UPPSALA
UNIVERSITET

MAT-VET-F-24008

Examensarbete 15 hp

Juni 2024

GPU Acceleration for FEM

Determining the Performance Advantage of GPU Hardware Acceleration for Finite Element Method Solvers

Eric Björfors, Jesper Carlsson, William Ekberg,
Albin Liljefors, Erik Näslund

Civilingenjörsprogrammet i teknisk fysik

UPPSALA
UNIVERSITET

GPU Acceleration for FEM

Eric Björfors, Jesper Carlsson, William Ekberg,
Albin Liljefors, Erik Näslund

Abstract

The finite element method is a numerical method used to solve partial differential equations. GPUs are special computing units that can perform many simple operations in parallel. If the problem is parallelisable, the GPU will often be more cost or energy efficient at solving the problem. The aim of this project, is to explore how GPUs can be used to solve FEM problems. The wave equation is discretised with hat functions and Hermite polynomials. The problem is then time integrated with Runge-kutta 4 and three solvers are used to compute the derivates of each step; a direct Gaussian solver, a direct solver with LU-decomposition and a conjugate gradient solver. The project found that some operations, especially linear algebra operations, were highly parallelisable and suitable for GPUs. This made the conjugate gradient solver, a good choice for GPU-computing. Conversely, direct solvers were harder to parallelise, as each row must be computed serially. In general, the GPU was faster for all relevant grid sizes.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Jörn Zimmerling Ämnesgranskare: Chao Xu

Examinator: Martin Sjödin

Table of Contents

1 Introduction

1.1 Background and Problem Description

The continuous advancements in computational technologies have consistently expanded the horizons of scientific research and engineering. The Finite Element Method is a widely used method in various disciplines to model complex physical phenomena and engineering problems [1]. In the past, the Finite Element Method has heavily relied on Central Processing Units (CPUs) for processing. However, the computational demands have increased with the complexity and scale of the problems, causing bottlenecks in processing speed and efficiency with CPUs.[2]

Graphics Processing Units (GPUs), originally designed for handling complex graphics and image processing have emerged as powerful tools for general-purpose scientific computing. GPUs have a solid ability to execute parallel operations. This ability can strongly complement the limitations of the CPU. [3]

Applying GPU technology therefore presents a significant opportunity in accelerating computational tasks for methods like Finite Element Method.

This project aims to contribute to the broader understanding of the potential efficiencies and limitations with GPU-based numerical analysis. Additionally the project is meant to help access the feasibility of FEM on GPUs and identify ways for further optimisation and improvement within computational science.

The approach involves a rigorous investigation of the theoretical background related to FEM and GPU coding, along with a comprehensive analysis of the techniques and methods that were used. This is followed by the implementation of three different FEM solvers for solving a specific wave equation with periodic boundary conditions. The solvers that were implemented were Gaussian elimination, LU decomposition and Conjugant gradient method. The implementation with Hat functions and Conjugant gradient method were then tested on both CPU and GPU nodes at UPPMAX, the supercomputer at Uppsala University. The rest of the implementations was used to gain insights to the challenges and feasibility of implementing the methods on GPUs.

1.2 Problem formulation

This projects ambition is to explore the efficiency and feasibility of implementing FEM on GPUs for solving such equations. The main scientific questions to be investigated are:

1. Feasibility of FEM on GPUs: Is the idea of leveraging GPUs for FEM calculations a viable approach? This involves understanding theoretical support and implementation of FEM as well as suitable solvers with GPU architectures.
2. Performance Benchmarking: What level of performance, relative to the theoretical maximum, can be achieved by implementing FEM on GPUs. (CPU)

2 Theory

In this section, all theory relevant to the project is explained. It begins by showing, in general, how partial differential equations can be solved using the Finite Element Method. It is further explained how different basis functions, hat functions and Hermite polynomials, are used to create the FEM discretisation. Following, important concepts for solving linear system of equations are explained. These include the solvers Gaussian elimination, LU decomposition and the conjugate gradient method. Additionally, it is described how ordinary differential equations can be solved using Runge-Kutta methods. Finally, relevant aspects of Computer architecture are addressed. Extra weight is put on understanding memory bandwidth limitations, GPU architecture and parallel programming using CUDA.

2.1 Finite Elements Method (FEM)

The finite element method (FEM) is a powerful method used in computational science to solve differential equations. It is commonly used in mathematics and to explore, understand, and solve real-world problems in a virtual environment. It is incredibly useful as it allows us to tackle problems that are too complex for straightforward mathematical solutions. When it comes to engineering it is also a huge cost cutter and time saver as it is much cheaper, safer and faster to simulate a problem than to do physical experiments.[4]

The method builds on the concept of divide and conquer. FEM begins with the division of the problem domain into smaller and simpler parts known as finite elements. The domain could be a physical structure or any other space in which a physical phenomenon is analysed. These elements are connected at points called nodes that together create a mesh that approximates the geometry of the domain. [5]

The physical behaviour within the domain is governed by differential equations, typically partial differential equations (PDEs), along with boundary conditions. The goal of FEM is to approximate the solution of these governing equations over the discretised domain.[5]

Within each element, interpolation functions called basis functions are used to estimate the physical quantities of interest. These functions are defined in terms of the values at the nodes of the element. There are multiple ways to choose these functions. Two examples are to use hat functions or hermitian polynomials.[6][7]

1. **Define the problem**
   Firstly, the differential equations for the problem are chosen. This could for example be the heat equation or the wave equation. Then the boundary conditions and initial conditions need to be decided. This is the strong form of the problem.
2. **Defining the vector space**
   Defining the vector space provides a convenient framework to formulate and solve these problems.
   $$H_0^1 = \left\{v : \|v\|^2 + \|v'\|^2 < \infty, \text{boundary conditions}\right\}$$
3. **The weak form**
   The weak form is the variational formulation of a problem and after defining the vector space it can be derived. This is done by using a test function $v \in H_0^1$ and multiplying it with the differential equation and then integrate with integration by p arts.This results in a solution that belongs to H_0^1 that therefore satisfy the boundary conditions.
4. **Finite element approximation**
   For the finite element approximation, the solution is found by choosing a finite dimensional subspace $V_{h,0} \subset H_0^1$ made up by picewise functions. This can be done using hat functions or Hermite polynomials.
5. **Deriving the system of equations** For this step $v \in V_{h,0}$ is expressed as a linear combination of the basis functions. Then use the weak formulation and rewrite the system of equations to a matrix form.
6. **Evaluating the stiffness matrix and the load vector**
7. **Solving the ODE** This requires a ODE solver and in tern a method for solving systems of equations. There are multiple methods for this.Three methods are by using Gaussian elimination, LU decomposition or the conjugate gradient method.

[5]

2.1.1 FEM with Hat functions

From piecewise linear functions, a finite dimensional subspace can be constructed. By partitioning the domain $[0, L]$ into n non-overlapping subintervals and one subinterval can be denoted by $[x_i, x_{i+1}]$, where $i = 1, 2, \ldots, n$ and $x_0 = 0$ and $x_{n+1} = L$. The subintervals determine the finite element domains, and the x_i's are the nodal points [6, c.1].

The length of the elements, $h_i = x_{i+1} - x_i$, are not required to be equal. A smaller h leads to a more refined mesh, for the case of equidistant partition the lengths are $h = \frac{1}{n}$

The basis functions, associated with an internal node, i.e $2 \leq i \leq n$, are constructed as

$$\phi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{h_{i-1}}, & x_{i-1} \leq x \leq x_i \\ \dfrac{x_{i+1} - x}{h_i}, & x_i \leq x \leq x_{i+1} \\ 0, & \text{otherwise.} \end{cases} . \tag{1}$$

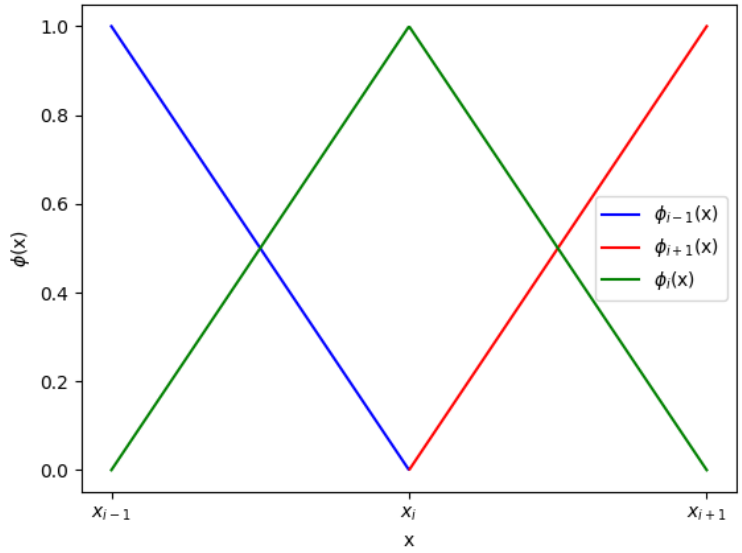


Figure 1: Piecewise linear functions

where each basis function, shown in equation 1 are commonly called "Hat Functions" and have some useful properties. [6, c.1]

1. The derivatives of equation 1 are easily derived analytically and useful for construction of the stiffness-matrices

$$\phi_i'(x) = \begin{cases} \frac{1}{h_{i-1}}, & x_{i-1} \leq x \leq x_i \\ \frac{-1}{h_i} & x_i \leq x \leq x_{i+1} \\ 0, & \text{else.} \end{cases} \tag{2}$$

2. The hat functions evaluated on a nodal point is determined by the overlapping support

$$\phi_i(x_j) = \delta_{ij} = \begin{cases} 1, & \text{if} \quad i = j \\ 0, & \text{if} \quad i \neq j \end{cases} \tag{3}$$

The weak formulation eventually ends up into constructing matrices where all elements are the inner-product of these basis functions defined by some integral. By construction, only a few of these entries will be non-zero, since most basis functions associated with each nodal point has no overlapping support. It can thus be shown that the stiffness and mass matrices take the form

$$A_{ij} = \begin{cases} \langle \phi_i^{'} | \phi_i^{'} \rangle, & i = j, \\ \langle \phi_i^{'} | \phi_j^{'} \rangle, & |i-j| = 1, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

$$M_{ij} = \begin{cases} \langle \phi_i | \phi_i \rangle, & i = j, \\ \langle \phi_i | \phi_j \rangle, & |i-j| = 1, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

with the inner product between two basis functions defined as,

$$\langle \phi_i | \phi_j \rangle = \int_\Omega \phi_i(x)\phi_j(x)\, dx. \tag{6}$$

These inner products can be calculated analytically, and the resulting local matrices are used to assemble matrices defining the discretisation on the entire domain, known as global matrices. Each local matrix has two inner-products that yield a non-zero result.

For a unifrom mesh the diagonal elements of the stiffnes matrix are given by,

$$\begin{aligned} A_{i,i} &= \int_\Omega \phi_i^{'}(x)\phi_i^{'}(x)\, dx && (7)\\ &= \int_{x_{i-1}}^{x_{i+1}} \phi_i^{'}(x)\phi_i^{'}(x)\, dx && (8)\\ &= \int_{x_{i-1}}^{x_i} \left(\frac{1}{h_{i-1}}\right)^2 dx + \int_{x_i}^{x_{i+1}} \left(\frac{-1}{h_i}\right)^2 dx && (9)\\ &= \frac{1}{h_{i-1}} + \frac{1}{h_i} && (10)\\ &= \frac{2}{h} \qquad \text{with a uniform mesh.} && (11) \end{aligned}$$

and the off-diagonal elements of the stiffness matrix are given by,

$$\begin{aligned} A_{i,i+1} = A_{i-1,i} &= \int_\Omega \phi_i^{'}(x)\phi_{i+1}^{'}(x)\, dx && (12)\\ &= \int_{x_i}^{x_{i+1}} \phi_i^{'}(x)\phi_{i+1}^{'}(x)\, dx && (13)\\ &= \int_{x_i}^{x_{i+1}} \frac{-1}{h_i}\frac{1}{h_{i+1}}\, dx && (14)\\ &= \frac{-h_i}{h_i \cdot h_{i+1}} && (15)\\ &= \frac{-1}{h} \qquad \text{with a uniform mesh} && (16) \end{aligned}$$

The diagonal elements of the mass matrix are given by,

$$M_{i,i} = M_{i-1,i} = \int_\Omega \phi_i(x)\phi_i(x)\,dx \tag{17}$$
$$= \int_{x_{i-1}}^{x_{i+1}} \phi_i(x)\phi_i(x)\,dx \tag{18}$$
$$= \int_{x_{i-1}}^{x_i} \left(\frac{x - x_{i-1}}{h_{i-1}}\right)^2 dx + \int_{x_i}^{x_{i+1}} \left(\frac{x_{i+1} - x}{h_i}\right)^2 dx \tag{19}$$
$$= \frac{h_{i-1}}{3} + \frac{h_i}{3} \tag{20}$$
$$= \frac{4h}{6} \qquad \text{with a uniform mesh} \tag{21}$$

The off-diagonal elements of the mass matrix are given by,

$$M_{i,i+1} = \int_\Omega \phi_i(x)\phi_{i+1}(x)\,dx \tag{22}$$
$$= \int_{x_i}^{x_{i+1}} \phi_i(x)\phi_{i+1}(x)\,dx \tag{23}$$
$$= \int_{x_i}^{x_{i+1}} \frac{x_{i+1} - x}{h_i}\frac{x - x_i}{h_i}\,dx \tag{24}$$
$$= \frac{h}{6} \qquad \text{with a uniform mesh [8, p.21]} \tag{25}$$
$$\tag{26}$$

In the special case of equidistant spacing, $h = \frac{1}{n+1}$, the global stiffness and mass matrices can be constructed symmetrically,

$$A = \frac{1}{h}\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} + \begin{matrix}\text{boundary} \\ \text{terms}\end{matrix}, \qquad M = \frac{h}{6}\begin{bmatrix} 4 & 1 & 0 & \cdots & 0 \\ 1 & 4 & 1 & \ddots & \vdots \\ 0 & 1 & 4 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & 4 \end{bmatrix} + \begin{matrix}\text{boundary} \\ \text{terms. [8, p.22]}\end{matrix} \tag{27}$$

2.1.2 Hermite

Hat functions are basis functions, where the interpolating requirement is only for the function value. It is possible to extend this with a requirement of matching derivatives of an arbitrary degree. The discrete solution for a time and space dependent function $u = u(x,t)$ in 1 dimension within an element $I_n = [x_{n-1}, x_n]$ is,

$$u_h(x,t) = \sum_{p=1}^{P} w(t)_{pn} H_p(x) \quad x \in I_n \quad \forall I_n \in [x_1, \ldots, x_n, \ldots x_N], \tag{28}$$

where N is the number of equidistant discretisation points and $H_p(x_i)$ are the basis functions that also interpolate the derivatives up to degree P, and w holds the time-dependent coefficients.

Substituting u_h and $v_h = H_q$ shown in section 2.1 The local mass matrix M is defined as:

$$M_{pq} = \int_{I_n} H_p(x)H_q(x)\,dx \tag{29}$$

This integral needs to be computed for all combinations of p and q ranging from 1 to P. The mass matrix entries represent the "mass" associated with each pair of basis functions over the element I_n.

The local stiffness matrix A is defined as:

$$A_{pq} = \int_{I_n} \frac{dH_p(x)}{dx}\frac{dH_q(x)}{dx}\,dx \tag{30}$$

This integral needs to be computed for all combinations of p and q ranging from 1 to P. The stiffness matrix entries represent the "stiffness" associated with the interaction between the derivatives of the basis functions over the element I_n.

To find basis functions, one can use their interpolating condition [7]. Let us use cubic polynomials, defined as

$$H_0(x) = a_0 + b_0x + c_0x^2 + d_0x^3 \tag{31}$$
$$H_1(x) = a_1 + b_1x + c_1x^2 + d_1x^3 \tag{32}$$
$$H_2(x) = a_2 + b_2x + c_2x^2 + d_2x^3 \tag{33}$$
$$H_3(x) = a_3 + b_3x + c_3x^2 + d_3x^3 \tag{34}$$
$$\tag{35}$$

and collect the weights in vector

$$\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}^T$$

Thus, the solution can be written as,

$$u(x) = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix} \cdot \begin{bmatrix} H_0(x) \\ H_1(x) \\ H_2(x) \\ H_3(x) \end{bmatrix} \tag{36}$$

and also

$$u(x) = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix} \cdot \begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \tag{37}$$

Then the interpolating conditions are used to find polynomials for the unit interval. These can later be transformed to fit any interval. The conditions yield,

$$\begin{bmatrix} u(0) \\ u'(0) \\ u(1) \\ u'(1) \end{bmatrix}^T = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix} \cdot \begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

Using equation 36 gives a solution,

$$\begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix} = I$$

such that,

$$\begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{bmatrix}$$

recall that,

$$\begin{bmatrix} H_0(x) \\ H_1(x) \\ H_2(x) \\ H_3(x) \end{bmatrix} = \begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix}$$

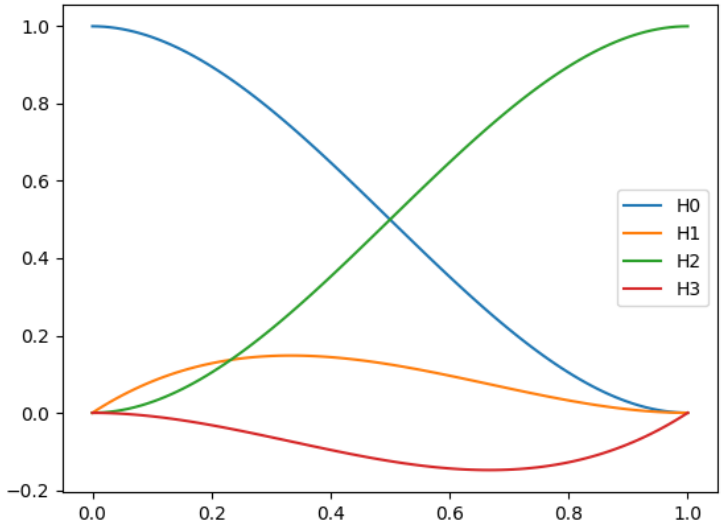


Figure 2: Cubic Hermite Polynomials

finally, polynomials are

$$H_0(x) = 1 - 3x^2 + 2x^3 \tag{38}$$
$$H_1(x) = x - 2x^2 + x^3 \tag{39}$$
$$H_2(x) = 3x^2 - 2x^3 \tag{40}$$
$$H_3(x) = -x^2 + x^3 \tag{41}$$
$$\tag{42}$$

as shown in figure 2. Notice that the only non-zero function values are

$$H_0(0) = 1 \tag{43}$$
$$H_1'(0) = 1 \tag{44}$$
$$H_2(1) = 1 \tag{45}$$
$$H_3'(1) = 1 \tag{46}$$
$$\tag{47}$$

For the example with cubic polynomials, local matrices

$$M_{local} = h \begin{bmatrix} \frac{13}{35} & \frac{11}{210} & \frac{9}{70} & -\frac{13}{420} \\ \frac{11}{210} & \frac{1}{105} & \frac{13}{420} & -\frac{1}{140} \\ \frac{9}{70} & \frac{13}{420} & \frac{13}{35} & -\frac{11}{210} \\ -\frac{13}{420} & -\frac{1}{140} & -\frac{11}{210} & \frac{1}{105} \end{bmatrix} \text{ and } A_{local} = \frac{1}{h} \begin{bmatrix} \frac{6}{5} & \frac{1}{10} & -\frac{6}{5} & \frac{1}{10} \\ \frac{1}{10} & \frac{2}{15} & -\frac{1}{10} & -\frac{1}{30} \\ -\frac{6}{5} & -\frac{1}{10} & \frac{6}{5} & -\frac{1}{10} \\ \frac{1}{10} & -\frac{1}{30} & -\frac{1}{10} & \frac{2}{15} \end{bmatrix} \tag{48}$$

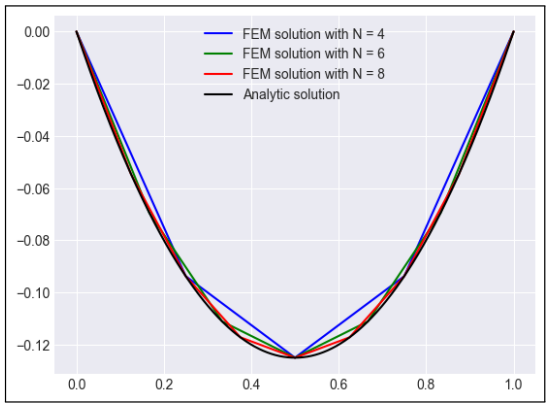
are calculated analytically, where h is distance between grid points. The global matrices are constructed by assembling all local matrices into a larger matrix that represents the entire problem domain. The elements of the global stiffness matrix A and the global mass matrix M are assembled in a manner similar to the hat functions described in section 2.1.1.

Although this was only shown for third degree interpolation, the same method can be used to generate polynomials of higher order. Naturally, it requires derivatives of higher orders that are calculated by repeatedly applying the stencil,

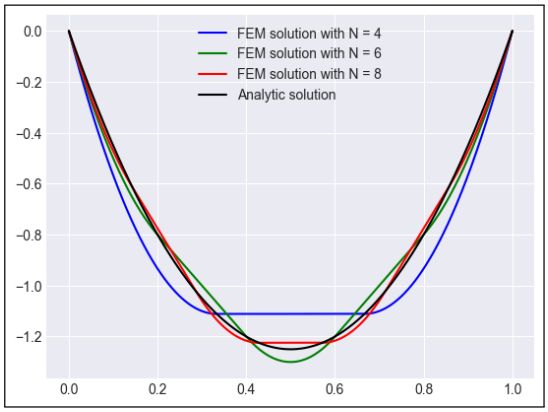
$$\frac{du(x)}{dx} \approx \frac{-u(x+2h) + 8u(x+h) - 8u(x-h) + u(x-2h)}{12h} \tag{49}$$

to solution vector u. Edge cases are handled with forward difference for the first 2 points and backward difference for the last 2 points.

2.1.3 Interpolation: Hat vs Hermite



(a) Hat basis functions.



(b) Hermite basis functions.

Figure 3: Solution to Poisson equation $u_{xx} = f$, with $f(x) = -1$

To highlight the smoother interpolation with hermite basis functions compared to hat basis functions, Poissons equation was solved with different discretisation. In figure 3 it is clear that the hermite polynomials mimic the true function with more flexibility, as expected.

2.2 Problem: Wave equation

The problem to be solved with FEM is the wave equation. It is a partial differential equation (PDE) that describes the propagation of waves through a medium. The wave equation in 1 dimension is,

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{50}$$

where u is the wave function, representing the displacement of the wave in time t and position x, v is the velocity of the wave. For simplicity, v will be set to 1.

Furthermore, will periodic boundary conditions be imposed, meaning that the values of the wave function and its derivatives are equal to those at the other boundary. For a wave equation on domain $x \in [0, L]$, periodic boundary conditions imply

$$\begin{aligned} u(0,t) &= u(L,t) \\ \frac{\partial u}{\partial x}(0,t) &= \frac{\partial u}{\partial x}(L,t) \end{aligned} \tag{51}$$

which is appropriate for simulations, since it is possible to simulate an infinite system using a finite domain.

2.3 Solvers

Solving linear systems of equations efficiently and accurately are important parts of computational science. Computational performance can differ a lot between different methods, highlighting the importance of choosing the optimal method for a problem. Two types of solvers are direct methods and iterative methods. Direct methods, such as Gaussian elimination and LU decomposition, aim to find the exact solution of the systems of equations in a finite number of operations [9, p.97]. Iterative methods, such as the Conjugate Gradient, aim instead to find a good approximation of the solution by updating a start guess until a certain tolerance is achieved. Another type of solver is used for solving ODEs, for example Runge-Kutta Methods.

2.3.1 Forward and Backward substitution

Solving a system $Ax = b$ consists of finding the vector x. If A is a lower triangular matrix, then the system

$$\begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

can be solved using the forward substitution algorithm[9, p.74]. It starts by computing

$$x_1 = \frac{b_1}{a_{1,1}} \tag{52}$$

and then use that value of x_1 compute

$$x_2 = \frac{b_2 - a_{2,1}x_1}{a_{2,2}} \tag{53}$$

This algorithm continues up to x_n and the full solution vector x is computed. If instead the matrix A is upper triangular, a similar algorithm called backward substitution can be used. It starts instead by computing the value of x_n and continues down to x_1 [9, p.74].

2.3.2 Gaussian elimination

Gaussian elimination is an algorithm for solving linear systems of equations. It is based on performing row-wise operations on the corresponding matrix of coefficients. The allowed operations are

- Swapping positions of two rows
- Multiplying by a scalar that is non-zero
- Adding a scalar multiple of one row to another row

The goal is to get the matrix on upper triangular form, since the solution then can be easily computed. Given a system $Ax = b$ where

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}, \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Different row operations will be performed until the augmented matrix is of the form

$$(A|b) = \left[\begin{array}{cccc|c} 1 & c_{12} & \dots & c_{1m} & c_{1m+1} \\ 0 & 1 & \dots & c_{2m} & c_{2m+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & cnm+1 \end{array}\right]$$

where c are the final coefficients of the row operations. The system can now be solved using backwards substitution.

2.3.3 LU Decomposition

LU decomposition is another method for solving linear systems of equations, it starts by factorising a matrix A into the product of two new matrices, one Lower triangular matrix L and one Upper triangular U as shown in equation 54.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, L = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix}, U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1m} \\ 0 & u_{22} & \dots & u_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nm} \end{bmatrix} [9, p.103] \tag{54}$$

Given the system $Ax = b$, LU decomposition results in $LUx = B$. Setting $Ux = y$ gives two new systems, $Ly = b$ and $Ux = y$, which can be solved by forward and backwards substitution respectively [9, p.109].

The matrices in the LU decomposition will be the same for every matrix A, regardless of the right-hand side b. Therefore, once the LU decomposition have been found for a problem, it can be reused for new right-hand sides. This decreases the complexity of the algorithm after the first calculation [9, p.109].

2.3.4 Conjugate Gradient Method

The Conjugate Gradient Method is an algorithm used to solve a linear system of equation on the form

$$Ax = b$$

When the matrix A is symmetric and positive definite (SPD) this method can be interpreted as an optimisation problem [10, c11.3]

The restriction of A being SPD leads to a reformulation of the problem equivalent to minimising the quadratic function

$$f(x) = \frac{1}{2}x^T Ax - x^T b \tag{55}$$

If x_c is an approximate solution to equation 55, it also approximates the solution to $Ax = b$. This reformulation allows the use of an alternative procedure for finding an ever-better approximate solution of 55, thereby approximating x in $Ax = b$. [10, c 11.3]

The conjugate gradient method is derived as an iterative process by solving the minimisation problem

$$\nabla f(x) = \mathrm{Ax} - \mathrm{b} = 0 \tag{56}$$

Starting with an initial guess, x_0 the residual $r_0 = Ax_0 - b$ is computed. The search direction for the solution vector is determined by the negative gradient of equation 56 at each point, before iteration $p_0 = r_0$.

An important aspect of the CG-method is that the search directions p_k are conjugate with respect to the matrix A. Two vectors p_i and p_j are A-conjugate if the satisfy

$$p_i^T Ap_j = 0, \qquad \text{for} \quad i \neq j \tag{57}$$

this relationship is also symmetric. [11, p.110]

This quality ensures that each search direction is orthogonal with respect to some inne-product defines by the matrix A. The search directions are used to minimise 56 and the conjugate relationship ensures that the previous search directions do not alter.

Algorithm:

$$\begin{aligned}
&k = 0, \quad r_0 = Ax_0 - b \\
&\text{While} \quad ||r_k|| > 0 \\
&\quad k = k + 1 \\
&\quad \text{if} \quad k = 1 \\
&\qquad p_k = r_0 \\
&\quad \text{else} \\
&\qquad \beta_{k-1} = (r_{k-1}^T r_{k-1})/(r_{k-2}^T r_{k-2}) \\
&\qquad p_k = r_{k-1} + \beta_{k-1} p_{k-1} \\
&\quad \text{end} \\
&\quad \alpha_k = (r_{k-1}^T r_{k-1})/(p_k^T Ap_k) \\
&\quad x_k = x_{k-1} + \alpha_k p_k \\
&\quad r_k = r_{k-1} - \alpha_k Ap_k \\
&\text{end} \\
&x = x_k
\end{aligned} \tag{34}$$

[10, p.635]

In each iteration-step α_k is calculated to determine the step-size in the direction of p_k. The solution vector, x_k is then re-calculated as well as the residual r_k. The new search-direction, p_k, is calcauted as a linear

combination of the current residual and the previous search-direction with the weight β_{k-1} to ensure that p_k is A-conjugated.

By repeating this process a convergent solution to $Ax = b$ is reached within n iteration, where n is the dimension of A. [11, p.120]

2.3.5 Runge-Kutta Methods

The Runge-Kutta methods comprise a large family of methods for solving ordinary differential equations of the form[12, p.247]:

$$y'(t) = f(t, y)$$

The most commonly used method is the classical Runge-Kutta 4 (RK4) [12, p.247]. For each time step $t_i = t_0 + i \cdot dt$ it has the form:

$$k_1 = dtf(t_i, y_i) \tag{58}$$

$$k_2 = dtf(t_i + \frac{dt}{2}, y_i + \frac{k_1}{2}) \tag{59}$$

$$k_3 = dtf(t_i + \frac{dt}{2}, y_i + \frac{k_2}{2}) \tag{60}$$

$$k_4 = dtf(t_i + dt, y_i + k_3) \tag{61}$$

$$y_{i+1} = y_i + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \tag{62}$$

[12, p.247]

The numerical stability of an ODE solver refers to the ability to solve differential equations correctly over long time periods. For RK4, numerical stability is ensured by choosing a sufficiently small time step dt [12, p.244].

2.4 Sparse Linear Algebra

The matrices, generated by FEM-discretisation, are sparse [8, p. 70]. There is a great performance advantage associated with just considering the non-zero parts of sparse matrices. This section, will focus on how to represent sparse matrices, as well as common operators for the sparse forms.

2.4.1 Ellpack Represenation

The ellpack sparse matrix representation, consists of two matrices where all non-zero elements are shifted to the very left. One array holds all the values, and the other one contains all the column indicies. To represent

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

the value matrix would be

$$\texttt{values} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 7 & 8 \\ 9 & \end{bmatrix}$$

and matrix for the column indicies are

$$\texttt{col\_idx} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \\ 3 & \end{bmatrix}.[13]$$

This project uses a simplifies variant where the number of elements per row is set to a fixed value and the matrix is flattened. This will henceforth be referred to as the flattened ellpack format. If the maximum number of elements per row was set to two, the arrays would be

$$\texttt{values} = \begin{bmatrix} 1 & 2 & 3 & 4 & 7 & 8 & 9 & 0 \end{bmatrix}$$

and

$$\texttt{col\_idx} = \begin{bmatrix} 0 & 1 & 1 & 2 & 2 & 3 & 3 & 0 \end{bmatrix}.$$

Note that some zeros may be stored with this variant.

2.4.2 Matrix-Vector Multiplication

Given a sparse matrix in the flattened ellpack format with `EPR` elements per row, each element in the resulting vector `product` for the matrix-vector multiplication of the matrix with a vector `vector`, is calculated as

$$\texttt{product[i]} = \sum_{j=0}^{\texttt{EPR}} \texttt{values}[j] \cdot \texttt{col\_idx}[j].$$

2.5 Computer Architechture

Computer Architecture describes the structure of computing systems and how its components, both hardware and software, interacts in order to fulfil the machine's purpose. The main modules of a computer are the memory, the processor and the exchange units. The memory stores all the data and instructions used by the processor, and the exchange units are responsible for the transfer of data between memory and processor [14, p.17-30].

2.5.1 Memory Hierarchies and Bandwidth Limitations

Ideally, a computer would have unlimited memory with high access speed. However, it is not economically feasible. Therefore, computer memory is structured in a hierarchy based on cost and performance, where memory "closer" to the processor has faster access speed but higher cost. When the processor wants to access data it tries to find it from the "closest" memory and if it can not find the data there, it tries the next one. The memory hierarchy consists of slow high level memory, such as RAM, and fast low level memory, such as cache. Cache memory which is located closest to the processor stores frequently accessed data and instructions, allowing the processor to quickly access them without waiting for data from slower memory devices [14, p.144-146].

The access speed of memory is also known as the bandwidth and is controlled by electrical properties and transportation methods. While these will not be discussed in further detail, it is important to understand the bandwidth limitations when addressing the runtime of programs. Programs that rely heavily on memory access may experience performance bottlenecks if the memory bandwidth is insufficient to meet their demands. Therefore, optimising memory access patterns and minimising the amount of memory needed from the higher memory levels can help improve program performance. [14]

2.5.2 Central Processing Unit (CPU)

The Central Processing Unit executes the instructions of computer programs. The principal components of a CPU is the, the arithmetic-logic unit (ALU), the control unit (CU) and the processor register [14, p.20]. Modern CPUs also feature multiple cores and different levels of cache memory hierarchies. The CU coordinates the operations in the CPU. It reads the instruction, fetches the correct data from memory and places it in registers. Registers are located inside the CPU and are where instructions and data is placed while being processed [14, p.21].

The ALU is, as the name suggests, used for performing arithmetic and logic operations. All operations are performed by boolean gates circuits. For each arithmetic operation, a separate gate circuit is needed. Therefore, simple computing machines use fewer circuits while more advanced calculations requires more circuits. The operation inputs and outputs are stored in registers, which updates when the operation is completed. A different type of register stores the next instruction to be executed [14, p.26-29].

2.5.3 Graphics Processing Unit (GPU)

The Graphics Processing Unit has its origin in the application of rendering 2D and 3D graphics. Unlike the CPU, a GPU is designed to accelerate specific tasks and workloads on a PC or smartphone. Its design for parallel processing has made the GPU widely applicable concerning video rendering, graphics and artificial intelligence. Furthermore has the structure of the GPU allowed for an dramatic acceleration of workloads in high performance computing and deep learning. [15, c.4]

Whilst the CPU is responsible for reading instructions and executing commands, the GPU is a way to off-load highly parallelisable workload from the CPU. Optimal synergy is achieved by having the CPU handle memory

allocation and sequential execution while the GPU is used to handle computationally intensive calculations on its thousands of cores.

In modern GPUs architecture components can vary between models but most use single instruction multiple data stream architecture, SIMD. A SIMD stream architecture is characterised by using one single control processor and instruction memory, allowing for one instruction to be run parallel on all cores. This can be done because each processor in the GPUs core has its one specific memory dedicated for parallel computation and data management. The big advantage for SIMD in GPUs is to rapidly execute and run calculations via data-parallelism, where multiple processors run the same task, with only one instructions-unit. [16, c.4]

Modern CUDA compatible GPUs (NVIDIA) are structured in arrays of highly threaded streaming multiprocessors (SMs), where each SM contains multiple unit processors knows as CUDA-cores. The SMs have different types of built-in memory and use gigabytes of global memory, also knows as DRAM. When a CUDA-kernel is launched, a network of treads is started that execute the kernel-code. Threads are divided into blocks and each block is assigned to one SM. [16, c.4] This enables all threads within a block to be executed simultaneously on the same SM, which is vital for synchronisation and efficiency. Synchronisation within a block is handled with barrier synchronisation. This ensures that all threads within a block has finished its part of the execution before starting another phase. Threads in a block are subdivided into warps, typically consisting of 32 threads. These warps are executed with the SIMD-model, which ensures that all threads in a warp execute the same instruction simultaneously. This enhances arithmetic flow and the efficiency of the GPUs. [15, c.4]

2.6 CUDA

Incorporating CUDA into computational projects significantly enhances the capability to perform parallel computations by making it possible to compute on NVIDIA GPUs. This harnessing of GPU architecture is crucial for accelerating a wide range of scientific, engineering, and commercial applications [3].

CUDA provides a robust framework that distinguishes between code executed on the host (CPU) and the device (GPU).This framework allows for the optimisation of compute-intensive parts by parallelising them on the GPU with CUDA. This is done with the help of CUDA kernels, which are functions run on the GPU. They allow the function to be executed on a lot of data simultaneously by spreading the computations among several thousand threads.[3]

Another critical area in CUDA is the management of memory. CUDA deals with different types of memory, each with its own unique purposes and differing access speeds. Efficient management of these memory types is crucial for performance. Developers must strategically manage data transfer between the host and the GPU to minimise latency and maximise throughput. Optimising CUDA applications therefore involves a deep understanding of both hardware limitations and programming techniques. For example, the choice of grid and block sizes can significantly affect the performance of a CUDA program.[3]

2.7 Parallel Programming

In order to utilise the entire computer, the program must be parallelised. This section, will discuss different parallel patterns as well as some differences on parallel computing for CPUs and GPUs.

2.7.1 Parallel Pattern – Parallel For

One of the simplest parallel patterns, is the parallel for pattern. For serial code with a for loop with independent iterations, different chunks of the for loop's interval, can be given to different parallel process.

2.7.2 Parallel Pattern – Reduction

Some problems, e.g. summing numbers in an array, are not perfectly patballisable. The problem arises when all the threads tries to read and write to the same variable at once. This is referred to as a *race condition*, since the different processes are competing for a variable [17, c. 2]. Imagine two processes adding $b = 2$ and $c = 5$ to $a = 0$.

As illustrated in Figure 4, the sum $2 + 5$ was wrongly calculated to 5. The reduction pattern, solves the issue of computing a single variable from an array of values.

The implementation, used in this project, is largely based on the reduction algorithm published by Nvidia's Distinguished Engineer *Mark Harris* [18]. To circumvent the race condition, a tree structure is used. Two

Process 1	Process 2
Reads $a = 0$	
	Reads $a = 0$
Computes $a = a + b = 0 + 2 = 5$	
	Computes $a = a + c = 0 + 5 = 5$
Writes $a = 2$	
	Reads $a = 5$

Figure 4: Example of a race condition.

numbers are summed into one number, and then the processes repeats until there is only one number left. Consequently, this operation will repeat $\lceil \log_2(n) \rceil$ times for n numbers.

Each array, is divided into blocks. The aforementioned process, will sum all the numbers in each block. Then the process is repeated until only one block remains.

2.7.3 Differences on CPU and GPU Parallelisation

On a CPU, there is virtually no performance gain when generating more threads than hardware threads. Since there is an overhead to create new threads, it is often beneficial to create each thread with a large workload. Since Nvidia GPU:s run each warp in SIMD (see section 2.6), each instruction should be simple and there should be one thread for each data point. A GPU will perform best if the neighbouring threads, read and write to neighbouring data. In th CPU case, the performance is best if different processes read and write from data located far away from each other.

The GPU can run many more processes in parallel than a CPU. For example, the Snowy Uppmax cluster has 16 CPU hardware threads, but it's GPU has 320 CUDA cores.

2.8 Analysing Performance with a Roofline Model

The roofline model is a performance metric tool used in high performance programming and scientific computing to understand the limitations and potential of modern processor's units of computation (kernels). It visualizes the balance between arithmetic intensity and the rate at which data can be read from or written to memory by the processor (bandwidth). Arithmetic intensity is the ratio of floating point operations (FLOPS) to memory operations, measured as FLOPS/byte. FLOPS is a measure of computational performance, indicating how many arithmetic operations can be performed per second.

The roofline diagram plots the arithmetic intensity on the x-axis and computational capacity on the y-axis. There are two main components of the diagram, bandwidth ceiling and compute ceiling. Bandwidth ceiling represents the maximum amount of FLOPS possible on the hardware. When arithmetic intensity is low, the computational capacity is limited by memory. The compute ceiling represents the maximum computational capacity. When arithmetic intensity is high, the system is compute-bound[19].

3 Method

3.1 Instruments

In the following section, the instruments used during the project are described in further detail. The section is divided into Software, which was used to build the program, and Hardware, which was used to run the program.

3.1.1 Software

1. **Git:** Git is a version control system which allows developers to save versions of their program during the development process. It is especially useful for cooperation on large projects, where multiple people needs to work in the same program. Git allows each developer to create a local ”branch” from the main program and work on it separately. Once a function has been developed on the local branch, it can be tested and then ”merged” into the main program.
2. **Python:** Python is a high level programming language with a large supply of built-in libraries. Writing code in python is much more intuitive than writing in lower level languages, making it good for writing and debugging code fast. However, since it is a high-level language, the code does not run as fast as with lower-level languages.
3. **C++:** C++ is a lower-level which is used for writing code for high-performance purposes. As C++ is a lower level language than for example python, debugging is much harder. This means that finding bugs takes more time and is more complicated. However, one big advantage with a lower level language is that it is much faster to run.[20]
4. **CUDA:** CUDA is a C++ extension which allows the developer to write code that executes on NVIDIA GPUs. It is used in high-performance computing to parallelise programs.

3.1.2 Hardware

1. **Personal computers:** All code development took place on project members' personal computers. The team also had access to personal computers with NVIDIA GPUs, which were used to try the CUDA programs during development.
2. **UPPMAX:** UPPMAX (Uppsala Multidisciplinary Center for Advanced Computational Science) has Uppsala University's resources of High-Perfomance Computing. The resources consist of multiple cluster's named after Tintin characters. The ones used in this project were Rackham and Snowy. Rackham is the general purpose cluster used for login, scheduling and smaller computations. Snowy is the cluster used for heavy computations and also has GPU access. UPPMAX clusters were accessed through ssh from personal computers.

3.2 Preparation

Before starting the project, multiple theory concepts were reviewed to get a deeper understanding of the subject, especially GPU architecture and CUDA programming, before starting with the code development. This process was supported by lectures and labs attained from the project supervisor. Additionally, in order to run code on UPPMAX, every team member needed to get an account approved. A research project with time-limited access to UPPMAX was then created, and all team members were added to the project.

3.3 Execution

3.3.1 Implementation Overview

All the computations, are contained within one executable. Input arguments control whether the data should be exported to the plotting tool or not, the number of time steps, the number of grid points, the size of the time step, the number of cuda blocks, the number of threads per cuda block, the discretisation (hat och hermite), the solver (conjugate gradient, LU or gaussian) and the device (CPU or GPU).

3.3.2 Linear Algebra Library for Heterogeneous Computing

A linear algebra library, supporting both CPU and GPU implementations, was written for this project. It contains classes for vectors, dense matrices and sparse matrices (ellpack). Each class has overloaded operator

functions for common operators such as vector add, dot product, matrix-vector multiplication and matrix-matrix multiplication. It also contains relevant constructors, an overloaded assignment operator and destructor, to handle dynamic memory allocation on the selected device and work with C++' garbage collector.

In general, the library is written in such a way that the same code can be used, no matter which device is selected. One example of this, is the implementation of the conjugate gradient solver, see section 3.3.8. Almost all operations are parallelised with a parallel for pattern. The only exception is the dot product, which firstly uses a parallel for pattern to calculate an elementwise product, and then secondly computes the sum with a reduction.

3.3.3 FEM Hat

In order to construct the global mass and stiffness matrices on the CPU and GPU with Ellpack representation, two slightly different techniques had to be deployed.

For the general assembly, constructing the global matrices is a rather straight forward process, involving copying the elements in their local counterparts over onto the global matrices. Since the local matrices for piece-wise smooth linear functions are independent of the size of the discretisation domain, these are hard coded into the program and can be used for an arbitrary sizing.

Assembling the matrices for a FEM problem using periodic boundry conditions involve invoking a circular structure that do not separate the boundry entries from the overall internal structure. In order to employ this using Ellpack the indexing for non-zero nodes was implemented using a modulo operation to check if the current nodes are out of bounds or not, if the former case these were connected around the matrix.

The structure for FEM Hat when imposing periodic boundary condition, on a 1-dimensional domain, can be futhur simplied by realising that every row in the global matrices have three non-zero entries. This realisation means that only the variable numElements is vital when going from sparse to dense matrix-representation, according to Ellpack representation in section 2.4.1.

The program responsible for constructing the global matrices only has to update two arrays, column indices and values of each entry. The approach for doing so on the CPU and GPU respectively is similar and only different to how the empty arrays are iterated and assigned values. For each row in the assembly-process, three local variables are stored to impose the boundary condition and assign non-zero values at the correct column indices

$$\begin{aligned} \text{Row index} &= 3 \cdot \text{current index} \\ \text{Next index} &= \text{index} + 1 \text{ modulo row size} \\ \text{Previous index} &= \text{index} - 1 \text{ modulo row size} \end{aligned}$$

With theses temporary variables the following psudo code were used to assemble the column indicies and values arrays corresponding to each row in the global matrices.

$$\begin{aligned} \text{Column indices[row index]} &= \text{Previous index} \\ \text{Column indices[row index + 1]} &= \text{Row index} \\ \text{Column indices[row index + 2]} &= \text{Next index} \end{aligned}$$

$$\begin{aligned} \text{Mass values[row index]} &= -\frac{1}{\mathrm{h}} \\ \text{Mass values[row index + 1]} &= \frac{2}{\mathrm{h}} \\ \text{Mass values[row index + 2]} &= -\frac{1}{\mathrm{h}} \end{aligned}$$

$$\text{Stiffness values[row index]} = \frac{\text{h}}{6}$$
$$\text{Stiffness values[row index + 1]} = \frac{4\text{h}}{6}$$
$$\text{Stiffness values[row index + 2]} = \frac{\text{h}}{6}$$

Where row index +1 corresponds to the diagonal elements in the mass- and stiffness-matrices in equation 27, and the neighbouring indexes are the off-diagonals.

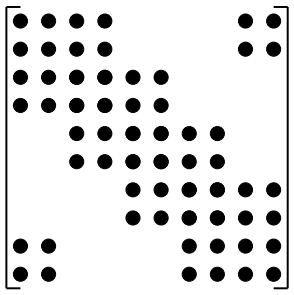
This CSR representation of the mass- and stiffness-matrices is computationally beneficial as well as easily accessible for other parts of the program that require a dense structure for execution.

3.3.4 Fem Hermite

Construction of global mass and stiffness matrices with Ellpack representation using polynomials of arbitrary degree for periodic boundary conditions was implemented. The algorithm achieving this involved finding a repeating pattern and utilization of symmetries. As mentioned in section 3.3.3, it is sufficient to store the row length since every row is the same length. It is easy to see that the length of every row is,

$$\text{rowlength} = 3\frac{P+1}{2}$$

where P is the polynomial degree of the basis functions. Since there is an overlap which starts at index $\frac{\text{P+1}}{2}$ and ends at index p, relative to starting index of row. For cubic hermite polynomials with periodic boundary conditions where $P = 3$, structure of global matrix is



put together by many 4×4 matrices. Utilizing the symmetry for local matrices, it is sufficient to calculate $\frac{\text{p+1}}{2}$ number of rows and build global matrices using indexing with offsets. This approach builds correct global matrices apart from the first $\frac{\text{p+1}}{2}$ rows and the last $\frac{\text{p+1}}{2}$ rows. This part is handled separately but in a similar manner as before, with some other offset to the indexing. Column indices and non-zero values are stored together with known row length.

3.3.5 System solution method

The second order system given from FEM discretisation is:

$$M\xi''(t) = -A\xi(t)$$

In order to solve the system, it is first reformulated into two first order systems accordingly:

$$\begin{cases} \xi'(t) = v(t) \\ Mv'(t) = -A\xi(t) \end{cases} \tag{63}$$

To find the solutions in time, RK4 was used with functions: $f_\xi(\xi_i, v_i, t_i) = v$ and $f_v(\xi_i, v_i, t_i) = -M^{-1}A\xi$. Using initial conditions ξ_0 and v_0 and time stepping $t_i = t_0 + i \cdot dt$, the solutions ξ_{i+1} and v_{i+1} were found using the following calculations.

$$k_{1\xi} = dt \cdot f_\xi(\xi_i, v_i, t_i) \tag{64}$$
$$k_{1v} = dt \cdot f_v(\xi_i, v_i, t_i) \tag{65}$$
$$k_{2\xi} = dt \cdot f_\xi\left(\xi_i + \frac{k_{1\xi}}{2}, v_i, t_i + \frac{dt}{2}\right) \tag{66}$$
$$k_{2v} = dt \cdot f_v\left(\xi_i, v_i + \frac{k_{1v}}{2}, t_i + \frac{dt}{2}\right) \tag{67}$$
$$k_{3\xi} = dt \cdot f_\xi\left(\xi_i + \frac{k_{2\xi}}{2}, v_i, t_i + \frac{dt}{2}\right) \tag{68}$$
$$k_{3v} = dt \cdot f_v\left(\xi_i, v_i + \frac{k_{2v}}{2}, t_i + \frac{dt}{2}\right) \tag{69}$$
$$k_{4\xi} = dt \cdot f_\xi\left(\xi_i + k_{3\xi}, v_i, t_i + dt\right) \tag{70}$$
$$k_{4v} = dt \cdot f_v\left(\xi_i, v_i + k_{3v}, t_i + dt\right) \tag{71}$$
$$\xi_{i+1} = \xi_i + \frac{1}{6}(k_{1\xi} + 2k_{2\xi} + 2k_{3\xi} + k_{4\xi}) \tag{72}$$
$$v_{i+1} = v_i + \frac{1}{6}(k_{1v} + 2k_{2v} + 2k_{3v} + k_{4v}) \tag{73}$$

For large systems, inverting the matrix M becomes computationally insufficient. Therefore, different solvers were used in this step to find the solution to

$$Mv'(t) = -A\xi(t) \tag{74}$$

These solvers were Gaussian Elimination, LU decomposition and Conjugate Gradient Method. Their implementation is described below.

3.3.6 Gaussian elimination

This section details the execution and implementation strategies used for the Gaussian elimination Solver and its adaptation for parallel computing with CUDA.

The solver begins by preparing the system matrix M and the right-hand side vector b into an augmented matrix format. This is fundamental for executing this Gaussian elimination method. Each row of matrix M is appended with the corresponding value from vector b, resulting in a new matrix MA. That means that the structure of MA becomes such that each element MA[i][j] corresponds to M[i][j] for j<n and MA[i][n]=b[i] for the last column.

Gaussian elimination methodology
The solver then takes the augmented matrix MA and creates an upper triangular matrix. This is done by:

1. **Row Swapping:** This operation ensures that the diagonal element of the current column is maximised to avoid numerical instability due to division by small numbers.
2. **Scalar multiplication:** Essential for scaling the pivot row so that the pivot element becomes unity.
3. **Row addition and subtraction:** This operation aims to zero out the elements below the pivot. This transforms the matrix into an upper triangular form.

Then by creating this upper triangular matrix the solution vector x is determined through back substitution.

3.3.7 LU decomposition

To solve the system 63 using LU decomposition, the matrix M needs to be decomposed into a lower triangular matrix L and an upper triangular matrix U.

Decomposition algorithm: First the matrices L and U are created as square matrices, the same size as M, where all elements are initially set to zero. The method used for calculating each row of L and the corresponding column of U is as follows.

1. Set the diagonal element $L_{i,i}$ to 1.

2. Each element in the $i:th$ row of U is calculated as

$$U_{i,j} = A_{i,j} - (L_{i,:i} \cdot U_{:i,j}). \tag{75}$$

3. Each element in the $i:th$ column of L is calculated as

$$L_{j,i} = \frac{A_{j,i} - (L_{j,:i} \cdot U_{:i,i})}{U_{i,i}}. \tag{76}$$

Forward and Backward substitution Once L and U are calculated, the new system reads $LUv'(t) = -A\xi(t)$. The solution $v'(t)$ is found by setting $Uv'(t) = y$, resulting in a system $Ly = -A\xi(t)$ which can be solved for y using forward substitution

$$y_i = \frac{-A\xi(t)_i - (L_{i,:i} \cdot y_{:i})}{L_{i,i}}. \tag{77}$$

. The solution vector $v'(t)$ is then found using backward substitution on $Uv'(t) = y$

$$v'(t)_i = \frac{y_i - (U_{i,i+1:} \cdot v'(t)_{i+1:})}{U_{i,i}}. \tag{78}$$

3.3.8 Conjugate Gradient Method

In order to solve the system 74 with a Conjugate Gradient Method on the CPU and GPU the structure of the process has to be taken into consideration. This solver is an iterative process, sequentially dependent on the previous calculation of both the residual and search direction.

Initialization: Implementing a Conjugate Gradient Method requires multiple vector, matrix and scalar values to be dynamically calculated in a sequential structure. At first these are initialized using the linalg library, where depending on the parameter "device" are stored on the CPU or GPUs memory respectively. These include the solution vector, new solution vector, residual vector, search direction vector, and the result of matrix-vector multiplications necessary for the computation in agreement with algorithm 34

Starting the Iterative Process: Before starting the iterative process of finding an approximate solution to equation 74 an appropriate guess for the solution is necessary, either passed as an input or set to zero. The residual, search direction and scalar variables are then calculated before starting the process.

Iterative Execution: The core of the execution revolves around the iterative process of refining the solution until the specified tolerance is met. Within a while-loop, the CG solver is invoked repeatedly until the error falls below the defined tolerance threshold. Since this iterative process may not converge to the exact true solution, a different metric than $||r_k||$ is required to terminate the iteration. The CG-algorithim continues until the solution is sufficiently close to the true solution, as determined by the specified tolerance. This approach ensures that the iteration continues until a satisfactory solution is obtained, based on the relative error between successive approximations.

Dynamic Memory and Computational Operations: All computation within this while-loop is either execute on the CPU or GPU depending on the parameter "device". Vector-Vector and Matrix-Vector operations are implemented for both cases in the linalg library with all necessary operators defined. For this reason there is only on function for the Conjugate Gradient Method, where the CPU and GPU are used for memory-operations aswell as computational-operations. The program is intended to work for both cases and discrepancies between the two can be found in the linalg library.

3.4 Performance Benchmarking

By comparing the performance of the implementation on the CPU and GPU it is important to note that this result is dependent on the hardware that is used. The project used an `Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz` and an `Nvidia T4`.This section will explain how the performance benchmarking was carried out.

3.4.1 Roofline Model

Profilers were used to collet data for the roofline model. The CPU code was profiled with `Intel Advisor` and the GPU code was profiled with `Nvidia Nsight Compute`.

3.4.2 Scaling

The aim of this test, to evaluate how the different devices perform with different grid sizes.

Each test was conducted on the CPU and GPU with varying grid sizes, starting from 10 000 and ending at 10 000 000, increasing by a factor of 10 in each iteration. Both the FEM Hat discretization and Conjugate Gradient Solver implementations were investigated. On the CPU, one core were utilized to parallelize the computation, and on the GPU, each kernel was launched with 256 threads and $\lceil \frac{N+255}{256} \rceil$ blocks to ensure all discretization points were covered.

4 Results & Discusion

The project's research questions will be answered in this part. First a discussion about the feasibility of different components of FEM on GPUs, then a discussion on performance benchmarking on GPUs and CPUs.

4.1 Feasibility of FEM on GPUs

The evaluation of FEM feasibility on GPU involves identifying components of the method that can be parallelised. This section will provide the results as a discussion about what parts of all the addressed FEM discretisation methods and solver algorithms could be parallelised on GPUs.

4.1.1 Hat functions

Considering the implementation of matrix assembly on the CPU versus GPU, only the traversal of the arrays was parallelisable. In the former case, a for loop was employed to iterate over the length of the matrices, whereas in the latter the structure of CUDA was utilised to parallelise the assignment of all index and values in the arrays for the CSR representation.

By imposing a periodic boundary condition in equation 27 the first and last rows of the matrices wraps around, thus creating a symmetric structure that could further simplify the implementation and execution of sparse matrix assembly. The implementation for assembly of stiffness and mass matrix on the CPU and GPU only differed on how these two functions were executed, given that the overall structure between them were similar.

While the spatial discretisation is rather small, a performance metric comparing GPU and CPU assembly will hardly differ. Only when the partioning, n, grows large will the GPU implementation be preferable, this can be understood from the for loop in the CPU implementation responsible for matrix-construction being limited by $\mathcal{O}(n)$ while on the GPU the assembly-process is mostly bound by memory operations, inherent from the linalg library, responsible for accessing and assigning values to the arrays given in section 2.4.1.

4.1.2 Hermite polynomials

Unfortunately, there was no time to implement the method. However, the feasibility of hermite polynomials for FEM can still be explored. The mass and stiffness submatrices are generated on CPU, since they are very small. The necessary data is then copied over to the GPU for the assembly.

Parallelising matrix assembly for Hermite polynomials was straight forward because mass and stiffness matrices were constructed as in section 3.3.4. The assembly was done with a nested for loop that is parallelised over the outer loop. Though it was never implemented in the system, the construction of the matrices are very likely limited by memory. The assembly meant reading from memory and writing to memory while performing only one arithmetic operation.

Expected system for implementation of hermite polynomials as basis functions is similar to increasing problem size N when using hat functions. As mentioned in section 3.3.4, if a polynomial basis of order P, the problem size will increase as P increases. The size of the sparse matrix components will be

$$\frac{N(P+1)}{2} \cdot \frac{3(P+1)}{2} = 3N \left(\frac{P+1}{2} \right)^2 \tag{79}$$

which can be compared to problem size for hat functions with quotient

$$\text{quotient} = \frac{\text{hermite size}}{\text{hat size}} = \frac{3N \left(\frac{P+1}{2} \right)^2}{3N \left(\frac{1+1}{2} \right)^2} = \left(\frac{P+1}{2} \right)^2 \tag{80}$$

where $P = 1$ for hat functions. Therefore, it is expected that the problem size increases with the quotient in equation 80 when using hermite polynomials of degree P. In theory, performance results for problem size N with hermite polynomials should be very similar to the results for problem size $N \cdot \texttt{quotient}$ with hat functions. Accuracy of solution is not taken into account, since hermite polynomials were not implemented in the final program to be run on the computer cluster.

4.1.3 Gaussian elimination

Introduction to GPU Acceleration for Gaussian Elimination The Gaussian elimination process, while inherently sequential, has components that lend themselves to parallelisation. This study explored the adaptation of

Gaussian elimination to a GPU environment using CUDA, aiming to identify and overcome the challenges posed by its sequential nature to leverage the parallel computing capabilities of GPUs.

The process of adapting Gaussian elimination to GPUs was approached by identifying the components of the algorithm that could potentially benefit from parallel execution:

1. Creating the Augmented Matrix (MA)

The initial step in the Gaussian elimination method chosen for this study involves constructing an augmented matrix from the system's coefficient matrix and the right-hand side vector. For the CPU implementation, this process is straightforward but inefficient due to its sequential nature. By contrast, this task could be fully parallelised on the GPU. Each operation of copying elements to the augmented matrix could be executed concurrently across multiple GPU threads.

2. Preparing the upper triangular matrix

The process of preparing the upper triangular matrix could not be parallelised to a high degree. The for loop driving the operation was still necessary to keep due to its sequential nature. But, some of the operations in the for loop could be parallelised.

The data movement for swapping two rows could be parallelised, similarly as for the MA matrix. On the GPU, multiple threads could simultaneously handle all the parts of the rows for swapping them.

The core of Gaussian elimination is the reduction of rows to achieve an upper triangular matrix. This step on the CPU involves iterating through rows and making calculations that depend on the results of previous iterations. This makes the process sequential but certain aspects of row reduction could be parallelised. Once the pivot is established, the modification of elements below the pivot in each column can be conducted in parallel. Multiple threads on the GPU can then simultaneously perform these subtractions and multiplications.

Something that became apparent when exploring the parallelising nature of the process of preparing the upper triangular matrix was that all computations related to the augmented matrix MA had to be done on the GPU even if they could not be parallelised. This ensured that as little data as possible had to be copied back and forth between the CPU and GPU for each step of the loop. If this had to be done on the MA matrix, it would have created a significant bottleneck, especially for running FEM with a large grid.

3. Back Substitution The final phase of this Gaussian elimination method involves solving the upper triangular matrix for the unknown vector. The process of back substitution involves solving each entry starting from the last row and moving upwards. This process typically requires sequential execution. However, the computations within each step, such as vector-vector multiplications, have been parallelised using the Linalg library. This allows for concurrent calculations, reducing the total computation time.

In conclusion, while the adaptation of Gaussian elimination to the GPU could be done, the inherently sequential steps of the algorithm, such as determining the pivot and ensuring the correct sequencing of row reductions, limit the extent of achievable parallelisation.

4.1.4 LU decomposition

The GPU parallelisation strategy to solve liner systems of equations using LU decomposition is restricted by the sequential nature of the LU decomposition algorithm and the dependencies of forward and backward substitution. The sequential nature of the LU decomposition algorithm arises from the equations 76 and 75. The element with index i, j requires both the previously calculated indexes i and j, making it poorly suited for parallelisation over those indexes. These must be implemented on a GPU similarly as on a CPU, using a nested loop over the indexes i and j. The calculation of the element $U_{i,j}$ and $L_{i,j}$ also needs the dot products $(L_{i,:i} \cdot U_{:i,j})$ and $(L_{j,:i} \cdot U_{:i,i})$ of previously calculated elements. However, the independent calculations in a dot product are not sequential, making it possible to do these calculations in parallel on a GPU.

The dependencies of forward and backward substitution arises from the equations 77 and 78. The calculation of element i in the solution vector is dependent on the previously calculated elements. In forward substitution, the previously calculated elements are the indexes from 0 up to i, and in backward substitution, the previously calculated elements are the indexes from the final vector index n down to i. Due to these dependencies, it is not possible to parallelise over index i, resulting in a GPU implementation similar to a CPU implementation,

using a loop. As in the LU decomposition algorithm, calculating each element i in the solution vector requires a dot product, $(L_{i,:i} \cdot y_{:i})$ in forward substitution and $(U_{i,i+1:} \cdot v'(t)_{i+1:})$ in backward substitution. Again, since the individual calculations in a dot product are not sequential, it is possible to parallelise these calculations on a GPU.

Due to the sequential nature of the LU decomposition algorithm and the dependencies of forward and backward substitution, the method implementations on CPU and GPU have many similarities. The main difference is that all dot products, which are the most time-consuming parts in the method, can be parallelised on the GPU. However, handling the sequential dependencies effectively remains a challenge and does not depend on CPU/GPU implementation. Therefore, even though GPU parallelisation decreases the computational complexity of the LU decomposition method, it is still a computational heavy method for solving a linear system of equations.

4.1.5 Conjugate Gradient method

Implementing the CG-algorithm 34 in section 3.3.8 requires that multiple vector, matrix and scalar values are dynamically calculated sequentially. This hiercal structure could not be parallelised since the residuals, search-direction and the updated solution vector are all dependent.

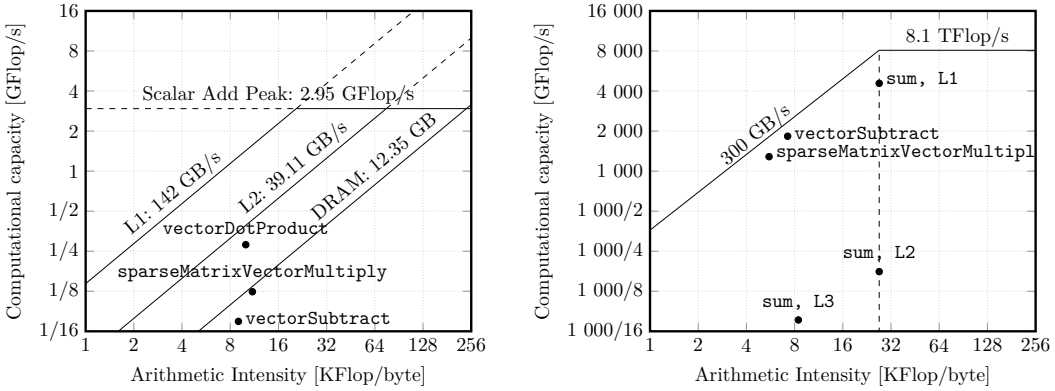
Operations and memory allocations regarding vectors and scalars in 34 were done on the CPU and GPU respectively, where most of the performance can be derived from individual kernel functions executing matrix-vector multiplication, vector-addition/subtraction, dot-product and vector-scalar operation. For instance, the matrix-vector multiplication, one of the most computationally intensive operations, can be distributed over multiple gpu-cores. This enables parallel calculation of parts of the product, which in turn reduces the operation-time compared to a traditional implementation on the CPU.

Further performance was achieved by good memory management and data-transfer between the CPU and GPU. To minimise latency and enhance performance, the implementation of the CG-method was done to execute only on either the CPU or GPU and only writing back to the CPU when necessary.

Even though the CG-methods sequential structure could not be fully parallelised, considerable performance could be reached by parallelising individual computations within each iteration. By strategically utilising the GPU ability for parallel processing and optimising memory-management can the CG-algorithm reach considerably performance improvement compared to the CPU implementation.

4.2 Performance Benchmarking

4.2.1 Roofline Model



(a) CPU `Intel(R) Xeon(R) CPU E5-2660`.

(b) GPU `Nvidia T4`. Run with 2 000 blocks and 512 threads per block.

Figure 5: Roofline model for different funcions and kernals using hat elements, conjugate gradient solver and 10 000 000 grid points.

Performance data from the Intel Advisor and Nvidia Nsight Compute profiler, is shown in Figure 5a and 5b respectively. The roofline shows the computional capacity or speed on y-axis and the arimetic intensity on the x-axis. The arythmetic intensity is the number of operations done, for each byte of memory loaded. If

the arythmetic intensity is low, the memory bandwith will be the bottleneck. Conversly, if it is high, the computational capacity will be the bottleneck. This is illustrated with the dashed line in Figure 5b. The CPU roofilne model in Figure 5a, shows different bandwithds for different memory modules.

Since all operations are more or less memory bound, the high bandwidth of GPU memory makes a big differance. The reduction implemented on the GPU can decently be improved. The algorithm used, was not even the fastest method supplied in Mark Harris' presentation [18]. The selected method was mainly used because it was easier to understand and debug.

The CPU performance would benefit considerably from parallelisation. Parallelising code with parallel for and reductions, can easily be done with openMP compiler directives. It only requires one extra line per loop to parallelize. Thus, despite the GPU code running faster, it might take longer to implement. Notably, it is only effective to parallelize up to the number of hardware threads available, on this CPU 20, and not to the level of GPUs.

4.2.2 Scaling

The expectations were that the GPU provides significant improvements to performance, compared toe the CPU, in cases where the computational work-load is highly parallelisable. This is clear when computing large datasets that are associated with the finite element method. The GPU architecture design for multiple operations makes it ideal for iterative and matrix-heavy computations related to FEM hat discretization and the conjugate gradient method.

With smaller grid size was the discrepancy in performance between the CPU and GPU implementations less pronounced. However, when increasing the grid size past 10 000 points, the advantages of using the GPU is much more clear. Especially during tests with the CG-method, which utilize parallel computing heavily, did the GPU outperform the CPU in regard to execution time. This solver method, that is highly dependent on matrix-operations and vector calculations, suits well to the parallel processing ability in modern GPUs.

Table 1: Discretization Performance Comparison using FEM Hat Functions

	Execution Time (ms)	
N	CPU	GPU
10 000	1,77	6,00
100 000	16,92	6,50
1 000 000	117,16	7,26
10 000 000	895,62	13,10

Table 2: Performance Comparison using the Conjugate Gradient Solver

	Execution Time (ms)	
N	CPU	GPU
10 000	434	30
100 000	1 060	74
1 000 000	7 126	335
10 000 000	77 146	1 354

4.3 Further Research

For further research, it would have been interesting to develop performance benchmarking for the implementations of Hermite basis function, Gaussian Elimination and LU decomposition. From the discussions about their feasibility on GPUs, all implementations should achieve better performance on GPUs for large matrices. However, it is not possible to form any conclusion on the exact performance improvement without further investigation.

It would also be interesting to analyse the difference in energy consumption for the difference devices, as well as analysing how the problem scales into higher dimensions.

5 Conclusions

This report has examined the possibilities, and limitations, of deploying GPU-acceleration for solving finite element methods problems arising from the wave equation. The results imply that GPU-acceleration techniques can offer considerable performance compared to traditional CPU-methods. These possibilities are however limited to the structures of subparts of the problem, and most adapted for parallelisable parts.

Throughout the experiment, it has been possible to conclude that the usage of GPUs don't just speed up the computational process, but are also well adapted for data-management of large data-sets. This is a valuable finding within technology and computational science, where efficient solutions to solving the wave equation is of considerable importance. In general, it was found that all operations are more or less memory bound and the GPU will be faster than the CPU for all relevant grid sizes.

To conclude this report, enhancement of understanding the GPU-acceleration potential has contributed to a deeper insight into the GPU's limitations and challenges. Future work should focus on expanding the investigation into higher dimensions, as well as gaining a deeper understanding of the different bottlenecks on the difference devices.

References

[1] Johnson C. *Numerisk lösning av partiella differentialekvationer med finita elementmetoden*. Studentlitt., Lund, 1980. ISBN 9144163517.

[2] Meng H.T, et al. Gpu accelerated finite-element computation for electromagnetic analysis. 56, 2014. ISSN 1045-9243.

[3] Garland M, et al. Parallel computing experiences with cuda. 28, 2008. ISSN 0272-1732.

[4] Kennedy, G. J. Martins, J. R. R. A. A parallel finite-element framework for large-scale gradient-based design optimization of high-performance structures. 2014. ISSN 0168-874X.

[5] Mattsson M, Almquist K. A guide to scientific computing for partial differential equations. Uppsala, 2023. Department of Information Technology.

[6] Hughes Thomas JR. Fundamental concepts; a simple one-dimensional boundary-value problem. In *Finite Element Method - Linear Static and Dynamic Finite Element Analysis*, pages 1–1. Dover Publications, 2000. ISBN 9780486411811.

[7] Hunter J, Fernandez P. An anatomically based patient-specific finite element model of patella articulation: Towards a diagnostic tool. *Biomechanics and modeling in mechanobiology*, 4:20–38, 09 2005. doi: 10.1007/s10237-005-0072-0.

[8] Saad Y. *Iterative methods for sparse linear systems*. PWS Publ., Company, Boston, Mass, 1996. ISBN 053494776X.

[9] Allaire G, Kaber SM. *Numerical Linear Algebra*. Springer New York., 1st ed edition, 2008.

[10] Golub GH, Van L, Charles F. *Matrix computations*. Johns Hopkins studies in mathematical sciences. Johns Hopkins Univ. Press, Baltimore, 4. ed. edition, 2013. ISBN 9781421407944.

[11] Hestenes MR. *Conjugate direction methods in optimization*. Applications of mathematics, 12. Springer-Vlg, New York ;, 1980. ISBN 0387904557.

[12] Lindfield G, Penny J. *Numerical Methods*. Academic Press, 4th ed edition, 2019.

[13] Konstantin I, Babeshko I, Krutikov A. 2021.

[14] Dupouy G, Blanchet B. *Computer Architecture*. ISTE, 2013.

[15] Hennessy J. L, Patterson D. A. *Computer architecture : a quantitative approach*. Morgan Kaufmann series in computer architecture and design. Morgan Kaufmann, Waltham, MA, fifth edition. edition, 2012 - 2012. ISBN 1-283-29897-X.

[16] Hwu WW, Kirk D, Hajj IE. *Programming massively parallel processors : A hands-on approach*. Morgan Kaufmann, Cambridge, MA, fourth edition. edition, 2022 - 2023. ISBN 0-323-91231-1.

[17] Pacheco P. S. *An introduction to parallel programming*. Elsevier Science Technology, 2013.

[18] Harris M. Optimizing parallel reduction in cuda. `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`. Accessed: 2024-05-22.

[19] Ofenbeck G, Steinmann R, Caparros V, and Spampinato D. G, Püschel M. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, 2014. doi: 10.1109/ISPASS.2014.6844463.

[20] Oden L. Lessons learned from comparing c-cuda and python-numba for gpu-computing. 28:216–223, 2020. doi: 10.1109/PDP50117.2020.00041.

[21] Meuer HW, Strohmaier E, Dongarra J, Simon H, Meuer M. Top500 the list, 2024. URL `https://top500.org/`. 1993-2024 TOP500.org (c).

[22] IBM Cloud Education. Why green coding is a powerful catalyst for sustainability initiatives. `https://www.ibm.com/blog/green-coding/`, January 2023. Accessed: 2024-5-21.

[23] Goal 9, "build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation". `https://sdgs.un.org/goals/goal9`. Accessed: 2024-05-21.

[24] Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Paulo Fernandes J, Saraiva J. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021. ISSN 0167-6423. doi:

https://doi.org/10.1016/j.scico.2021.102609. URL https://www.sciencedirect.com/science/article/pii/S0167642321000022.

Popular Scientific Summary

In the project, a mathematical problem was solved using something called the Finite Element Method (FEM). It is like solving a puzzle by breaking it into smaller, more manageable pieces. To increase performance of the method, the project assesses if using different parts of the computer would affect solution speed. The computer parts investigated were CPUs and GPUs. The CPU is the brain of the computer. It does the overall thinking and decision-making. A GPU, or Graphics Processing Unit, is a component in the computer that's specially designed to handle graphics and visual tasks quickly.

This makes GPUs incredibly effective for tasks that can be broken down into many smaller, independent tasks, such as rendering images, processing large datasets, or performing repetitive mathematical operations. Tests were conducted using both CPUs and GPUs, and it was found that GPUs were the more appropriate computer architecture for solving the problem. It surpassed CPUs in regard to both computational performance and sustainability performance.

The project and Sustanability

Power efficiency for super computers: GPU vs CPU

The top 500 list [21] is tracking the performance of the best supercomputers in the world. It shows system specifications, number of cores, flops and power usage. Flops stands for floating operations per second. Operations include addition, subtraction, multiplication, division, and other complex mathematical functions performed on floating-point numbers. System specifications on the list reveal that both CPU and GPU architecture is in the best computers in the world. The green 500 list which tracks the same information as top 500 but with the addition of energy efficiency measured in Giga flops per watts. GPUs completely dominate the system architecture used in the most energy efficient super computers. the project experiments show that a GPU is the superior architecture for FEM, but that is clearly not the only advantage of using GPUs.

Green coding

In the project experiments, C++ was used to run large-scale tests. The choice of C++ was driven by its reputation for efficiency in terms of speed, energy consumption, and memory usage. These tests required extensive computational power, and using C++ allowed us to optimize the performance and energy efficiency of the simulations. This approach aligns with the principles of green coding, which seeks to minimize the energy required to process code.

Green coding involves writing software in a manner that minimizes energy consumption, thereby reducing the carbon footprint associated with IT operations. Many organizations have set greenhouse emission reduction goals in response to the climate change crisis and global regulations[22], and green coding especially supports sustainable development goal 9, ”Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation”[23].

Recent research into the speed and energy use of different programming languages found that C++ was one of the most efficient in speed[24], reducing energy and memory usage and providing another potential opportunity for energy savings. Using C in the FEM experiments demonstrates the tangible benefits of green coding. This approach leads to reduced energy consumption and accelerated progress toward sustainability goals. By leveraging the efficiency of C++, it can be ensured that large-scale tests are both effective and environmentally friendly, supporting broader sustainability initiatives.

Acknowledgements

A special thanks to Jörn Zimmerling who has guided us through the entire project with great passion and enthusiasm. He has provided valuable insights into GPU-computing, as well as helping us put the project in a broader picture. We would also like to thank Gunilla Kreiss who has helped us gain a better understanding of Finite Element Methods, especially with the Hermite basis functions.