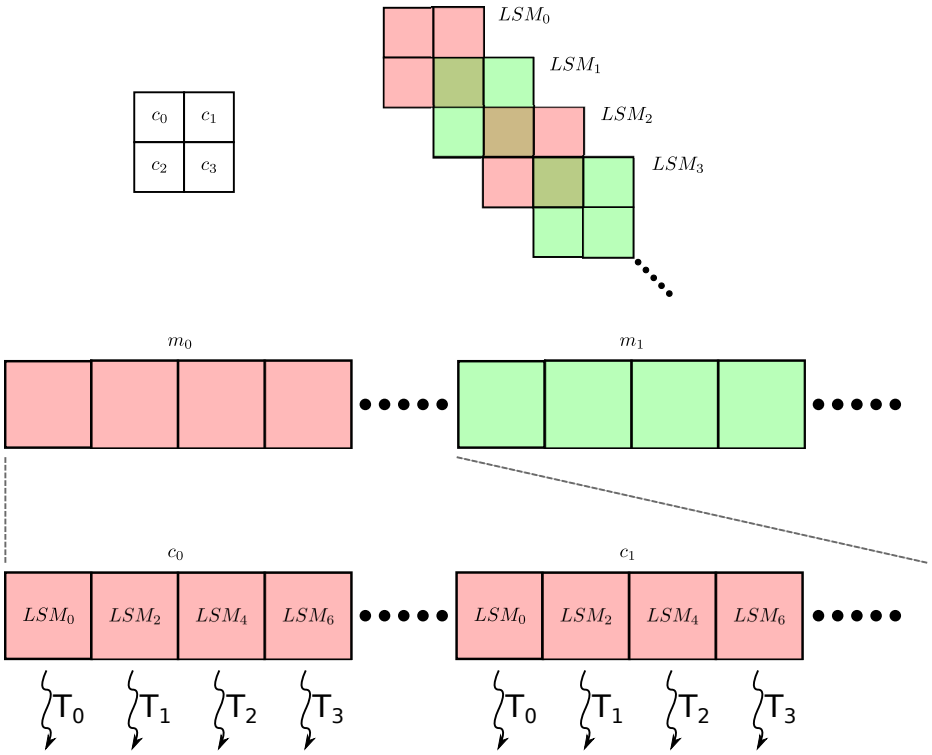

FIGURE 4.4: MEMORY LAYOUT FOR LFMS FOR $p = 1$

4.3. Matrix "decomposition"

This step is called matrix decomposition because it is analogical to LU matrix decomposition. When matrix M is factorized to lower triangular L and upper triangular U parts, they can be applied for multiple RHSes to acquire a linear system solution. This is very important since decomposition is the most time consuming part of the whole multifrontal solver – instead of factorizing M for each RHS it is factorized once and then it is used for an arbitrary number of RHSes. The final processing of RHSes with L and U consists of the forward and backward substitution. In this aspect this implementation of the multifrontal solver is slightly different than the graph grammar model introduced in the previous chapter suggests. Instead of merging FMs and RHSes at each level of elimination tree, only FMs are merged and data needed for RHS transformation (forward and backward substitution) is stored in a separate place in memory.

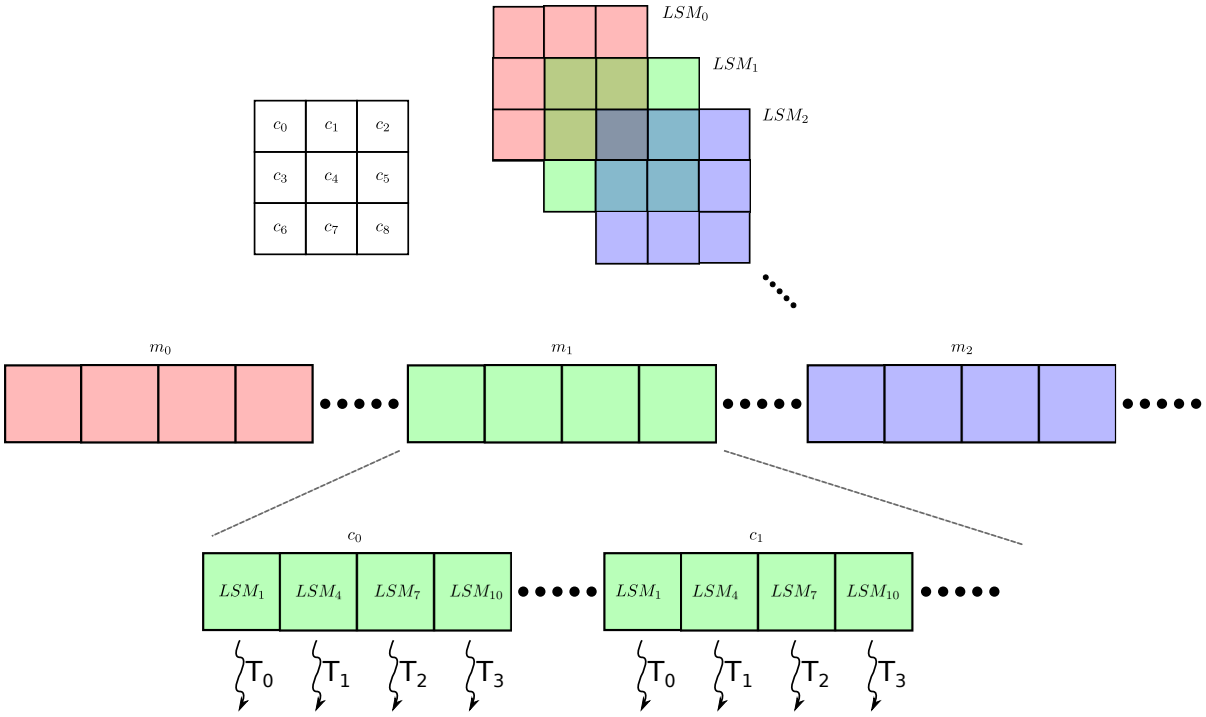

FIGURE 4.5: MEMORY LAYOUT FOR LFMS FOR $p = 2$

4.3.1. First merge

This step is an implementation of the graph grammar productions $(M_{first})_p$ and $(E_{first})_p$. Depending on B-spline basis function degree p it merges $p + 1$ local frontal matrices created in previous step. After this merge exactly one fully assembled degree of freedom is eliminated.

GPU implementation

From the GPU programming point of view there are two very important aspects which influenced the implementation of this step:

1. **shared memory** – merging step includes multiple summations, matrix rows and columns pivoting, also multiplications and subtractions. All these operations need to be as fast as possible so they are carried out in a GPU shared memory. Local frontal matrices are small enough to be stored in the shared memory. After completion of all heavy computations, data is stored back to the global memory.

2. **coalesced access to the global memory** – after previous step, LFMs are stored in the global memory in such a way that consecutive threads always read data from consecutive addresses in the global memory. What is more, after merging and elimination, data is written to the global memory also in a coalesced way.

In this step one thread is responsible for loading data from the global memory (merging), eliminating row corresponding to the fully assembled degree of freedom and writing new IFM back

to the global memory. Before writing data to the global memory, the matrices are again divided into groups. This time there are always only two groups since at next levels of the elimination tree always two matrices are merged. Whole procedure for a single thread is stated as a pseudo-code 4.2.

Algorithm 4.2 Thread procedure for graph grammar production $(M_{first})_p$

Require: $tidx$ – thread index
Require: N – number of LFM after this step
Require: P – B-spline basis degree
shared memory $M_{tidx}[2P+1][2P+1] \leftarrow$ initialize with 0
for $p = 0$ to $P-1$ **do**
for $c = 0$ to $(P+1)^2 - 1$ **do**
Read $tidx$ entry from global memory (cell c from group m_p) and store in M_{tidx}
end for
end for
Eliminate fully assembled row from M_{tidx}
Write first column to global memory. (For forward substitution)
Write first (eliminated) row to global memory. (For backward substitution)
synchronize with other threads
if $tidx < \frac{N}{2}$ **then**
Write Schur complement of M_{2*tidx} to global memory (group m_0)
else
Write Schur complement of $M_{2*(tidx-\frac{N}{2})+1}$ to global memory (group m_1)
end if

Time cost of this single thread procedure is a cost of:

- merging $O((p+1)^3) = O(p^3)$
- elimination of the fully assembled row $O((2p+1) + (2p)^2) = O(p^2)$
- writing data to the global memory $O(2p + 2p + (2p)^2) = O(p^2)$

So assuming that all threads run concurrently, time cost of this step is $O(p^3)$

4.3.2. General merging step

This step is an implementation of the graph grammar productions $(M)_p$ and $(E)_p$. Independently of the B-spline basis function degree p, always two matrices are merged. The thing that depends on p is the length of FM row, which is $3p$. This is a generic merging step for all levels of the elimination tree besides the first and the last one. The only thing that changes between consecutive runs of this step is the number of processed FMs.

GPU implementation

Analogically to the previous step this one extensively uses shared memory and the effective memory layout introduced earlier. Another optimization introduced here is the swapping buffer scheme used in the B-splines values calculation – Schur complements from the current level of the elimination tree are stored in the same place where they were stored two levels before. As in the previous step one thread is responsible for merging, elimination and storing data back to the global memory. The size of the merged matrix is $3p \times 3p$, p fully assembled rows are eliminated, p first columns are saved for the forward substitution, the Schur complement size is $2p \times 2p$. Whole procedure is depicted on the figure 4.6.

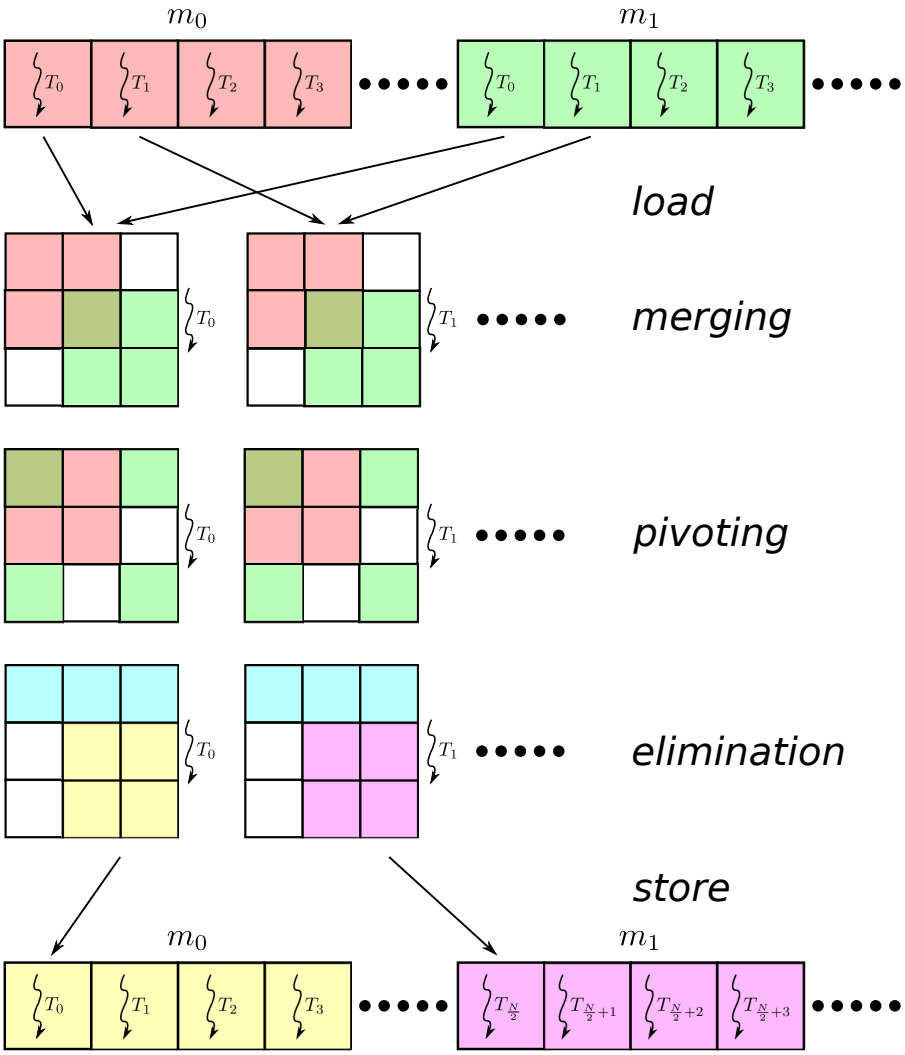

FIGURE 4.6: ONE STEP OF GENERAL MERGING STEP. NOTE THAT DATA IS READ AND PROCESSED BY ONE THREAD BUT WRITTEN BY ANOTHER

The time cost of a single run of this step by a single thread is:

- merging $O(2(2p^2)) = O(p^2)$
- elimination of the fully assembled rows $O(p(3p)^2) = O(p^3)$
- writing data to a global memory $O(3p^2 + 3p^2 + (2p)^2) = O(p^2)$

Assuming that all threads run concurrently, the time cost of this step is $O(p^3)$ at one level of the elimination tree. In total there is $log\ N$ levels so the cost for all intermediate levels is $O(p^3 log\ N)$

4.3.3. Final merge

This step is an implementation of the graph grammar production $(R)_p$. After the merge in this step all degrees of freedom are fully assembled and therefore can be eliminated. The size of resulting matrix depends on the B-spline degree and it is $3p \times 3p$ as in the previous step. After completion of this step all data required to solve the linear system is stored in the GPU global memory.

GPU implementation

This step differs from the previous only in to aspects:

1. Procedure is run by a single thread. Since only two matrices are merged there is no need to run more than one thread. Even if more threads were engaged in the Gaussian elimination it would not buy a better performance because of required synchronization.

2. When all degrees of freedom are fully assembled it is required to perform full Gaussian elimination. This is actually LU decomposition of frontal matrix. L part is stored for forward substitution and U is stored for backward substitution.

The time cost of this final step is equal to the cost of full Gaussian elimination. It is $O((3p)^3) = O(p^3)$.

4.4. Applying decomposed matrix to RHSes

If RHS vector was processed according to graph grammar model together with the matrix, it would be sufficient to perform recursive backward substitution using fully assembled rows from previous step. However, this would allow to solve only one equation (one RHS). When processing RHS is separated from the matrix factorization the great GPU abilities to parallel data processing can be utilized to solve multiple equations at the same time.

What is more, when only one RHS is processed GPU does not use its full potential. Synchronization is required at each level of elimination tree, so idle cores instead of waiting can run threads which process other RHSes.

4.4.1. Forward substitution

This step mirrors merging and elimination that would be done if RHS was processed along with the matrix. However, the procedure is extended for multiple RHSes. Similarly to ordinary LU decomposition, once factorized matrix can be utilized for the arbitrary number of RHSes.

GPU implementation

To make this implementation efficient and to utilize maximum GPU power, following optimizations were introduced:

1. **rectangular thread blocks** – threads were organized in rectangular blocks of size $x \times y$. To assure coalesced access to a global memory x is the number of RHSes processed by single block and y is the number of "merging" tasks performed by this block. When one block of threads is used for multiple RHSes it allows use of ...

2. **shared memory for common data** – data computed in previous step (merging frontal matrices) now used for forward substitution is the same for all threads in a block. Since shared memory is much faster than global memory and there is no enough GPU registers to store factorized frontal matrices, shared memory comes as a perfect solution in this case.

3. **coalesced global memory access** – x dimension of the block was chosen to process different RHSes to ensure that threads with growing x coordinate and the same y coordinate will access (read and write) consecutive addresses in a global memory. This is depicted on the figure 4.7.

After completion of this step the global memory contains RHS after forward substitution as if each of them was processed along with the frontal matrices according to the graph grammar model. Assuming that all r RHSes are processed at once, thanks to the introduced optimizations, the time cost of this operation is reduced from $O(rp^2 log\ N)$ to $O(p^2 log\ N)$.

4.4.2. Backward substitution

This step is an implementation of the graph grammar productions $(B)_p$ and $(B_{last})_p$. It is generalized to higher number of RHSes in a similar way it was done for the forward substitution.

GPU implementation

Implementation of this step makes use of the same optimizations that were applied in the forward substitution, i.e.:

1. **rectangular thread blocks** – blocks of size $x \times y$ (number of RHSes $\times$ number of splitting tasks)

2. **shared memory for common data** – assembled rows merged and eliminated during the matrix decomposition are the same for all RHSes. Therefore, they can be stored in fast shared memory.

3. **coalesced global memory access** – threads with consecutive x indexes access consecutive addresses in the global memory. See figure 4.7.

Similarly to the forward substitution, cost of this step for r RHSes is $O(p^2 log\ N)$ assuming that all vectors are processed at the same time.

After completion of this step global memory contains the final solution to the stated problem (problems if multiple RHSes were used) – coefficients u_i for B-spline basis functions. This vector is a solution to the differential equation in the finite dimensional space (with B-spline basis) chosen at the beginning.

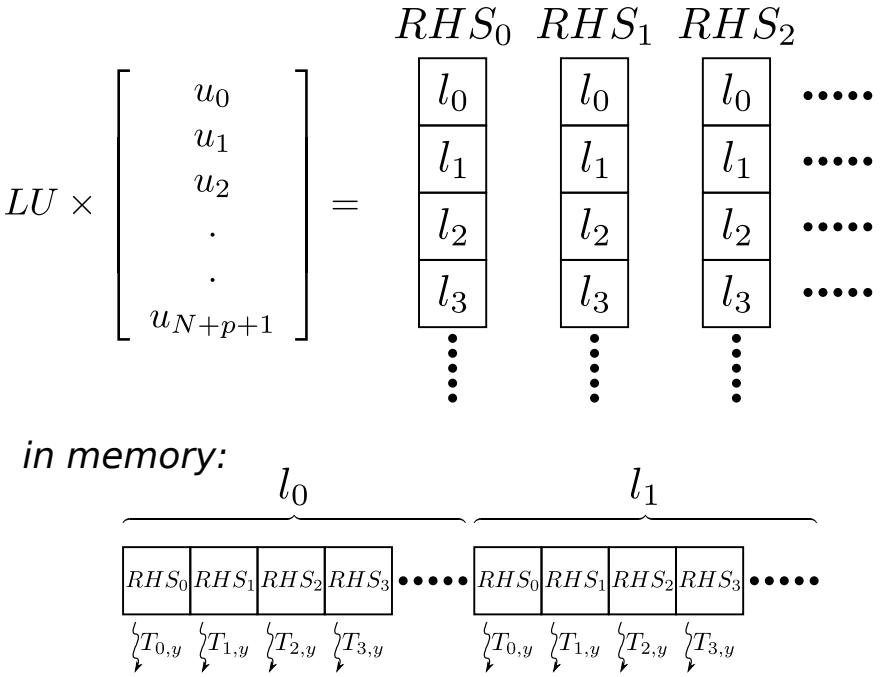

FIGURE 4.7: MEMORY LAYOUT FOR MULTIPLE RHS VECTORS ASSURING COALESCED ACCESS TO A GLOBAL MEMORY

4.5. Obtaining the solution and calculating the error

The final part of the solver implementation is to gather all computed coefficients and check whether the obtained solution is correct.

4.5.1. Solution

As stated before the approximate solution to the differential equation is a linear combination of basis functions (B-splines in this case). To obtain the value of resulting function u in a point x one has to evaluate the following formula 4.6:

$$u(x) = \sum_{i=0}^{N+p} u_i(x) N_{i,p}(x) \tag{4.6}$$

Calculating this for numerous points might be really expensive. The cost of the entire computation is a combination of:

1. cost of calculating the value of a B-spline in a point, which was estimated in previous sections to be roughly $O(2^p)$.

2. cost of summing up $N + p + 1$ products of coefficients and values of basis functions in a point.

Summing up, the total time cost of evaluating $u(x)$ for m points can be estimated as $O(m(N + p + 1)2^p)$. This is unacceptable, because despite having robust way to solve differential equation it would be extremely difficult and ineffective to use this solution. Even if there was a single thread per each point time complexity is still too big – $O((N + p + 1)2^p)$. Luckily, there are two optimizations that can be incorporated to avoid this problem:

1. Note that for a single point there is at most $p+1$ basis functions which have non-zero value. Situation is illustrated on the figure 4.8 for quadratic B-splines. Basis functions drawn in red are the only ones that are significant part of sum 4.6.

2. If there are multiple points to be calculated, scattered across whole domain it might reasonable to follow "step-by-step" or "degree-by-degree" procedure shown in chapter 4.1.

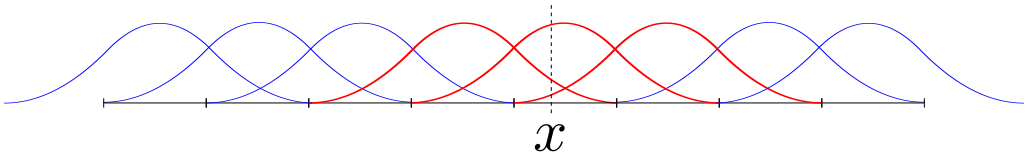

FIGURE 4.8: THREE NON-ZERO QUADRATIC ($p = 2$) B-SPLINES FOR POINT x

Situation becomes even better because some values of basis functions have already been calculated. Values calculated in chapter 4.1 are perfect for this purpose. They can be used for example for plotting final solution. In this implementation one thread is responsible for summing up $(p+1)$ products. This way the time cost of calculating $m = gN$ values is reduced from $O((N+p+1)2^p)$ to $O(p)$ if m threads run concurrently.

4.5.2. Error calculation

Error calculation serves two important functions:

1. **verification of obtained solution** – in case there is known analytical solution to the differential equation. This is very useful method of finding whether implementation works as designed.

2. **stop condition** - usually when analytical solution is unknown solver is executed for constantly growing number of elements. One can compare two consecutive solutions and check whether relative error is decreasing, then stop execution when relative error falls below initially defined level.

There are several measures for error estimation. The one implemented here is based on H^1 norm:

$$\|f\|_{H^1(\Omega)} = \sqrt{\int_\Omega (f^2 + f'^2) dx} \tag{4.7}$$

Error measure implied by H^1 norm is defined by formula 4.8:

$$E_{H^1}(f,g) = \|f-g\|_{H^1(\Omega)} = \sqrt{\int_\Omega ((f-g)^2 + (f'-g')^2) dx} \tag{4.8}$$

Note that derivative u' of the final solution u is required to calculate E_{H^1} error properly. Derivative of the final solution also can be represented as a linear combination of basis functions derivatives. What is more, coefficients calculated for basis functions are the same for their derivatives:

$$u' = \left(\sum_i u_i N_{i,p}\right)' = \sum_i (u_i N_{i,p})' = \sum_i u_i N'_{i,p} \tag{4.9}$$

To do the integration, Gauss-Legendre quadratures are used again. This means that values of basis functions and their derivatives calculated in chapter 4.1 can be reused to speed up the computation.

This paragraph ends entire description of the solver implementation. The next chapter gathers all results obtained from the parallel multifrontal solver. They are carefully analyzed and compared against theory and CPU implementation.

5. Results

The first part of this chapter contains all basic information about the hardware and the software used for the multifrontal solver testing. Data presented here is vital for a better analysis of the results presented later. Further parts contain the results, the execution time analysis and the comparison against the ordinary single threaded solver implementation for CPU.

5.1. Technical data

5.1.1. GPU data

The solver implementation created as an integral part of this thesis uses NVIDIA CUDA 4.0 API. Program output 5.1 shows the output of `deviceQuery` program distributed as a part of CUDA SDK. It contains all important information about the GPU device used for performing the multifrontal solver tests. The multifrontal solver code is tested on the NVIDIA GeForce 560 Ti device which compute capability is **2.1** and which supports both single and double precision computations. The device is equipped with 384 CUDA cores and 1GB of the global memory. The shared memory size is 48 kilobytes per multiprocessor.

The solver is ready to run also on older devices with the compute capability of **1.x**. However, they are equipped with smaller shared memory and might not support double precision computations. Smaller shared memory influences mainly the maximal degree of B-spline basis functions that can be used.

5.1.2. Host data

The host code is executed on a dual core machine with 4GB of RAM. Program output 5.2 shows all important data about the CPU. This data is mostly relevant for the GPU vs. CPU comparison. It is worth to note that the host processing unit is almost twice as fast as the graphical one. However, thanks to numerous cores, execution time for the GPU version of the solver is much shorter.

5.1.3. Used compilers

The CUDA program consists of the host code and the device(GPU) code. A source file is processed by **nvcc**. It generates a binary code for the GPU and delegates the compilation of the

Program output 5.1 `deviceQuery` output

```
$ deviceQuery
[deviceQuery] starting...
/usr/local/cuda4/computingSDK/C/bin/linux/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "GeForce GTX 560 Ti"
  CUDA Driver Version / Runtime Version          4.10 / 4.0
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1024 MBytes (1073283072 bytes)
  ( 8) Multiprocessors x (48) CUDA Cores/MP:     384 CUDA Cores
  GPU Clock Speed:                               1.67 GHz
  Memory Clock rate:                             2050.00 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
```

Program output 5.2 CPU data

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Duo CPU     E8500  @ 3.16GHz
stepping        : 6
cpu MHz         : 3166.297
cache size      : 6144 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
 dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm constant_tsc arch_perfmon pebs bts rep_good
 nopl aperfmperf pni dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm sse4_1 lahf_lm dts
 tpr_shadow vnmi flexpriority
bogomips        : 6332.59
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

... (and the same for 2nd core)
```

Program output 5.3 `nvcc` version

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_May_12_11:09:45_PDT_2011
Cuda compilation tools, release 4.0, V0.2.1221
```

Program output 5.4 `g++` version

```
$ g++ --version
g++ (GCC) 4.4.6
Copyright (C) 2010 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

host code to the ordinary compiler such as **gcc**.

Device code

The device code was written in *C++*. It was compiled using the NVIDIA compiler (**nvcc**) distributed as a part of the SDK. Program output 5.3 contains the description of this compiler. Some of the features of *C++* are not supported in the device code (e.g. virtual functions, function pointers, etc.). However, they were not necessary for the implementation of the solver. *C++* was chosen mainly because of the mechanisms of templates, classes and operators overloading. The resulting program is as fast as its *C* counterpart but the code is much easier to maintain.

Host code

The host code was also written in *C++*. It was compiled using **g++** compiler from GNU GCC family. Program output 5.4 contains the description of this compiler. **nvcc** requires that the host code is compiled with **gcc** version $\leq$ **4.4** to assure correct linking.

5.2. Numerical results

The most important aspect of a solver implementation is its correctness. The convergence of the solver is measured using H^1 measure against the ideal solution divided by H^1 norm of this ideal solution. The solution obtained by the solver is expected to converge to the ideal solution when the number of elements and the order of the basis functions is increased.

5.2.1. Benchmark problem

Following equation is used as a benchmark problem:

$$\text{Find } u \in C^2(\Omega) \text{ such that:}$$
$$\begin{cases} -\frac{d}{dx}\left(a(x)\frac{du(x)}{dx}\right) + b(x)\frac{du(x)}{dx} + c(x)u(x) &= f(x) \\ u(0) &= 0 \\ a(l)\frac{du}{dx}(l) + \beta u(l) &= \gamma \end{cases} \tag{5.1}$$

Where:

$$\Omega = [0, l] \tag{5.2}$$
$$a \in C^1(0, l) \tag{5.3}$$
$$b, c, f \in C^0(0, l) \tag{5.4}$$
$$\beta, \gamma \in \mathbb{R} \tag{5.5}$$

Following example solution was chosen:

$$u(x) = x\ sin(15x)cos(24x) \tag{5.6}$$

And following parameters:

$$l = 1 \tag{5.7}$$
$$a(x) = sin(x) \tag{5.8}$$
$$b(x) = x \tag{5.9}$$
$$c(x) = -x \tag{5.10}$$
$$\beta = 0 \tag{5.11}$$

The value of γ and the formula for the forcing vector f were obtained with a usage of above data to assure the correct solution:

$$\gamma = a(1)\frac{du}{dx}(1) + \beta u(1) \tag{5.12}$$
$$f(x) = -\frac{d}{dx}\left(a(x)\frac{du(x)}{dx}\right) + b(x)\frac{du(x)}{dx} + c(x)u(x) \tag{5.13}$$

5.2.2. Solution visualization

Figure 5.1 presents the solutions calculated by the multifrontal solver for various degrees of B-spline basis functions and various number of elements in computational domain. The ideal solution is presented for the comparison in the bottom right corner of figure 5.1. The solutions are represented as a set of points – the values in the quadrature points.

It is important to distinguish between the result of the solver and the visualization of this result. The result obtained from the multifrontal solver is a **function** – a linear combination of basis functions. Later on, it is possible to get a value of this function in any point of the computational domain without any additional calculations (i.e. interpolation, approximation or another solver execution). On the other hand, the results presented on figure 5.1 are nothing more than values of functions (solver results) in several points.

5.2.3. Convergence

The solution $\hat{u}$ converges to the ideal solution u if its error is decreasing. As mentioned earlier the error of the solution $\hat{u}$ is calculated as follows:

$$E = \frac{\|\hat{u} - u\|_{H^1}}{\|u\|_{H^1}} \tag{5.14}$$

To ensure that the quality of the solution is good and it is not affected by any method errors the calculation of the integrals in H^1 norm uses more quadrature points than is used for the solver algorithm. If g points are used for the solver then $g+1$ are used for the error calculation. For Gauss-Legendre quadratures this means that all points are different. Therefore, calculated H^1 norm will indicate whether the obtained solution approximates the ideal solution in the entire domain, not only in the quadrature points chosen for the the solver.

Figure 5.2 shows the convergence rates for basis functions of order $p = 1, 2, 3, 4, 5$. As expected, the higher is the order of basis functions the faster and better convergence is. The rate of

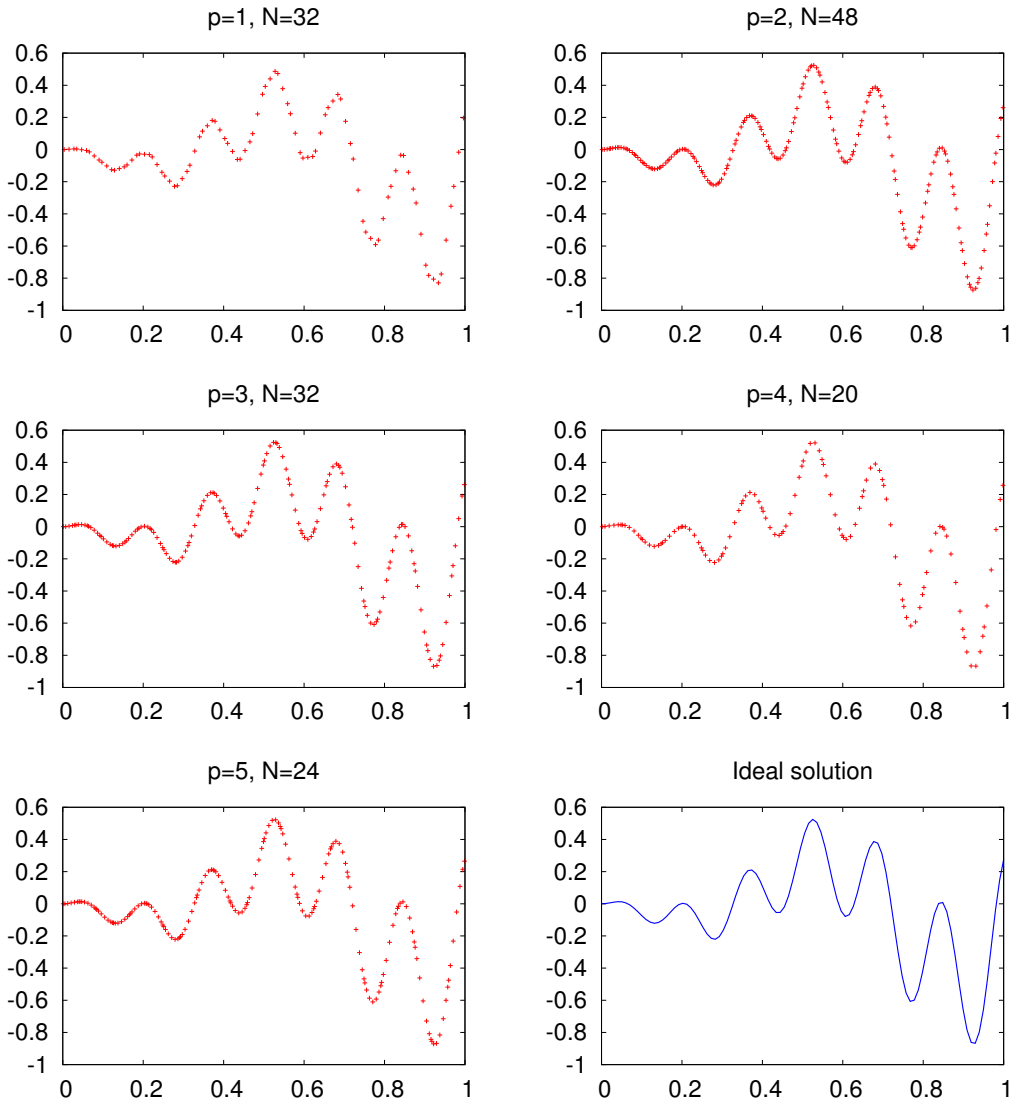

FIGURE 5.1: SOLUTIONS OBTAINED BY MULTIFRONTAL SOLVER FOR $p = 1, 2, 3, 4, 5$

convergence (the slope of the error function) is highest for $p = 5$ and lowest for $p = 1$. For quintic B-splines accuracy of 10^{-10} is achieved with usage of c.a. 1000 elements. For linear B-splines convergence is linear – doubling the number of elements, error gets twice smaller. Therefore, it would take $\approx 10^{10}$ elements to obtain accuracy given by B-splines of order $p = 5$ on 1000 elements. Note that the minimal error obtained by the solver reaches 10^{-11}. After this point, increasing the number of elements causes only bigger numerical error.

Figure 5.3 compares the convergence for all orders of basis functions ($p = 1, 2, 3, 4, 5$) in the interval where increasing the number of elements causes decreasing of the error. Note that for c.a. 1000 elements increasing the order of B-spline basis by 1 causes the error to fall over two orders of magnitude.

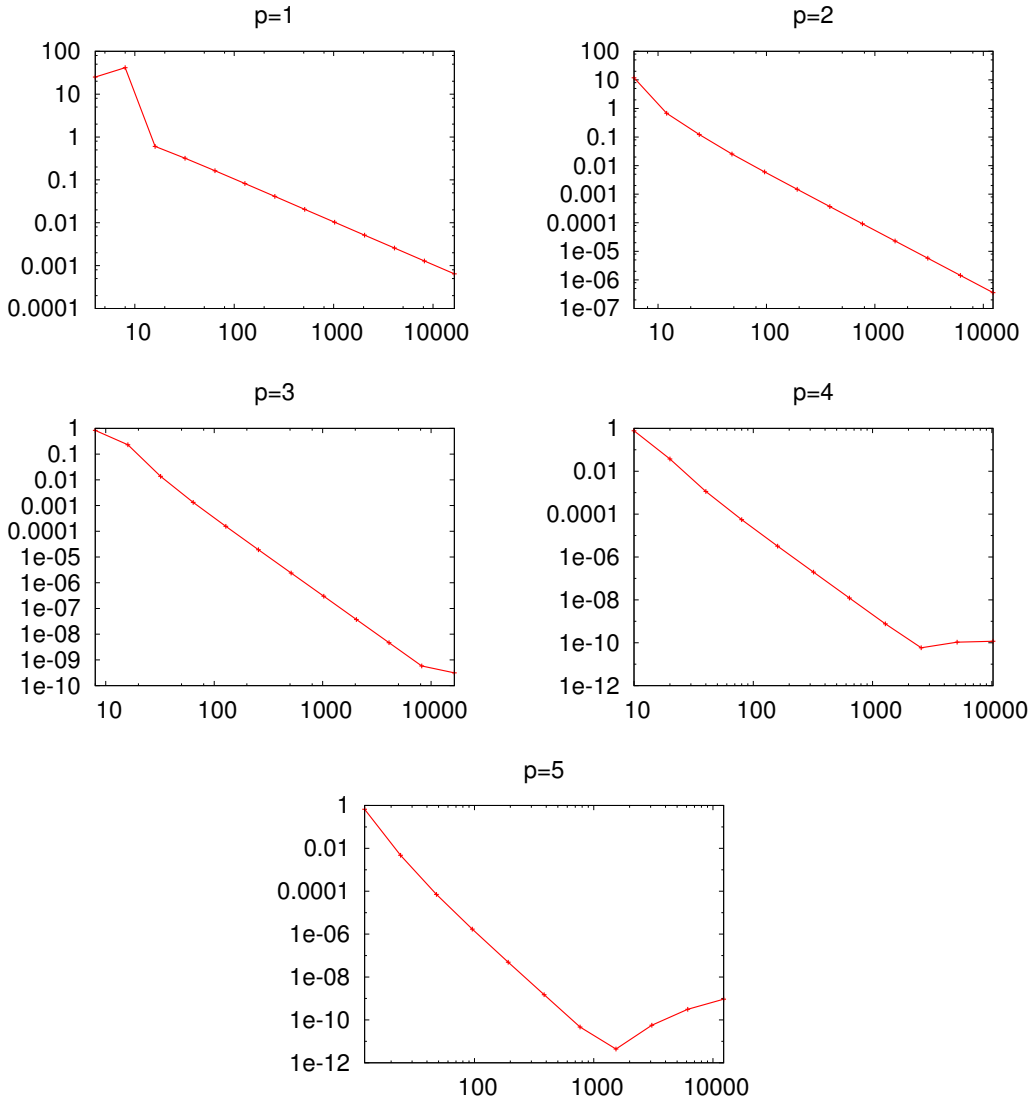

FIGURE 5.2: CONVERGENCE OF THE SOLUTIONS FOR GROWING NUMBER OF ELEMENTS ($p = 1, 2, 3, 4, 5$). X AXIS IS THE NUMBER OF ELEMENTS, Y AXIS IS THE CALCULATED ERROR E. NOTE LOGARITHMIC SCALE ON BOTH AXES.

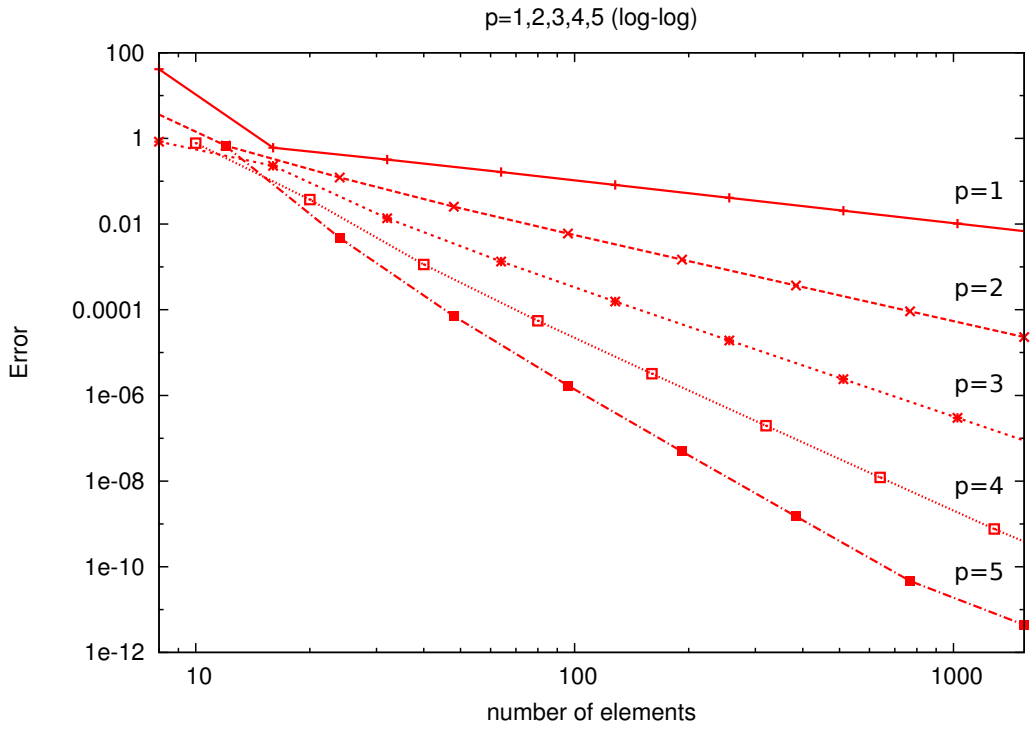

FIGURE 5.3: CONVERGENCE FOR ALL ORDERS OF BASIS FUNCTIONS IN INTERVAL OF DECREASING ERROR

5.3. Execution time analysis

The following section contains details on a solver execution time. The most important part is the verification whether the execution time of the solver is growing logarithmically with growing number of elements.

5.3.1. Assembly

This step is a combined calculation of basis functions in quadrature points and generation of the local frontal matrices. As predicted in the chapters 4.1 and 4.2 this step is independent of the number of elements in the domain. Figure 5.4 shows the assembly time for B-spline basis of order $p = 1, 2, 3, 4, 5$. Time is slightly growing for the increasing number of elements but this is caused by the GPU scheduler which has to manage more threads.

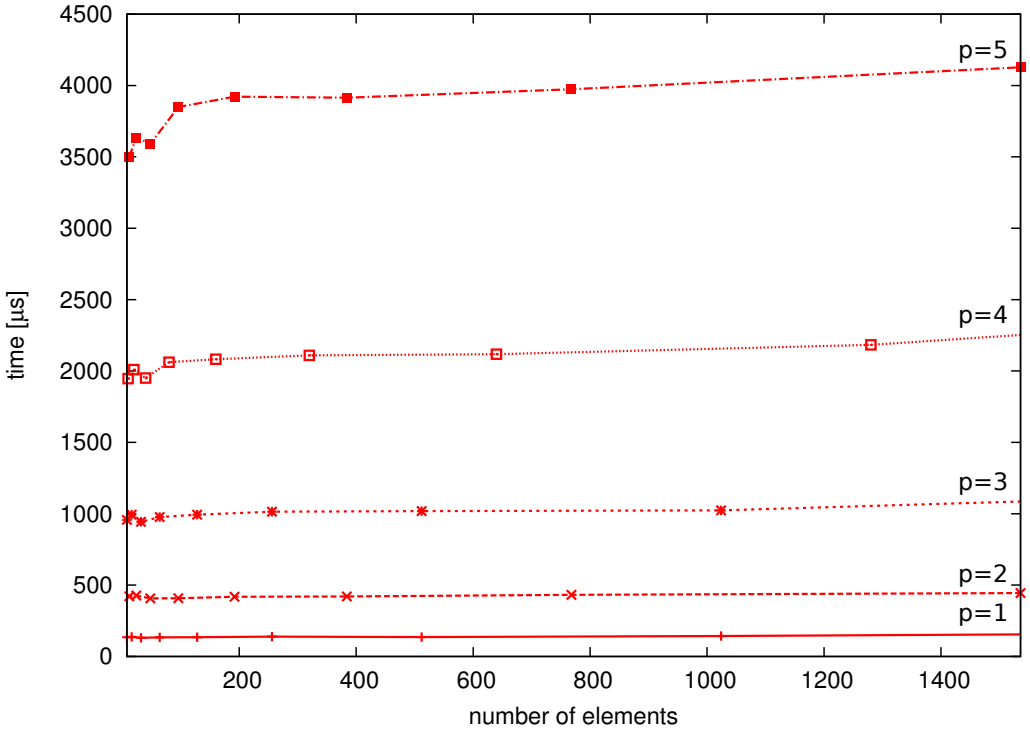

FIGURE 5.4: ASSEMBLY TIME FOR ALL ORDERS OF BASIS FUNCTIONS

5.3.2. Matrix factorization

This is the key step of the multifrontal solver – the matrix factorization. Execution time for this step is presented on figure 5.5. As expected execution time complexity for this step is $O(log\ N)$, where N is the number of elements.

The thing that needs a comment here is the result for $p = 4$ and $p = 5$. As described in the chapter 4.3, matrix decomposition makes heavy use of a shared memory. For basis functions of order $p = 4, 5$ LFMs get pretty big so whole problem has to be divided in a different way. In this case the number of threads in a block is decreased, which causes this improportional time increase.

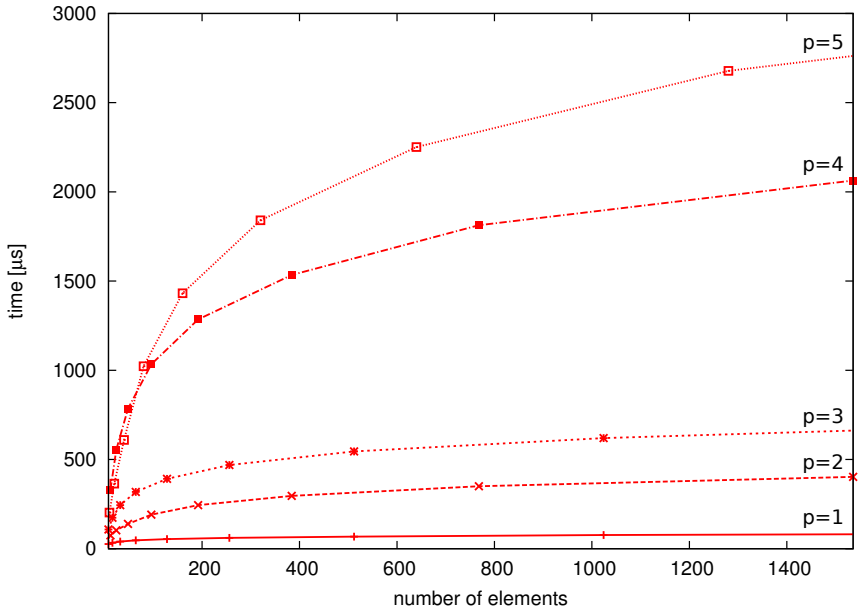

FIGURE 5.5: MATRIX FACTORIZATION TIME FOR ALL ORDERS OF BASIS FUNCTIONS

5.3.3. RHS processing

The final step is to apply the factorized matrix to RHS. Figure 5.6 presents the time results for B-spline basis $p = 1, 2, 3, 4, 5$ and single RHS. This step does not scale logarithmically in the provided interval. However, this is completely normal. The reason for this is that RHS processing step of the solver was optimized for multiple RHSes. When single RHS is processed thread blocks dimensions are $(1 \times y)$, which is not optimal in terms of the access to GPU global memory.

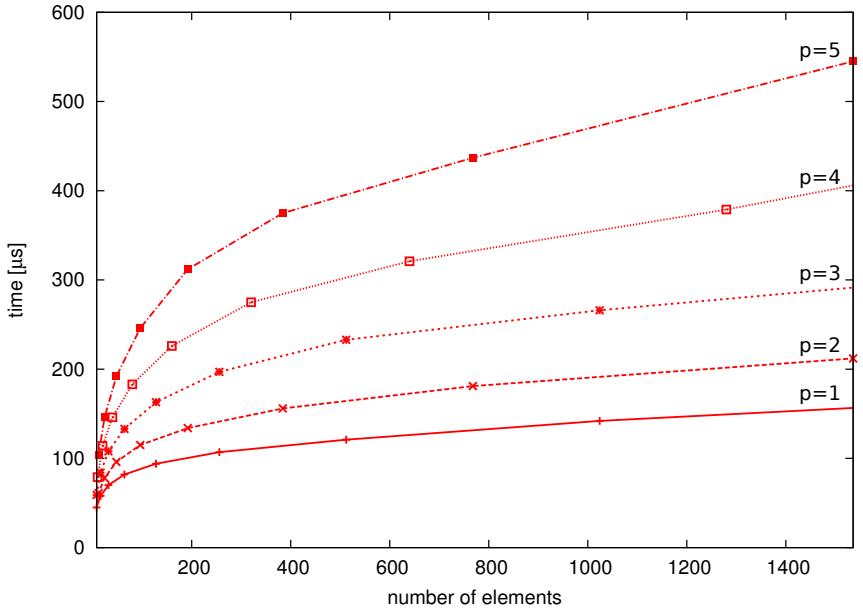

FIGURE 5.6: RHS PROCESSING TIME FOR ALL ORDERS OF BASIS FUNCTIONS AND ONE RHS

5.3.4. Multiple RHS processing

Solving multiple differential equations (with a common stiffness matrix and different forcing vectors) at the same time is the greatest strength of the implemented solver. Figure 5.7 presents execution time measured for basis functions of order $p = 1, 2, 3, 4, 5$. The number of elements in each case was equal $128(p + 1)$.

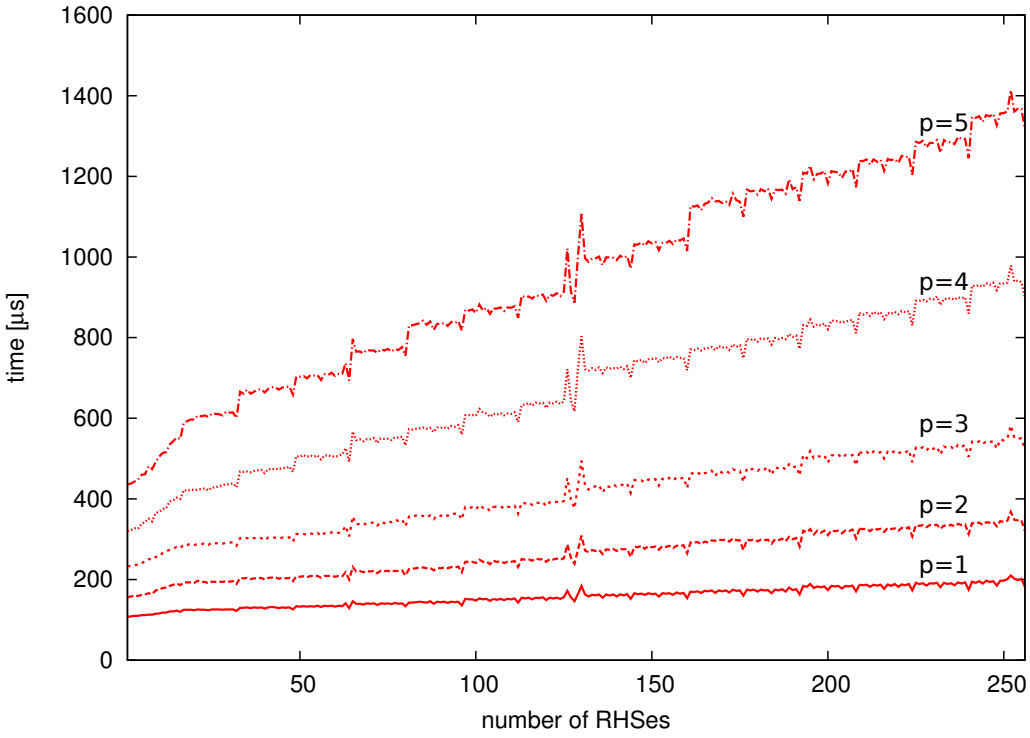

FIGURE 5.7: TIME FOR GROWING NUMBER OF RHSES

The plot on the figure 5.7 is interesting because of a few reasons:

– For lower order basis functions ($p = 1, 2, 3$) the cost of calculating 256 RHSes instead of 1 is close to none. For $p = 1$ its hardly tens of **microseconds**.

– Execution time does not grow uniformly – it forms steps, which length is 16. This means that every 16 RHSes the GPU device is required to allocate a new block of threads to process data.

– When the number of RHSes is divisible by 16 the solver runs faster. This means that GPU uses its full potential and does not need to handle idle threads in a special way.

5.4. Comparison against sequential solver

The final part of the result analysis is the comparison of two implementations of multifrontal solvers – the GPU implementation, which is the subject of this work, and CPU implementation created with a usage of well known libraries for linear systems solving. Despite logarithmic scaling of execution time, the GPU solver is also expected to work much faster than the ordinary sequential solver.

5.4.1. Sequential solver implementation details

The sequential solver created for CPU was implemented with a usage of **MUMPS** – *a MUltifrontal Massively Parallel sparse direct Solver* [ADKL01, AGLP06, ADL00]. MUMPS is an implementation of the multifrontal solver created in FORTRAN with C interface. It is widely used in the scientific community as it allows to quickly solve big systems of equations on multi-core clusters. However, MUMPS can also be executed on a personal computer in a sequential mode, on a single CPU. This is the mode which was used for the purpose of the following tests. MUMPS has been optimized for years and it is almost always taken as a reference when new implementation of a direct solver is tested. The version of MUMPS used for the tests was **4.9.2**.

5.4.2. Matrix factorization

Stiffness matrix factorization is the key part of every multifrontal solver. Figure 5.8 shows the comparison of execution times for the GPU solver and MUMPS. For lower order basis functions ($p = 1, 2$) GPU solver is faster from the beginning. For $p = 3, 4, 5$ it is faster for number of elements greater than c.a. 1000. This is caused mainly by the constant overhead of running functions on GPU which is estimated to be $\approx 10\mu s$ per call.

The profit from using GPU solver is quite obvious – for higher numbers of elements it is at least one order of magnitude faster than MUMPS (3 orders of magnitude for $p = 1$). Good time complexity of GPU solver allows it to be much more efficient for a larger number of elements.

5.4.3. RHS processing

When comparing GPU and CPU solver in RHS processing conclusions are similar to those for the matrix factorization. Figure 5.9 shows the comparison of time spent on RHS processing. MUMPS is faster for lower number of elements but for higher its linear time complexity makes it less efficient than the GPU solver. As profit for a single RHS is not big it gets much better for bigger number of RHSes which is discussed in next section.

5.4.4. Multiple RHS processing

This final comparison combines two biggest advantages of the GPU solver – fast matrix factorization and fast processing of multiple right hand sides. Figure 5.10 shows the result of this

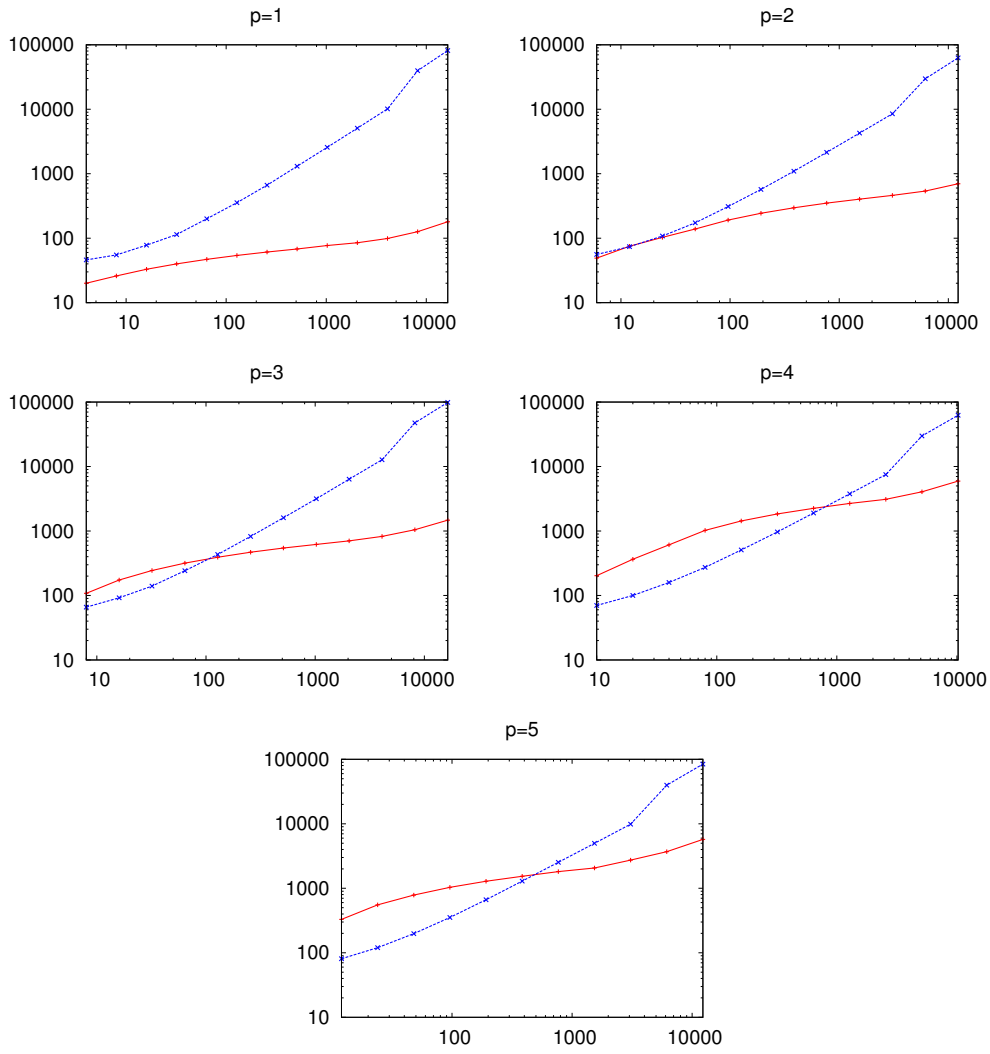

FIGURE 5.8: COMPARISON OF EXECUTION TIMES FOR MATRIX FACTORIZATION. BLUE PLOT IS FOR MUMPS AND RED IS FOR GPU SOLVER. X AXIS IS THE NUMBER OF ELEMENTS AND Y AXIS IS FACTORIZATION TIME IN **MICROSECONDS**. BOTH AXES ARE IN LOGARITHMIC SCALE.

comparison. Note that slopes of both plots (MUMPS and GPU solver) are getting parallel for bigger number of elements. The reason for this is that theoretical logarithmic time complexity holds only for small number of elements. Small number of elements allows to simulate situation presented in theoretical model where there is infinite number of processing units. When number of elements gets bigger time complexity becomes the same for both solvers. However, GPU abilities to concurrently process huge amounts of data give it advantage of one order of magnitude in execution time.

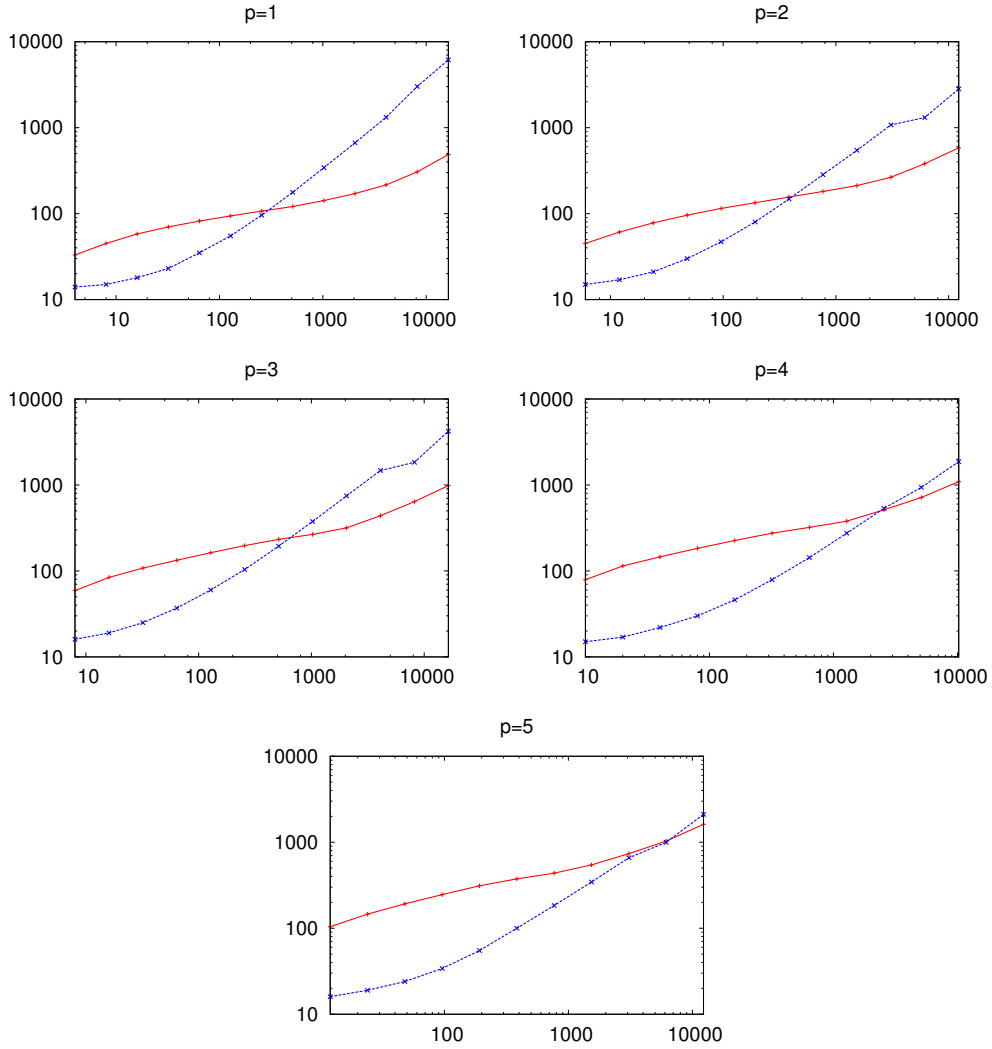

FIGURE 5.9: COMPARISON OF EXECUTION TIMES FOR RHS PROCESSING. BLUE PLOT IS FOR MUMPS AND RED IS FOR GPU SOLVER. X AXIS IS THE NUMBER OF ELEMENTS AND Y AXIS IS TIME IN **MICROSECONDS**. BOTH AXES ARE IN LOGARITHMIC SCALE.

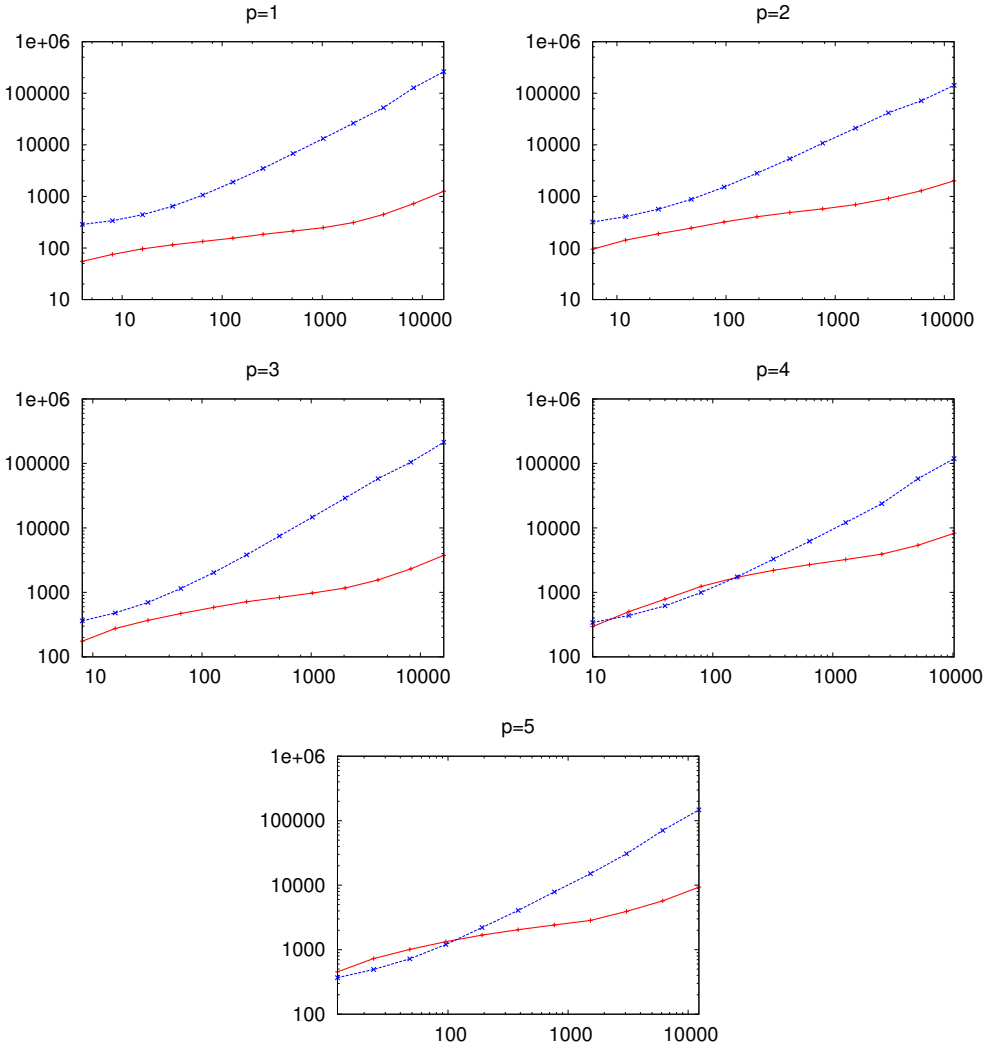

FIGURE 5.10: COMPARISON OF EXECUTION TIMES FOR COMBINED MATRIX FACTORIZATION AND MULTIPLE (32) RHS PROCESSING. BLUE PLOT IS FOR MUMPS AND RED IS FOR GPU SOLVER. X AXIS IS THE NUMBER OF ELEMENTS AND Y AXIS IS TIME IN **MICROSECONDS**. BOTH AXES ARE IN LOGARITHMIC SCALE.

6. Summary

The results presented in the previous chapter confirm the main thesis:

It is possible to implement a parallel multifrontal solver for the isogeometric final element method, which time of execution grows as $O(log\ N)$ ***for the number of elements in the computational domain denoted by*** N**.**

The implementation was entirely based on the specific graph grammar model for the parallel multifrontal solver. The model provided foundations for the time cost analysis and the design of the algorithm being implemented. The theoretical model was positively verified for a certain set of cases, which were allowed by the current state of the hardware available on the market.

B-spline basis functions used in the solver algorithm make it useful for isogeometric simulations. They also allow to solve differential equations of higher order. The difficulty in the implementation of the solver algorithm using B-splines is rewarded with a solution of higher continuity required by many engineering problems.

The huge role in a whole implementation was played by CUDA – the computing architecture for the graphical processing units. The GPU abilities to run thousands of concurrent threads allowed to simulate theoretical model conditions for a sufficient set of cases to prove correctness of this model. Moreover, the great advantages of GPU computations were used to significantly speed up solving of one dimensional differential equations on a personal computer. The solver implemented as a part of this work is **10** to **1000** times faster than currently known sequential solvers.

Future work

There are several ways to continue the work started for this thesis

- **extend the solver for higher dimensions** – 1D solver has only scientific value. Problems solved by engineers require to use at least two dimensions.
- **create a direct GPU multifrontal solver for banded matrices** – the idea is to separate solver for linear systems of equations from the solver for differential equations. This solver could be used for arbitrary banded matrices (with a limited size of band).
- **extend the solver for multiple GPUs** – in CUDA it is possible to use more than one device to compute. However, this requires special computer architecture and is not common solution in personal computers.

7. Dictionary

CPU	—	central processing unit
CUDA	—	Compute Unified Device Architecture
DE	—	differential equation
FDM	—	finite difference analysis
FEA	—	finite element analysis
FEM	—	finite element method
FM	—	frontal matrix
GPU	—	graphics processing unit
ILFM	—	intermediate local frontal matrix
LFM	—	local frontal matrix
MFM	—	merged frontal matrix
NURBS	—	non-uniform rational basis splines
ODE	—	ordinary differential equation
PDE	—	partial differential equation

List of Figures

List of Algorithms

Bibliography

[ADKL01] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[ADL00] Patrick R. Amestoy, Iain S. Duff, and Jean-Yves L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[AGLP06] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[CH58] John W. Cahn and John E. Hilliard. Free energy of a nonuniform system. i. interfacial free energy. *The Journal of Chemical Physics*, 28(2):258–267, 1958.

[CHB09] J.A. Cottrell, T.J.R. Hughes, and Y. Bazilevs. *Isogeometric Analysis: Toward Integration of CAD and FEA*. Wiley, 2009.

[dB01] C. de Boor. *A Practical Guide to Splines*. Number v. 27 in Applied Mathematical Sciences. Springer, 2001.

[DKP+07] L. Demkowicz, J. Kurtz, D. Pardo, M. Paszyński, W. Rachowicz, and A. Zdunek. *Computing with hp-ADAPTIVE FINITE ELEMENTS: Volume II Frontiers: Three Dimensional Elliptic and Maxwell Problems with Applications*. Chapman & Hall/CRC Applied Mathematics and Nonlinear Science Series. Taylor & Francis, 2007.

[DR83] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.*, 9(3):302–325, September 1983.

[Eva98] L.C. Evans. *Partial Differential Equations*. Graduate Studies in Mathematics. American Mathematical Society, 1998.

[GCH09] H. Gomez, V. M. Calo, and T. J. R. Hughes. Isogeometric analysis of phase-field models: Application to the Cahn-Hilliard equation. In *ECCOMAS Multidisciplinary Jubilee Symposium*, volume 14 of *Computational Methods in Applied Sciences*, pages 1–16. Springer Netherlands, 2009.

[Ger98] N. Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press, 1998.

[mat12a] *Mathematica website – http://www.wolfram.com/mathematica/*, September 2012.

[mat12b] *Matlab website – http://www.mathworks.com/products/matlab/*, September 2012.

[NVI11] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[Pas09] Maciej Paszyński. *Graph Grammar-Driven Parallel Adaptive PDE Solvers*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne AGH, Kraków, 2009.

[PS10] Maciej Paszyński and Robert Schaefer. Graph grammar-driven parallel partial differential equation solver. *Concurr. Comput. : Pract. Exper.*, 22(9):1063–1097, June 2010.

[PW07] M. Potter and D.C. Wiggert. *Schaum's Outline of Fluid Mechanics*. Schaum's Outline Series. McGraw-Hill, 2007.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[SF08] G. Strang and G.J. Fix. *An Analysis of the Finite Element Method*. Wellesley-Cambridge Press, 2008.

[wol12] *Wolfram Alpha website – http://www.wolframalpha.com/*, September 2012.

[Zha05] F. Zhang. *The Schur Complement and Its Applications*. Numerical Methods and Algorithms. Springer, 2005.

[Zwi98] D. Zwillinger. *Handbook of Differential Equations*. Number v. 1. Academic Press, 1998.

List of my publications

Below is the list of related papers which I co-authored during my M.Sc. course.

- K. Kuźnik, M. Paszyński and V. M. Calo *Graph Grammar-Based Multi-Frontal Parallel Direct Solver for Two-Dimensional Isogeometric Analysis*, Volume 9 of Procedia Computer Science, pages 1454 - 1463, June 2012
- M. Paszyński, K. Kuźnik and V. M. Calo *Graph grammar based parallel direct solver for 1D and 2D isogeometric finite element method*, ECCOMAS 2012, September 2012
- M. Paszyński, K. Kuźnik and V. M. Calo *Grammar-Based Multi-Frontal Solver for Isogeometric Analysis in one and two dimensions*, submitted for Scientific Programming