Acceleration of the Multi Level Fast Multipole Algorithm on a Graphics Processing Unit

C. Wagenaar

Acceleration of the Multi Level Fast Multipole Algorithm on a Graphics Processing Unit

by

C. Wagenaar

in partial fulfillment of the requirement for the degree of

Master of Science
in Applied Mathematics

at the Delft University of Technology,

to be defended publicly on Friday August 19, 2016 at 10:00 AM.

Student number:	4090772	
Project duration:	October 15, 2016 – August 19, 2016	
Supervisor:	Dr. ir. D. R. van der Heul,	TU Delft
	Dr. ir. H. van der Ven.	NLR
Thesis committee:	Prof. dr. ir. C. Vuik,	TU Delft
	Dr. ir. D. R. van der Heul,	TU Delft
	Dr. ir. R. F. Remis,	TU Delft
	Dr. ir. H. van der Ven.	NLR

Abstract

It is of great importance that enemy aircraft can be detected by a radar. Good knowledge about one's own detectability is also needed to make one's own visibility as low as possible. Obtaining the aircraft radar signature involves solving a scattering problem with a large, low-observable aircraft. Briefly explained; the incident radar wave induces a current distribution on the surface of the scatterer. Using the Method of Moments (MoM), this current distribution can be found as the solution of an integral equation. The current distribution is expanded in a set of basis functions. Because every pair of basis functions interacts through the Green's function, the continuous equation turns into a matrix equation after discretization. The matrix in this equation is a dense matrix. It is not desirable for practical problems to save the large system matrix, because this limits the problem size considerably. The Fast Multipole Method (FMM) became popular because it makes it possible to reduce the memory storage and the work needed to solve the discretized integral equation greatly. The work scales proportional to $O(N^{\frac{3}{2}})$, where N is the number of unknowns. The Multi Level Fast Multipole Algorithm (MLFMA) reduces the required memory and computational complexity even more to $O(N \log N)$ by having different levels of clustering.

Radar signature computations are, with the use of the Multi Level Fast Multipole Algorithm, still computationally intensive. Graphics Processing Units (GPU's) have the potential of high computational power at relatively low cost. Given the hardware and software restrictions of Graphics Processing Units, some parts of the Multi Level Fast Multipole Algorithm are better suited for calculations on Graphics Processing Units than others. Only by perfectly understanding the properties of Graphics Processing Units, the applicability of the Multi Level Fast Multipole Algorithm can be fully exploited.

Matrix vector multiplications are the most time consuming parts in the Multi Level Fast Multipole Algorithm and are suitable for parallel computations. Therefore, this part of the algorithm is implemented on the Graphics Processing Units. A discretized Poisson equation will serve as a model problem for the Multi Level Fast Multipole Algorithm computation. The Poisson equation is discretized with finite differences on a structured grid in two dimensions. The discretized Poisson equation is iteratively solved on a Graphics Processing Unit. When the calculations are performed on a Graphics Processing Unit, they take up to 11 times less time in comparison with the same system performed on a single core of a Central Processing Unit (CPU).

Three small test problems for the Multi Level Fast Multipole Algorithm are used to compare the performance of different implementation methods. The test problems have a realistic structure, but have a smaller number of unknowns compared to real problems. Like in many other applications, the bottleneck of Graphics Processing Units is the fact that information has to be sent back and forth. When a large amount of data has to be sent back and forth, the Graphics Processing Unit is not faster than a Central Processing Unit. Three test problems are accelerated 15 to 21 times on a Graphics Processing Unit, which is a promising result for application to real problems.

Keywords Radar signature · Multi Level Fast Multipole Algorithm · Parallel computing · Graphics Processing Unit

Preface

This document about the 'Acceleration of the Multi Level Fast Multipole Algorithm on a Graphics Processing Unit' has been written to fulfill the graduation requirements of the Master Applied Mathematics at the University of Technology in Delft. The project has been conducted in cooperation with the Netherlands Aerospace Center (NLR) in Amsterdam.

I would like to thank the people involved in this project, with special thanks to my supervisors Harmen van der Ven, Duncan van der Heul and Kees Vuik for guiding me and for always being willing to answer my queries.

C. Wagenaar
Delft, August 2016

Contents

List of Figures

List of Tables

1
Introduction

Radars have been used to detect enemy aircraft since World War II. Ever since, knowing the radar signature of an aircraft, being friend or foe, has been used for modern warfare. The extent to which aircraft are detected by a radar is called the radar signature of an aircraft. There are two ways of obtaining this radar signature: experimentally or by using advanced computational methods. These methods should become better and better at meeting the ever increasing demand for analyzing larger and more intricate platforms.

For simplicity, in this thesis it is assumed a single radar sends a plane wave of a single frequency to the aircraft. The idea of radar signature computations is to determine the equivalent current induced by the radar wave on the airplane. This current can then be used to determine the electromagnetic waves, reflected from the aircraft in any direction. The current distribution on the airplane induces an electric field somewhere else on the airplane. The current is approximated using a finite element method, where the unknowns are the scatterers or the degrees of freedom. The current distribution can be found as the solution of an integral equation. The integral equation is derived directly from the Maxwell equations, and therefore describes all the physics: it is a full-wave method. After discretization of the integral equation, the problem can be written as a matrix equation in the form $A\boldsymbol{x} = \boldsymbol{b}$, where A is the full interaction matrix between the basis functions, $\boldsymbol{x}$ contains the coefficients of the basis functions and $\boldsymbol{b}$ contains the excitation by the incoming wave. Because of the fact that several problems have a number of unknowns in the order of 10^7, this complete matrix cannot be stored. The Multi Level Fast Multipole Algorithm (MLFMA) is a method to access the system matrix without having to store the whole matrix. This is the reason why the algorithm became popular.

There is a need to perform calculations of the MLFMA in the shortest possible time. This results in the question to speed up the MLFMA further. In this thesis Graphics Processing Units (GPU's) are used to accelerate the MLFMA. GPU's perform calculations in parallel to obtain the result faster. The suitability of the building blocks of the MLFMA are assessed one by one to see if they are suitable for the use of GPU's. After the description of the possible methods and bottlenecks of performing the MLFMA on a GPU, the matrix vector multiplication parts of the MLFMA are chosen to have the focus in this thesis.

As a first implementation on the GPU, a Poisson solver is used as a model problem to face the difficulties while utilizing the GPU. CPU and GPU performances are compared and the results are explained. With this information, different implementation methods are compared for matrix vector multiplications in three MLFMA test problems. The results of the different implementations are compared and explained. The best implementation method is used to determine the acceleration for this test problem on the GPU.

The goal of the project is to accelerate the Multi Level Fast Multipole Algorithm with the help of a GPU on a 'given' test problem. NLR probably starts using GPU's for this type of problems if the test problem is 5 times faster on a GPU, compared to a CPU with one core. It is important that the number of lines in the code is not increased by 5 as well.

The research question for this thesis is:
Which parts of the Multi Level Fast Multipole Algorithm could be calculated faster when those parts are transferred to a GPU, what is the expected speed-up and what are the bottlenecks of using GPU's for those parts?

In the next chapters the physics of electromagnetic scattering is explained in more detail and the Method of Moments is introduced. In Chapter 4 the MLFMA is explained in detail and the building blocks of the algorithm are presented. The characteristics of GPU's and possible ways of implementation are explained in Chapter 5. After that, a first analysis of the MLFMA on GPU's is described and different possibilities of parallelizing the MLFMA on GPU's from literature are discussed. In Chapter 7 implementation details and results for a model problem are explained and finally part of the MLFMA is parallelized for three test problems and the results are given. Conclusions and suggestions for future work finalize the thesis.

2

Physics Behind Electromagnetic Scattering by Homogeneous Objects

When a radar sends a transversal wave to an object, the object partly rebounds this wave as soon as it reaches the object. The idea is to determine the equivalent current distribution induced by the radar wave on the airplane. This current distribution can be used to determine the electromagnetic waves, reflected from the aircraft in any direction (in particular in the direction of the radar). Because of this radar wave, currents on one part of the airplane induce an electric field somewhere else on the airplane: those two parts of the aircraft interact with each other. Calculations are performed as if there is just a single radar and as if this radar transmits waves of just a single frequency. For real problems, this is more complex with different radars sending intricate wave forms. Typical radar frequencies are in the range of 1 to 50 GHz, resulting in wave lengths between 30 cm and 6 mm.
More information about formulas or calculations for the physical model of the MLFMA can be found in [1].

It is assumed that the aircraft is made of Perfectly Electrically Conduction (PEC) materials. An advantage of this is that there is no tangential electric field on the surface, so the magnetic current is zero. But all algorithmic computations of the MLFMA are needed to solve the scattering problem for PEC surfaces. Only the equations for the Electric Field Integral Equation (EFIE) are used.
The results for this thesis will be approximately the same for cases where the aircraft is made of non Perfectly Electrically Conduction materials.

The wave forms can be derived from the Maxwell equations and the constitutive relations:

$$\nabla \times \nabla \times \boldsymbol{E} - k^2 \boldsymbol{E} = -i\omega\mu \boldsymbol{J} - \nabla \times \boldsymbol{M}, \tag{2.1}$$

where E, J and M are fields as function of the place. The equations hold in the vacuum around the scatterer. $\boldsymbol{E}(\boldsymbol{x}, t) = \text{Re}\,(\boldsymbol{E}(\boldsymbol{x})e^{i\omega t})$ is the electric field intensity. The complex-valued $\boldsymbol{E}(\boldsymbol{x})$ is called a phasor, $\boldsymbol{J}$ is the current density and $\boldsymbol{M}$ the magnetic charge. μ is the permeability, ω the frequency and k the wave number. $k^2 = \omega^2\mu\epsilon$ and ϵ is the permittivity of the medium in F/m. The frequency domain equations are defined as $e^{-i\Omega t}$.

The wave form of the Maxwell equations and the wave equation are similar, therefore solutions to the Maxwell equations can be constructed from Green's functions [1].

If $\boldsymbol{E}^{inc}$ is the incident electrical field and $\boldsymbol{H}$ the magnetic field, for $\boldsymbol{r} \in S$ (the scattering surface) the Electric Field Integral Equation (EFIE) is given by:

$$\boldsymbol{E}^{inc}(\boldsymbol{r}) - \frac{1}{4\pi}\int_S (i\omega\mu(\boldsymbol{n} \times \boldsymbol{H})\phi - \boldsymbol{n} \times \boldsymbol{E} \times \nabla'\phi - (\boldsymbol{n} \cdot \boldsymbol{E})\nabla'\phi)\, d\boldsymbol{r}' = \frac{\Omega(\boldsymbol{r})}{4\pi}\boldsymbol{E}(\boldsymbol{r}), \tag{2.2}$$

where $\boldsymbol{n}$ is the normal.

In Equation 2.2, the term $\frac{\Omega(\boldsymbol{r})}{4\pi}$ finds its origin in a limit process. ∇' is the gradient on the second variable and ϕ is the Green's function and is given by:

$$\phi(\boldsymbol{r},\boldsymbol{r}') = \frac{e^{-ik|\boldsymbol{r}-\boldsymbol{r}'|}}{|\boldsymbol{r}-\boldsymbol{r}'|}. \tag{2.3}$$

The subtended solid angle $\Omega(\boldsymbol{r})$ is defined as:

$$\Omega(\boldsymbol{r}) = \lim_{h\to 0}\int_{|\boldsymbol{r}-\boldsymbol{r}'|=h,\boldsymbol{r}'\in V} \frac{1}{h^2} d\boldsymbol{r}', \tag{2.4}$$

where h is the distance between $\boldsymbol{r}$ and $\boldsymbol{r}'$ and V is the free space surrounding the scatterer [1].

The surface current density $\boldsymbol{J}_S$ and the fictitious magnetic surface charge $\boldsymbol{M}_S$ are given by:

$$\boldsymbol{J}_S := \boldsymbol{n}\times\boldsymbol{H}, \tag{2.5}$$

$$\boldsymbol{M}_S := -\boldsymbol{n}\times\boldsymbol{E}. \tag{2.6}$$

As explained before, the magnetic current is zero, so $\boldsymbol{M}_S = 0$.

The following equations are derived from Maxwell equations:

$$\boldsymbol{n}\cdot\boldsymbol{E} = \frac{i}{\omega\epsilon}\nabla\cdot\boldsymbol{n}\times\boldsymbol{H} = \frac{i}{\omega\epsilon}\nabla\cdot\boldsymbol{J}_S, \tag{2.7}$$

$$\boldsymbol{n}\cdot\boldsymbol{H} = \frac{-i}{\omega\epsilon}\nabla\cdot\boldsymbol{n}\times\boldsymbol{E} = \frac{i}{\omega\mu}\nabla\cdot\boldsymbol{M}_s. \tag{2.8}$$

Using those equations, (2.2) for EFIE can be rewritten to:

$$\boldsymbol{E}^{inc} - \frac{1}{4\pi}\int_S (i\omega\mu(\boldsymbol{J}_S\phi - \frac{i}{\omega\epsilon}(\nabla'\cdot\boldsymbol{J}_S)\nabla'\phi)d\boldsymbol{r}' = \frac{\Omega(\boldsymbol{r})}{4\pi}\boldsymbol{E}(\boldsymbol{r}). \tag{2.9}$$

The vector field is completely determined by the tangential part of the integral equation, because the electric current is a tangential vector field. Let $\boldsymbol{E}_t^{inc}$ be the tangential part of the incoming field. Then using $\boldsymbol{f}_t = -\boldsymbol{n}\times(\boldsymbol{n}\times\boldsymbol{f})$ for any vector field $\boldsymbol{f}$, EFIE can be written in the currents:

$$\boldsymbol{E}_t^{inc} - \frac{1}{4\pi}(\int_S i\omega\mu\boldsymbol{J}_S\phi - \frac{i}{\omega\epsilon}(\nabla'\cdot\boldsymbol{J}_S)\nabla'\phi d\boldsymbol{r}')_t = 0. \tag{2.10}$$

Remembering $k = \omega/c$ and $\epsilon\mu c^2 = 1$ give the equations $\omega\mu = k\eta$ and $\omega\epsilon = k/\eta$. Together with the impedance $\eta = \sqrt{(\frac{\mu}{\epsilon})}$, EFIE is the following:

$$\boldsymbol{E}_t^{inc} - \frac{ik\eta}{4\pi}(\int_S \boldsymbol{J}_S\phi - \frac{1}{k^2}(\nabla'\cdot\boldsymbol{J}_S)\nabla'\phi d\boldsymbol{r}')_t = 0. \tag{2.11}$$

The goal is the determination of the radar cross section σ:

$$\sigma = \lim_{r\to\infty} 4\pi r^2 \frac{|\boldsymbol{E}_S|^2}{|\boldsymbol{E}^{inc}|^2}, \tag{2.12}$$

where $\boldsymbol{E}_S$ is the scattered electric field intensity, determined by the surface current density $\boldsymbol{J}_S$. More information about how to determine $\boldsymbol{E}_S$ can be found in [1].

The incoming signal and the signal which was sent are compared and finally the radar cross section is a measure of the detectability of the object.

3

Method of Moments

The Method of Moments (MoM) [19] is a method that is used to numerically solve linear partial differential equations that are written in boundary integral form. It is a finite element discretization of an integral equation that can be written as a matrix equation.

Let $\boldsymbol{W}$ be a test vector field, which is tangential to S and δS. The weak formulation of EFIE is obtained by multiplying (2.11) by $\boldsymbol{W}$ and integrating over S:

$$\int_S 4\pi \boldsymbol{W}\cdot\boldsymbol{E}_t^{inc} d\boldsymbol{r} = \int_S\int_S ik\eta\left(\phi(\boldsymbol{r},\boldsymbol{r}')\boldsymbol{W}(\boldsymbol{r})\cdot\boldsymbol{J}_S(\boldsymbol{r}') - \frac{1}{k^2}(\nabla\cdot\boldsymbol{W}(\boldsymbol{r})\phi(\boldsymbol{r},\boldsymbol{r}')(\nabla'\cdot\boldsymbol{J}_S(\boldsymbol{r}')\right) d\boldsymbol{r}\,d\boldsymbol{r}'. \tag{3.1}$$

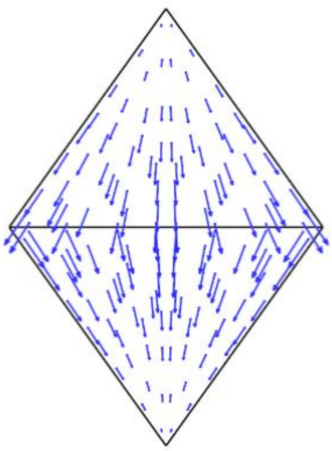

Figure 3.1: Degree of freedom or basis function living on the edge connecting two triangles.

The solution vector $\boldsymbol{J}_S$ is expanded into a linear combination of basis functions $\boldsymbol{f}_j$ (Figure 3.1), where the sum is over all internal edges. The functions have compact support.

$$\boldsymbol{J}_S(\boldsymbol{r}) = \sum_j^N J_j \boldsymbol{f}_j(\boldsymbol{r}). \tag{3.2}$$

Inserting this expansion for the current gives the following expression:

$$< \boldsymbol{f}_i, 4\pi \boldsymbol{E}_t^{inc} >= \sum_j \eta \mathscr{L}_{ij} J_j, \tag{3.3}$$

where $\mathscr{L}_{ij} = \mathscr{L}(\boldsymbol{f}_i, \boldsymbol{f}_j)$:

$$\mathscr{L}_{ij} = ik \int_S \int_S (\boldsymbol{f}_i(\boldsymbol{r}) \cdot \boldsymbol{f}_j(\boldsymbol{r}') - \frac{1}{k^2} (\nabla \cdot \boldsymbol{f}_i)(\boldsymbol{r})(\nabla \cdot \boldsymbol{f}_j)(\boldsymbol{r}'))\phi(\boldsymbol{r}, \boldsymbol{r}') d\boldsymbol{r} d\boldsymbol{r}'. \tag{3.4}$$

i and j are the degrees of freedom between which the interaction is determined.

The system matrix is fully populated and only required for matrix vector products (no other calculations have to be performed with this fully populated matrix). For realistic radar-cross section computations, this system matrix is too large to multiply by a vector on a computer. Therefore, the Fast Multipole Method (FMM) is discussed in Chapter 4. The FMM is a method of finding an approximation of the outcome of this matrix vector multiplication without the need of storing the matrix explicitly.

4

Multi Level Fast Multipole Algorithm

The Multi Level Fast Multipole Algorithm (MLFMA) is a numerical method that solves the matrix equation iteratively without calculating the whole matrix itself. The distinguishing feature is that the matrix is full because the equation is an integral equation and no Partial Differential Equation. The matrix vector multiplications are the most time consuming parts of the iterative solver, as long as the Krylov basis is not too large (the amount of powers is small compared to the number of matrix vector products in the MLFMA). A direct method or LU decomposition is time consuming because of the fact that LU scales with $O(N^3)$.
The MLFMA saves memory while increasing the computing efficiency rapidly. The approximations in the algorithm are chosen such that they do not have a significant effect on the precision.

As the MLFMA is an expansion of the Fast Multipole Method, the concept of the Fast Multipole Method is explained first. After this, the MLFMA is introduced and the building blocks of the MLFMA are explained.

4.1. Fast Multipole Method for the Electromagnetic Scattering Problem

In short, the Fast Multipole Method (FMM) is based on an expansion of the Green's function using multipoles. The method accelerates the iterative solution of boundary-integral equations significantly. The surface scatterers or basis functions are divided into groups: basis functions that lie close together are treated as if they are a single source. Because of the use of the FMM algorithm, the complexity of the matrix vector multiplication is reduced and the interaction matrix does not have to be stored.
On a more abstract level, a discrete operator works on a vector with unknown coefficients of the current distribution and the FMM simplifies the discrete operator.

In Chapter 2, the physical model for EFIE is explained for PEC materials. In this chapter, the FMM for the problem of this thesis is discussed using the notation of the Netherlands Aerospace Center (NLR). This section is therefore closely based on the derivation presented by H. van der Ven and H. Schippers in [1].
As explained in Chapter 2, for this research only the equations for the EFIE will be derived.

Interactions between basis functions in the system matrix $\mathscr{L}$ can be called 'near' or 'far'. This means that the matrix can be split in two matrices $\mathscr{L} = \mathscr{L}^{near} + \mathscr{L}^{far}$. $\boldsymbol{r}'$ is called near $\boldsymbol{r}$ if $|\boldsymbol{r} - \boldsymbol{r}'| < D$ and $\boldsymbol{r}'$ is called far from $\boldsymbol{r}$ if $|\boldsymbol{r} - \boldsymbol{r}'| > D$, with $\boldsymbol{r}$ and $\boldsymbol{r}'$ two different degrees of freedom and D a specific distance.

A more exact method of explaining if a degree of freedom is near or far: if a degree of freedom i is in box b (a group of degrees of freedom), then all degrees of freedom j are near i if they are also in box b, or if they are in a neighbour box of b.

When two groups are in each other's far field, the calculations can be reduced to the midpoints of the boxes instead of calculations for every pair of nodes within that box. As for now, it is not relevant how this happens exactly. Something very important is the fact that if the parameters in the method for this problem are chosen right, there is almost no loss of accuracy, which makes this method a good improvement. When two clusters are not in each other's far field, the traditional MoM is used.

The Fast Multipole Method consists of three phases: the *aggregation* phase, the *translation* phase and the *disaggregation* phase.

In the *aggregation* phase, all contributions of the sources in a group are summed into a radiation pattern. This radiation pattern starts from the center of a group. From this center, outgoing waves are converted into incoming waves. This is called the *translation* step. In this step, the dense matrix that represents the interaction between individual members in the two groups can be manipulated to a diagonal matrix.

In the *disaggregation* phase, the unknown coefficients of the basis functions in a specific group are found.

The field at $\boldsymbol{r}$ from a source at $\boldsymbol{r}'$ can be computed using two displacements. The relation between the locations $\boldsymbol{r}, \boldsymbol{r}'$ and the displacement D, d are shown in Figure 4.1. $\boldsymbol{r}-\boldsymbol{r}'$ can also be written as $\boldsymbol{r}-\boldsymbol{r}' = \boldsymbol{r}-\boldsymbol{c}_m+\boldsymbol{c}_m-\boldsymbol{c}_{m'}+\boldsymbol{c}_{m'}-\boldsymbol{r}'$ where $|\boldsymbol{r}-\boldsymbol{c}_m| = d_m$ is the distance between $\boldsymbol{r}$ and its group center $\boldsymbol{c}_m$, $\boldsymbol{c}_m-\boldsymbol{c}_{m'} = \boldsymbol{r}_{mm'}$ the distance between the group centers $\boldsymbol{c}_m$ and $\boldsymbol{c}_{m'}$ and $|\boldsymbol{c}_{m'}-\boldsymbol{r}'| = d_{m'}$ the distance between $\boldsymbol{r}'$ and its group center $\boldsymbol{c}_{m'}$. D is close to $|\boldsymbol{r}-\boldsymbol{r}'|$, such that d is small and $d < D$ [19].
In Figure 4.1 the relations between locations and displacements can be seen, where $d = \boldsymbol{r}-\boldsymbol{c}_m+\boldsymbol{c}_{m'}-\boldsymbol{r}'$.

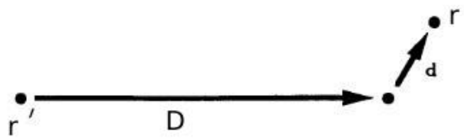

Figure 4.1: Relations between locations and displacements.

The addition theorem states the following:

$$\frac{e^{-ik|\boldsymbol{D}+\boldsymbol{d}|}}{|\boldsymbol{D}+\boldsymbol{d}|} = -ik\sum_{l=0}^{\infty}(-1)^l(2l+1)j_l(kd)h_l^{(2)}(kD)P_l(\hat{\boldsymbol{d}}\cdot\hat{\boldsymbol{D}}). \tag{4.1}$$

Here, j_l is a spherical Bessel function of the first kind, $h_l^{(2)}$ is a spherical Hankel function of the second kind and P_l is a Legendre polynomial.

The term $j_l P_l$ from (4.1) can be expanded in propagating plane waves. It is advantageous to working with plane waves, because the waves can be split in incoming and outgoing waves: $e^{-ikd} = e^{-ik(\boldsymbol{r}-\boldsymbol{c}_m)}e^{ik(\boldsymbol{r}'-\boldsymbol{c}_{m'})}$. The term $j_l P_l$ expanded in propagating plane waves is equal to the following:

$$4\pi(-i)^l j_l(kd)P_l(\hat{\boldsymbol{d}}\cdot\hat{\boldsymbol{D}}) = \int_{\hat{S}} e^{-ik\hat{\boldsymbol{k}}\cdot\boldsymbol{d}}P_l(\hat{\boldsymbol{k}}\cdot\hat{\boldsymbol{D}})d\hat{\boldsymbol{k}}, \tag{4.2}$$

where $\hat{S}$ is the unit sphere and $\hat{\boldsymbol{k}}$ the wave directions.

This expression can be substituted in (4.1) and summation and integration can be interchanged:

$$\frac{e^{-ik|\boldsymbol{D}+\boldsymbol{d}|}}{|\boldsymbol{D}+\boldsymbol{d}|} \approx \frac{k}{4\pi}\int_{\hat{S}} e^{-ik\hat{\boldsymbol{k}}\cdot\boldsymbol{d}}\sum_{l=0}^{K}(-i)^{l+1}(2l+1)h_l^{(2)}(kD)P_l(\hat{\boldsymbol{k}}\cdot\hat{\boldsymbol{D}})d\hat{\boldsymbol{k}}. \tag{4.3}$$

Calculation of an infinite sum is impossible, so the series has to be truncated. The series is divergent, so the sum should not be stopped too early, but also not too late. Stopping the sum at the right time, does not give any problem and gives a relatively small error. Information about errors in the FMM can be found in [30].
It is allowed to interchange the summation and integration in (4.1), because of the fact that the sum is finite.

The transfer function α is defined by:

$$\alpha(k\hat{\boldsymbol{k}},\boldsymbol{D}) = \frac{k}{4\pi}\sum_{l=0}^{K}(-i)^{l+1}(2l+1)h_l^{(2)}(kD)P_l(\hat{\boldsymbol{k}}\cdot\hat{\boldsymbol{D}}). \tag{4.4}$$

The approximation in (4.3) becomes:

$$\frac{e^{-ik|\boldsymbol{D}+\boldsymbol{d}|}}{|\boldsymbol{D}+\boldsymbol{d}|} := \int_{\hat{S}} e^{-ik\hat{\boldsymbol{k}}\cdot\boldsymbol{d}} \alpha(k\hat{\boldsymbol{k}}, \boldsymbol{D}) d\hat{\boldsymbol{k}}. \tag{4.5}$$

D is chosen such that $r - r' - D$ is small. In this way, for modest values of K, the method is still accurate. When the contribution of different degrees of freedom are combined, the FMM accelerates the calculation.

In order to give an idea of the division of the basis functions, Figure 4.2 is used [19]. To keep it simple, the small boxes in the figure consist just of one basis function. It is assumed that there are N basis functions in total, which are divided into G different groups. This means that there are $\frac{N}{G}$ basis functions in one group m. β is called the number of the basis function in a specific group, such that (m, β) could be used as notation for all different basis functions and $n(m, \beta)$ is equal to numbers from 0 to N. The center of a group m is called c_m.

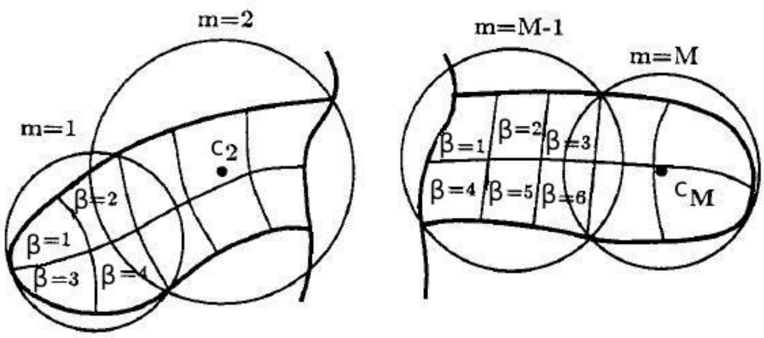

Figure 4.2: Simple example to group basis functions.

When two basis functions have regions of support that are separated by a distance not larger than $d = \frac{1}{4}\lambda$, they are called nearby basis functions. For the group pairs with nearby basis functions, the sparse near matrix $\mathscr{L}^{near}$ is constructed using the MoM.

In Chapter 3, $\mathscr{L}$ was derived in (3.4). Using the Fourier Transform, the derivatives become multiplications and this gives the following matrix entries $\mathscr{L}_{ij}$ for the vector problem:

$$\mathscr{L}_{ij} \approx ik \int_{\hat{S}} \left(\int_S \boldsymbol{f}_i e^{-ik\hat{\boldsymbol{k}}(r-c_m)} \, \mathrm{d}\boldsymbol{r} \right) \alpha(k\hat{\boldsymbol{k}}, \boldsymbol{r}_{mm'})(\mathbb{I} - \hat{\boldsymbol{k}}\hat{\boldsymbol{k}}) \left(\int_S \boldsymbol{f}_j e^{ik\hat{\boldsymbol{k}}\cdot(r'-c_{m'})} \, \mathrm{d}\boldsymbol{r}' \right) \mathrm{d}\hat{\boldsymbol{k}}, \tag{4.6}$$

where $\boldsymbol{f}_i$ and $\boldsymbol{f}_j$ are basis functions introduced in Figure 3.1 in Chapter 3 and $\mathbb{I} - \hat{\boldsymbol{k}}\hat{\boldsymbol{k}}$ is the dyadic notation for the projection operator $\boldsymbol{v} \mapsto \boldsymbol{v} - (\hat{\boldsymbol{k}} \cdot \boldsymbol{v})\hat{\boldsymbol{k}}$. $\mathbb{I}$ is because of the basis functions $\boldsymbol{f}_i$ and $\boldsymbol{f}_j$ and $\hat{\boldsymbol{k}}\hat{\boldsymbol{k}}$ is because of the derivatives of the basis functions.

The 3 steps of the Fast Multipole Method can already be seen in this equation: the incoming waves, the transfer and the outgoing waves.

The incoming and outgoing waves, $\boldsymbol{v}^{(\mathrm{in})}_{im}(\hat{\boldsymbol{k}})$ and $\boldsymbol{v}^{(\mathrm{out})}_{jm'}(\hat{\boldsymbol{k}})$, respectively, are given by:

$$\boldsymbol{v}^{(\mathrm{in})}_{im}(\hat{\boldsymbol{k}}) = (\boldsymbol{v}^{(\mathrm{in})}_{i}(\hat{\boldsymbol{k}}))_m := \int_S \boldsymbol{f}_i e^{-ik\hat{\boldsymbol{k}}\cdot(r-c_m)} \, \mathrm{d}\boldsymbol{r}, \tag{4.7}$$

$$\boldsymbol{v}^{(\mathrm{out})}_{jm'}(\hat{\boldsymbol{k}}) = (\boldsymbol{v}^{(\mathrm{out})}_{j}(\hat{\boldsymbol{k}}))_{m'} := (\mathbb{I} - \hat{\boldsymbol{k}}\hat{\boldsymbol{k}}) \int_S \boldsymbol{f}_j e^{ik\hat{\boldsymbol{k}}\cdot(r'-c'_m)} \, \mathrm{d}\boldsymbol{r}', \tag{4.8}$$

where $\boldsymbol{v}^{(\mathrm{in})}(\hat{\boldsymbol{k}}) \in \mathbb{C}^G$ and $\boldsymbol{v}^{(\mathrm{out})}(\hat{\boldsymbol{k}}) \in \mathbb{C}^G$ are both column vectors and G is the number of groups.

In (4.6) the inner product of $\boldsymbol{v}_{im}^{(\text{in})}(\hat{\boldsymbol{k}})$ and $\boldsymbol{v}_{jm'}^{(\text{out})}(\hat{\boldsymbol{k}})$ is taken, so the radial component in (4.8) does not matter and (4.7) and (4.8) are each other's complex conjugate.
The result of multiplying $(\mathbb{I}-\hat{\boldsymbol{k}}\hat{\boldsymbol{k}})$ by a vector does not depend on the wave directions $\hat{\boldsymbol{k}}$, so θ and ϕ can be used as the only components. In the interpolation step the wave directions change, so all three components are needed.

The N basis functions are divided over G localized groups, each supporting about $M = N/G$ basis functions. Let G_m be the set of basis functions belonging to group m and let B_m be all groups which are near group m.

The matrix vector product $\mathscr{L}\boldsymbol{J}$ is computed as follows:

$$\sum_{j=1}^{N}\mathscr{L}_{ij}J_j = \sum_{m'\in B_m}\sum_{j\in G_{m'}}\mathscr{L}_{ij}J_j + ik\int_{\hat{S}}\boldsymbol{v}_{im}^{(\text{in})}(\hat{\boldsymbol{k}})\cdot\left(\sum_{m'\notin B_m}\alpha_{mm'}(k\hat{\boldsymbol{k}},\boldsymbol{r}_{mm'})\sum_{j\in G_{m'}}\boldsymbol{v}_{jm'}^{(\text{out})}(\hat{\boldsymbol{k}})\right)J_j\,\mathrm{d}\hat{\boldsymbol{k}}, \tag{4.9}$$

where the first part of the formula are the near field interactions, the second part are the far field interactions and $\boldsymbol{J}$ is the unknown vector.

The integral in (3.4) can be discretized using a quadrature rule for a sphere:

$$\int f(k)d\hat{k} = \sum_{k=1}^{K} f(k_k)w_k, \tag{4.10}$$

where w_k are the quadrature weights.

The sub matrix (4.6) defines the action of degree of freedom i on degree of freedom j. The FMM transformation for all degrees of freedom i and j leads to:

$$\mathscr{L}_{ij}\to\sum_{\hat{\boldsymbol{k}}_i}(\boldsymbol{v}_{im}^{(\text{in})})^T(\hat{\boldsymbol{k}})T_{mm'}(\hat{\boldsymbol{k}})\boldsymbol{v}_{jm'}^{(\text{out})}(\hat{\boldsymbol{k}}) = \sum_{\hat{\boldsymbol{k}}_i}\left(\overline{\boldsymbol{v}_{im}^{(\text{out})}(\hat{\boldsymbol{k}})}\right)^T T_{mm'}(\hat{\boldsymbol{k}})\boldsymbol{v}_{jm'}^{(\text{out})}(\hat{\boldsymbol{k}}), \tag{4.11}$$

where $T_{ij} = \alpha(k\hat{\boldsymbol{k}}, r_{mm'})w_i \in \mathbb{C}^G$ is a dense matrix with the coefficients and information about the integration and $\left(\overline{\boldsymbol{v}_{im}^{(\text{out})}(\hat{\boldsymbol{k}})}\right)^T$ is the conjugate transpose of $\boldsymbol{v}_{im}^{(\text{out})}(\hat{\boldsymbol{k}})$. The vectors $\boldsymbol{v}_{im}^{(\text{in})}$ and $\boldsymbol{v}_{jm'}^{(\text{out})}$ are both column vectors.

The impedance matrix $\mathscr{L}$, can then be written as follows:

$$\mathscr{L} = \mathscr{L}' + \overline{V}_{FM}^T T_{FM} V, \tag{4.12}$$

where $\mathscr{L}'$ is the near field impedance matrix. For all degrees of freedom (i,j) that interact using FMM, $\overline{V}^T$, T_{FM} and V contain the factors $\overline{\boldsymbol{v}}_{jm'}^{(\text{out})}$, $T_{FM_{mm'}}$ and $\boldsymbol{v}_{jm'}^{(\text{out})}$, respectively. The different wave directions are included in matrices $\overline{V}^T$, T_{FM} and V, where T_{FM} is the transfer matrix for the FMM.

$\mathscr{L}'$ consists information about the near field interactions, calculated using the original MoM and $\overline{V}^T T_{FM} V$ consist the information of the far field interactions.

Having a closer look at (4.9) gives the dimensions of matrices $\overline{V}^T$, T_{FM} and V:

$$\begin{aligned}\text{Linear operator } \boldsymbol{v}_{jm'}^{(\text{out})}(\hat{\boldsymbol{k}}_k) = V_{m'kj} &\Rightarrow V\in\mathbb{C}^{GK\,\text{x}\,N\,\text{x}\,3}\\ \text{Linear operator } \alpha_{mm'}(k\hat{\boldsymbol{k}}_k, r_{mm'})w_k = T_{FM_{mkm'k}} &\Rightarrow T_{FM}\in\mathbb{C}^{GK\,\text{x}\,GK\,\text{x}\,3\,\text{x}\,3}\\ \text{Linear operator } \boldsymbol{v}_{im}^{(\text{in})}(\hat{\boldsymbol{k}}_k) = V_{ikm} &\Rightarrow \overline{V}\in\mathbb{C}^{N\,\text{x}\,GK\,\text{x}\,3}\end{aligned}$$

where K is the number of wave directions and G the number of groups. This means that the final matrix is $\mathscr{L}\in\mathbb{C}^{N\,\text{x}\,N}$.

Having a closer look at the structure of the matrices gives the following result for matrix V:

$$V = \begin{pmatrix} V_1 \\ \vdots \\ V_G \end{pmatrix},$$

where $V_1, ..., V_G \in \mathbb{C}^{K \times N}$, different for each group.

The transfer matrix T_{FM} is a block-diagonal matrix with the following structure:

$$T_{FM} = \begin{pmatrix} \boldsymbol{T}_{FM_1} & 0 & \cdots & 0 \\ 0 & \boldsymbol{T}_{FM_2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \boldsymbol{T}_{FM_G} \end{pmatrix},$$

where $\boldsymbol{T}_{FM_1}, ..., \boldsymbol{T}_{FM_G} \in \mathbb{C}^{K \times K}$, different for each group.

This dense matrix $\overline{V}^T T_{FM} V$ has to be multiplied by J. This is done by first calculating VJ, afterwards multiplying T_{FM} by the result of VJ and continuing like this. The solve part is the part of the FMM that uses most of the computation time.

Those computations give the following matrix dimensions:

$$\begin{aligned} (VJ) &\in \mathbb{C}^{GK \times 3}, \\ (T_{FM} VJ) &\in \mathbb{C}^{GK \times 3}, \\ (\overline{V}^T T_{FM} VJ) &\in \mathbb{C}^{N \times 3}. \end{aligned}$$

After these specifications of the matrices and the matrix vector multiplications, it is easier to see what can be improved when performing the calculations on a GPU.

4.2. Multi Level Fast Multipole Algorithm

The Multi Level Fast Multipole Algorithm is a recursive form of the Fast Multipole Method. It makes use of different levels with different group-sizes, where groups (or boxes) are made out of smaller groups. The new terms interpolation and anterpolation are used to switch between different group-sizes [1]. The interpolation and anterpolation are needed because of the fact that the group diameter increases with coarser levels and therefore the number of plane wave directions required for an accurate solution increases as well. This accelerates the computation speed even more compared to the FMM.
In the Multi Level Fast Multipole Algorithm, various calculations can be reused. For example once matrix V is known, it can be reused and does not have to be calculated again.

As explained in the previous section, in the FMM $\mathscr{L}'$ is still calculated exactly. Therefore, it is preferable that $\mathscr{L}'$ contains very few elements and therefore a fine grid is needed. For the far interactions, it is preferable to have a coarser grid, to have larger groups and less interactions. This is why the Multi Level part is added to the FMM.

An example to understand the idea of the Fast Multipole Method better, is a telephone network. There can be direct lines between all users, but there can also be hubs to connect groups of users to other groups of users. Messages can be combined, which makes the process much more efficiently.

The explanation above is a very brief explanation of the Multi Level part of the MLFMA. To explain better how the method works, Figure 4.3 shows a schematic example of the MLFMA [1].

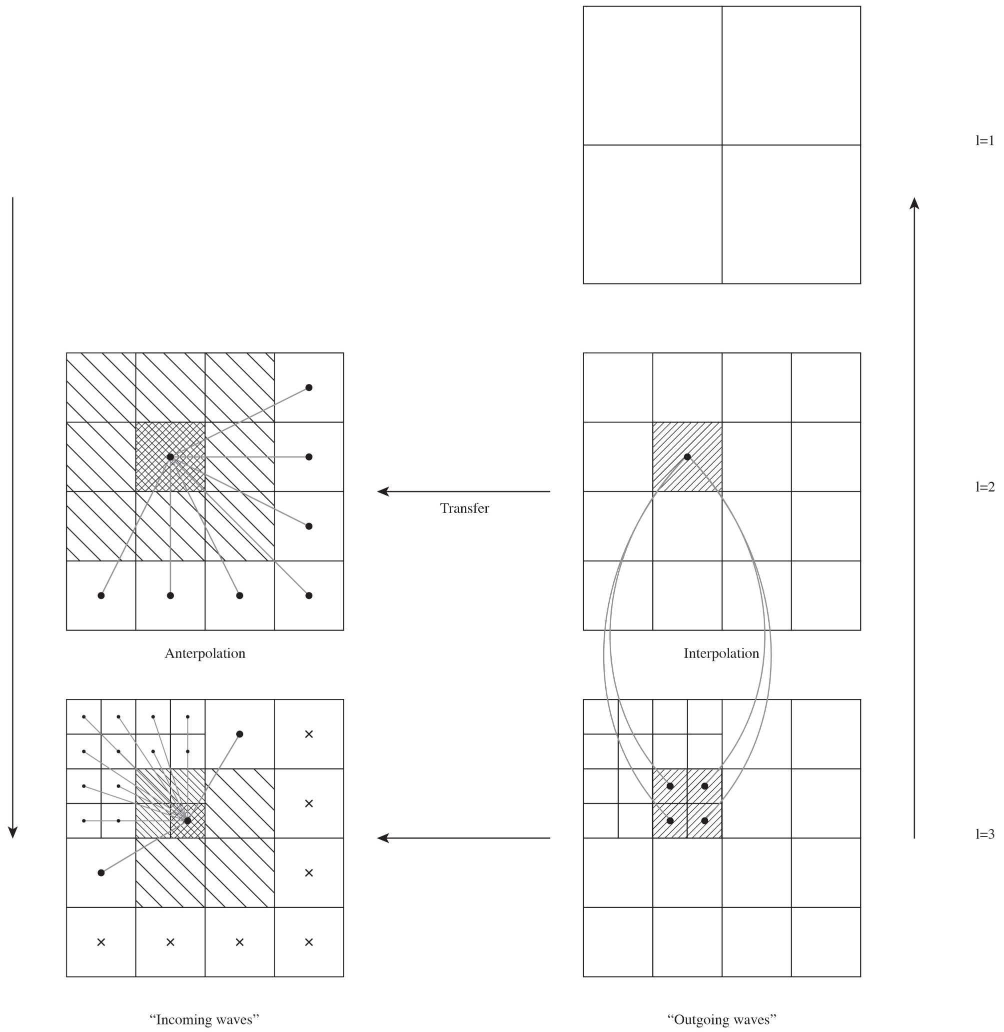

Figure 4.3: Schematic example of the MLFMA.

The classification at the different levels is based on an octree division. The finest level contains the smallest cubes (or boxes) in the octree and the cubes (or boxes) contain at most a few degrees of freedom. In a coarser level, one cube contains eight fine cubes. Continuing like this, the coarsest level just consists of a single cube, encompassing the complete scatterer. Note that the figures are in 2D and so based on a quadtree division.

Figure 4.3 shows the outline of the MLFMA. Starting with the outgoing waves on the level of the degrees of freedom and interpolating (agglomeration) this into clusters until level $l = 2$, brings the outgoing wave to the coarsest level. At this level it is not possible to go to larger clusters, because at this level there are no far field interactions at all. Later in this section it is explained why.

The transfer operator transfers the outgoing waves at each level. Then anterpolation changes the clusters back to finer levels and finally to the level of degrees of freedom. At the finest level the outgoing wave is translated into an incoming wave.

The distance between the basis functions is important for the level on which the interactions are calculated: interactions between scatterers at large distance are computed on a coarse level, while interactions between scatterers at a smaller distance are computed on a finer level.

In the anterpolation phase, at each level the far field interactions are calculated. The interaction with all degrees of freedom that are neighbours of neighbours are calculated at that level. At a finer level, those far field interactions do not have to be calculated again.

Having the schematic representation of the MLFMA of Figure 4.3 in mind, the MLFMA is explained in more detail. At the finest level $l = L$, the outgoing waves $\boldsymbol{v}^{(0)}$ and $\boldsymbol{w}^{(0)}$ for all group centers at this level are calculated. After this calculation, the outgoing waves are interpolated to the group centers at the levels $l = L-1, \ldots, 2$. At each level, starting from the coarsest level $l = 2$, the interactions between the groups at this level are computed. Now, at each level the incoming waves computed before are inversely interpolated (or anterpolated) to level $l = l+1$ and added to the incoming waves computed directly at this level. At the finest level $l = L$ the near interactions are computed using the MoM.

In the previous section about the FMM it is explained that the geometric size determines the accuracy in which the integral over the unit sphere has to be determined.

As explained in the previous section, at a specific level, only the interactions between degrees of freedom that are neighbours of neighbours are calculated. This means that for a given group m at level l, only the interactions with groups that are children of the neighbours of the parents of group m (which are not a neighbour of m) are calculated. In this way, the interactions are computed in an efficient way. Other groups at this level are further away from group m and are computed at a coarser level.
As can be seen in Figure 4.3, $l = 2$ is the coarsest active level. This is because of the fact that level 1 consists of eight cubes which are all neighbours, so the FMM cannot be applied at this level.

At all finer levels, the incoming waves consist of the interactions at this level and the incoming waves projected from the next coarser level. The anterpolation step is the transpose of the interpolation step.
At the finest level $l = L$, the incoming waves are combined with the near interactions computed with the MoM.

During the interpolation step, the plane wave expansions from each child to its parent are interpolated (using Lagrange interpolation). After this, the interpolated plane waves are translated from the center of the child to the center of the parent and finally the plane waves over all children are summed.

The translation step consists of a pre-multiplication for a given wave direction $\hat{\boldsymbol{k}}$:

$$e^{-ik\hat{\boldsymbol{k}}(\boldsymbol{c}_l^i - \boldsymbol{c}_{l-1}^j)}, \tag{4.13}$$

where $\boldsymbol{c}_{l-1}^j$ is the group center of the j-th box at level $l-1$ (parent box b_{l-1}^j), and box b_l^i at level l is one of the children of box b_{l-1}^j.

For a given set of data points $\{g_k = g(x_k) | 0 \le k \le N\}$, the Lagrange interpolated function $\tilde{g}$ is defined by:

$$\tilde{g}(x) = \sum_{k=0}^{N} l_k(x) g_k, \tag{4.14}$$

where the interpolators l_k are defined as:

$$l_k(x) = \Pi_{i=0, i \neq k}^{N} \frac{x - x_i}{x_k - x_i}. \tag{4.15}$$

The interpolation is two-dimensional, but is split in two one-dimensional interpolations with spherical coordinates θ and ϕ.

The interpolator is the same for all groups at a given level, so the interpolation matrices for all levels can be precomputed and stored.

The group-to-group interactions between two groups m and m' at a given level consist of the pre-multiplication with $\alpha(k\hat{\boldsymbol{k}}, \boldsymbol{c}_m - \boldsymbol{c}_{m'})$ for a given wave direction $\hat{\boldsymbol{k}}$. For this given wave direction, the transfer operator only depends on the distance vector $\boldsymbol{c}_m - \boldsymbol{c}_{m'}$.

The influence list consists at most of all boxes contained in the three box layers around the given box: $(3+1+3)^3 = 7^3$ boxes. The box itself and its neighbours are excluded from the influence list and therefore the maximum number of possible paths $\boldsymbol{c}_m - \boldsymbol{c}_{m'}$ is equal to $7^3 - 27 = 316$.

For the MLFMA, the algorithm can be given in five steps. First, at the finest level, the outgoing waves are calculated as before:

$$\boldsymbol{v}^{(\text{out})}_{m,l_{max}}(\hat{\boldsymbol{k}}^{l_{max}}_i) := \sum_{j \in G^{l_{max}}_m} (\mathbb{I} - \hat{\boldsymbol{k}}^{l_{max}}_i \hat{\boldsymbol{k}}^{l_{max}}_i) \int_S \boldsymbol{f}_j e^{ik\hat{\boldsymbol{k}}^{l_{max}}_i \cdot (\boldsymbol{r} - \boldsymbol{c}^{l_{max}}_m)} d\boldsymbol{r} J_j, \tag{4.16}$$

where $m = 1, .., N^{l_{max}}$, with $N^{l_{max}}$, is the number of groups at level l and $i = 1, ..., K_{l_{max}}$, with $K_{l_{max}}$, is the number of integration points used in the k-space integral at level l.

The aggregation from fine to coarser levels for $l = l_{max}, ..., 2$ is given as follows:

$$\boldsymbol{v}^{(\text{out})}_{m,l}(\hat{\boldsymbol{k}}^l_j) := \sum_{m_c \in G^l_m} e^{ik\hat{\boldsymbol{k}}^l_j \cdot (c^{l+1}_{mc} - c^l_m)} \tilde{\boldsymbol{v}}^{(\text{out})}_{m_c,l+1}(\hat{\boldsymbol{k}}^l_j), \tag{4.17}$$

where $m = 1, ..., N^l$, with N^l the number of groups at level l, $j = 1, ..., K_l$ the directions at the coarser levels and $\tilde{\boldsymbol{v}}^{(\text{out})}_{m_c,l+1}(\hat{\boldsymbol{k}})$ the interpolated far fields at the finer level $\boldsymbol{v}^{(\text{out})}_{m_c,l+1}(\hat{\boldsymbol{k}}^{l+1}_i)$, where m_c is the center of one of the child boxes c.

After this aggregation step, the far field radiation patterns are transferred to the groups in the influence list at each level as follows:

$$\boldsymbol{v}^{(\text{out})}_{m_2,l}(\hat{\boldsymbol{k}}^l_i) := \sum_{m_1 \in G^{int,l}_{m_2}} w^l_i \alpha(k\hat{\boldsymbol{k}}^l_i, \boldsymbol{c}^l_{m_2} - \boldsymbol{c}^l_{m_1}) \boldsymbol{v}^{(\text{out})}_{m_1,l}(\hat{\boldsymbol{k}}^l_i), \tag{4.18}$$

where $m_2 = 1, ..., N^l$ and $l = l_{max}, .., 2$.

The incoming waves should be disaggregated from the coarse levels to the finest level:

$$\boldsymbol{v}^{(\text{out})}_{m,l}(\hat{\boldsymbol{k}}^l_j) := e^{ik\hat{\boldsymbol{k}}^l_j \cdot (\boldsymbol{c}^{l-1}_{mp} - \boldsymbol{c}^l_m)} \tilde{\boldsymbol{v}}^{(\text{out})}_{m_p,l-1}(\hat{\boldsymbol{k}}^l_j) + \boldsymbol{v}^{(\text{out})}_{m,l}(\hat{\boldsymbol{k}}^l_j), \tag{4.19}$$

where $m_p = 1, ..., N^{l-1}$ and $\tilde{\boldsymbol{v}}^{(\text{out})}_{m_p,l-1}$ with $m \in m_p$ the interpolated incoming waves.

The final incoming wave is the following:

$$O_j := ik \sum_{i=1}^{K_{l_{max}}} (\mathbb{I} - \hat{\boldsymbol{k}}^{l_{max}}_i \hat{\boldsymbol{k}}^{l_{max}}_i) \int_S \boldsymbol{f}_j e^{ik\hat{\boldsymbol{k}}^{l_{max}}_i \cdot (\boldsymbol{r} - \boldsymbol{c}^{l_{max}}_i)} d\boldsymbol{r} \cdot \boldsymbol{v}^{(\text{out})}_{m,l_{max}}(\hat{\boldsymbol{k}}^{l_{max}}_i), \tag{4.20}$$

where $j = 1, ..., N$.

The nature of the wave changes from outgoing to incoming when the wave arrives in the group center of the degree of freedom on the finest level.

To give a better overview of the five steps of the MLFMA, the impedance matrix $\mathscr{L}$ can be written as matrix multiplications:

$$
\begin{aligned}
\mathscr{L} &= \mathscr{L}' + \overline{V}^T T_{l_{max}} V + \overline{V}^T A_{l_{max}-1}^{l_{max}T} T_{l_{max}-1} A_{l_{max}}^{l_{max}-1} V + \ldots \\
&= \mathscr{L}' + \overline{V}^T (T_{l_{max}} + A_{l_{max}-1}^{l_{max}T} T_{l_{max}-1} A_{l_{max}}^{l_{max}-1}) V + \ldots \\
&= \mathscr{L}' + \overline{V}^T (T_{l_{max}} + A_{l_{max}-1}^{l_{max}T} (T_{l_{max}-1} + A_{l_{max}-2}^{l_{max}-1T} T_{l_{max}-2} A_{l_{max}-1}^{l_{max}-2}) A_{l_{max}}^{l_{max}-1}) V + \ldots \\
&= \mathscr{L}' + \overline{V}^T (T_{l_{max}} + A_{l_{max}-1}^{l_{max}T} (T_{l_{max}-1} + \ldots (\ldots + A_{l=2}^{l=3T} T_{l=2} A_{l=3}^{l=2}) \ldots) A_{l_{max}}^{l_{max}-1}) V, \qquad (4.21)
\end{aligned}
$$

where matrix A is the agglomeration step, so this matrix contains the information about the interpolation and anterpolation and the shift.

The structure of each matrix in (4.21) is given to get an idea of the dimensions of the matrices:

$$
\begin{aligned}
\text{Linear operator } \boldsymbol{v}_{m,l_{max}}^{(\text{out})}(\hat{\boldsymbol{k}}_i^{l_{max}}), \forall \hat{\boldsymbol{k}}^{l_{max}} = V_{m'j} &\quad\Rightarrow\quad V \in \mathbb{C}^{G_{l_{max}} K_{l_{max}} \text{ x } N} \\
\text{Linear operator } \boldsymbol{v}_{mj,l_{max}-1}^{(\text{out})} = A_{l_{max}}^{l_{max}-1} &\quad\Rightarrow\quad A_{l_{max}}^{l_{max}-1} \in \mathbb{C}^{G_{l_{max}-1} K_{l_{max}-1} \text{ x } G_{l_{max}} K_{l_{max}}} \\
&\vdots \\
\text{Linear operator } \boldsymbol{v}_{mj,l=2}^{(\text{out})} = A_{l=3}^{l=2} &\quad\Rightarrow\quad A_{l=3}^{l=2} \in \mathbb{C}^{G_{l=2} K_{l=2} \text{ x } G_{l=3} K_{l=3}} \\
\text{Linear operator } \boldsymbol{v}_{m,l=2}^{(\text{out})}(\hat{\boldsymbol{k}}_i^{l=2}), \forall \hat{\boldsymbol{k}}^{l=2} = T_{l=2} &\quad\Rightarrow\quad T_{l=2} \in \mathbb{C}^{G_{l=2} K_{l=2} \text{ x } G_{l=2} K_{l=2}} \\
&\vdots \\
\text{Linear operator } \boldsymbol{v}_{m,l_{max}}^{(\text{out})}(\hat{\boldsymbol{k}}_i^{l_{max}}), \forall \hat{\boldsymbol{k}}^{l_{max}} = T_{l_{max}} &\quad\Rightarrow\quad T_{l_{max}} \in \mathbb{C}^{G_{l_{max}} K_{l_{max}} \text{ x } G_{l_{max}} K_{l_{max}}} \\
\text{Linear operator } \boldsymbol{v}_{mj,l=2}^{(\text{out})} = A_{l=2}^{l=3} &\quad\Rightarrow\quad A_{l=2}^{l=3} \in \mathbb{C}^{G_{l=3} K_{l=3} \text{ x } G_{l=2} K_{l=2}} \\
&\vdots \\
\text{Linear operator } \boldsymbol{v}_{mj,l_{max}-1}^{(\text{out})} = A_{l_{max}-1}^{l_{max}} &\quad\Rightarrow\quad A_{l_{max}-1}^{l_{max}} \in \mathbb{C}^{G_{l_{max}} K_{l_{max}} \text{ x } G_{l_{max}-1} K_{l_{max}-1}} \\
\text{Linear operator } \boldsymbol{v}_{m,l_{max}}^{(\text{in})}(\hat{\boldsymbol{k}}_i^{l_{max}}), \forall \hat{\boldsymbol{k}}^{l_{max}} = V_{im} &\quad\Rightarrow\quad V \in \mathbb{C}^{N \text{ x } G_{l_{max}} K_{l_{max}}}.
\end{aligned}
$$

This means that the final matrix is $\mathscr{L} \in \mathbb{C}^{N \text{ x } N}$.

Matrices V and T have the same structure as explained in the Section 4.1 about the FMM method. The agglomeration matrix A_{l-1}^{l} is constructed in the same way as matrix A_l^{l-1}: calculation of the Lagrange interpolation matrix $I_{l-1}^{l} \in \mathbb{R}^{G_l K_l \text{ x } G_l K_{l-1}}$ and afterwards the phase shift and sum over all child groups. The Lagrange interpolation matrix I_{l-1}^{l} looks as follows:

$$
I_{l-1}^{l} = \begin{pmatrix} \mathscr{I}_{l-1}^{l} & 0 & \cdots & 0 \\ 0 & \mathscr{I}_{l-1}^{l} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \mathscr{I}_{l-1}^{l} \end{pmatrix},
$$

where $\mathscr{I}_{l-1}^{l} \in \mathbb{R}^{K_l \text{ x } K_{l-1}}$. This matrix $\mathscr{I}_{l-1}^{l}$ is the same for each group. After the phase shift and the sum, matrix $A_{l-1}^{l} \in \mathbb{C}^{G_l K_l \text{ x } G_{l-1} K_{l-1}}$ is obtained.

The agglomeration matrix A_l^{l-1} is the transpose of matrix A_{l-1}^{l} and I_l^{l-1} is the transpose of matrix I_{l-1}^{l}.

The final matrix I_l^{l-1} looks as follows:

$$I_l^{l-1} = \begin{pmatrix} \mathscr{I}_l^{l-1} & 0 & \cdots & 0 \\ 0 & \mathscr{I}_l^{l-1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \mathscr{I}_l^{l-1} \end{pmatrix},$$

where $\mathscr{I}_l^{l-1} \in \mathbb{R}^{K_{l-1} \text{ x } K_l}$. This matrix $\mathscr{I}_l^{l-1}$ is the same for each group. Finally, matrix $A_l^{l-1} \in \mathbb{C}^{G_{l-1}K_{l-1} \text{ x } G_l K_l}$ is obtained.

The matrix for the far interactions in the second part of (4.21) has to be multiplied by J. This is done by first calculating VJ, afterwards multiplying $A_{l_{max}}^{l_{max}-1}$ by the result of VJ and continuing like this.

Those calculations give the following matrix dimensions:

$$\begin{array}{rcl}
(VJ) & \in & \mathbb{C}^{G_{l_{max}} K_{l_{max}} \text{ x } 1} \\
(A_{l_{max}}^{l_{max}-1} VJ) & \in & \mathbb{C}^{G_{l_{max}-1} K_{l_{max}-1} \text{ x } 1} \\
& \vdots & \\
(A_{l=4}^{l=3} ... A_{l_{max}}^{l_{max}-1} VJ) & \in & \mathbb{C}^{G_{l=3} K_{l=3} \text{ x } 1} \\
((T_{l=3} + A_{l=2}^{l=3^T} T_{l=2} A_{l=3}^{l=2}) A_{l=4}^{l=3} ... A_{l_{max}}^{l_{max}-1} VJ) & \in & \mathbb{C}^{G_{l=3} K_{l=3} \text{ x } 1} \\
(A_{l=3}^{l=4^T} (T_{l=3} + A_{l=2}^{l=3^T} T_{l=2} A_{l=3}^{l=2}) A_{l=4}^{l=3} ... A_{l_{max}}^{l_{max}-1} VJ) & \in & \mathbb{C}^{G_{l=4} K_{l=4} \text{ x } 1} \\
& \vdots & \\
(A_{l_{max}-1}^{l_{max}^T} ... A_{l=3}^{l=4^T} (T_{l=3} + A_{l=2}^{l=3^T} T_{l=2} A_{l=3}^{l=2}) A_{l=4}^{l=3} ... A_{l_{max}}^{l_{max}-1} VJ) & \in & \mathbb{C}^{G_{l_{max}} K_{l_{max}} \text{ x } 1} \\
(\overline{V}^T A_{l_{max}-1}^{l_{max}^T} ... A_{l=3}^{l=4^T} (T_{l=3} + A_{l=2}^{l=3^T} T_{l=2} A_{l=3}^{l=2}) A_{l=4}^{l=3} ... A_{l_{max}}^{l_{max}-1} VJ) & \in & \mathbb{C}^{N \text{ x } 1}.
\end{array}$$

After this specification of the matrices and the matrix vector multiplications, it is easier to see what can be improved when performing the calculations on a GPU.

4.3. Linear Solver

The NLR in-house developed algorithm for high-frequency scattering analysis for large objects (up to 30 million unknowns) can be used. This algorithm solves the resulting linear system using the Generalized Minimal Residual (GMRES) solver, combined with a preconditioner based on the near interaction matrix. EFIE with the same test functions and basis functions obtains a symmetric matrix. For complex and symmetric matrices, too little information is known to use a more precise solver: the matrix should have been hermition. More information about the algorithm can be found in [1].

GMRES is used for solving sparse linear systems. It minimizes the residual over the Krylov subspace span$\{r_0, Ar_0, A^2 r_0, ..., A^i r_0\}$, with $r_0 = b - Ax_0$. The advantages of a GMRES solver are its excellent convergence characteristics and the fact that the memory use only depends on the size of the Krylov space in the GMRES solver. An important problem of the GMRES solver is the fact that the basis for the Krylov subspace must be stored in the iterative process, which demands memory space. Because of the fact that it is allowed to work with non-symmetrical matrices, each new vector has to be orthogonalized against all previously generated basis vectors, which results in long recurrences.

A flowchart of the GMRES method can be seen in Figure 4.4.

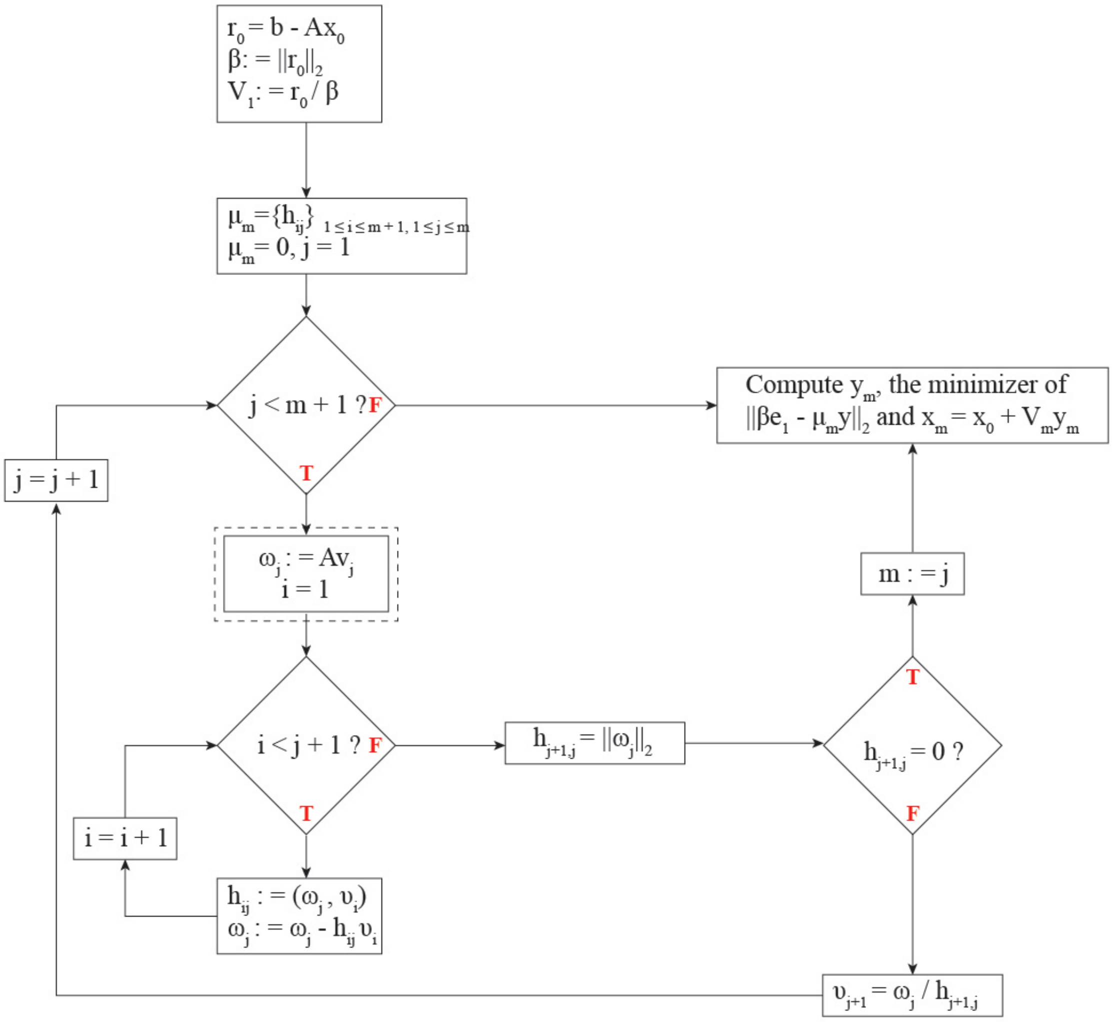

Figure 4.4: Flowchart of the GMRES method.

The MLFMA is used in the dotted area. To make it clear what happens in this specific part, a flowchart of the MLFMA is shown in Figure 4.5.

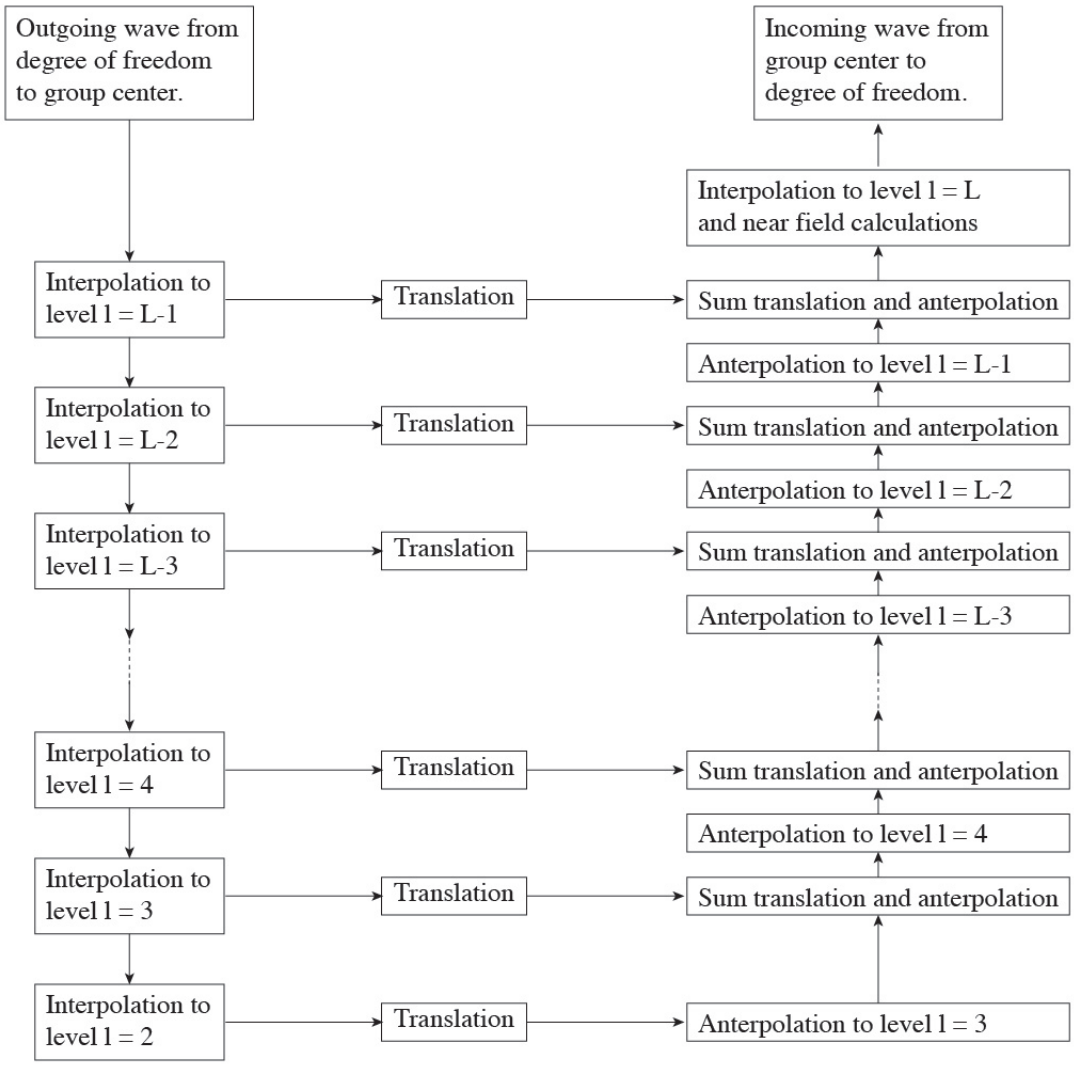

Figure 4.5: Flowchart of the MLFMA.

The different components of the MLFMA use substantially more memory compared to the Krylov basis. The Krylov basis uses a very little amount of memory compared to the matrices that have to be stored for the MLFMA.

4.4. Overview Building Blocks MLFMA

In Sections 4.2 it is explained that the MLFMA is a method that splits the interactions in two parts: far and near field interactions. The near interaction matrix is still calculated using the original MoM method. For the far interactions, a few matrix vector multiplications have to be performed.

The building blocks of the MLFMA consist of four different components, which are discussed in more detail in this section.

First of all, the operators for the MLFMA have to be determined. This means that matrices T_l, V, $\overline{V}$, A^l_{l-1} and A^{l-1}_l have to be constructed. Matrix $T_l \in \mathbb{C}^{G_l K_l \times G_l K_l}$ is a sparse block-diagonal matrix that contains information for different wave directions and contains information about the transfer. To construct this matrix, several Hankel functions and Legendre polynomials have to be calculated. Matrix $V \in \mathbb{C}^{N \times G_l K_l}$ is a dense matrix that contains information about the incoming or outgoing waves and requires the computation of various exponential functions. Interpolation (or anterpolation) matrix $A^{l-1}_l \in \mathbb{C}^{G_{l-1} K_{l-1} \times G_l K_l}$ is a sparse matrix that contains information about the agglomeration. For the interpolation step, a local interpolation method is used. This method applies the same function repeatedly to a small amount of the total data set. Again, several exponential functions have to be calculated to construct this interpolation matrix. It is possible to perform the construction of those matrices in parallel, as the matrices are not inherently sequential (the indices are not connected to each other).

Local interpolation techniques apply the same function repeatedly to a small portion of the total set of sample points for which data has been observed.

Secondly, the construction of the near field matrix $\mathscr{L}'$ can be performed in parallel, as this matrix computation is also not inherently sequential: each entry can be computed independent of the others. Also the matrix vector multiplications can be performed in parallel. The construction of the sparse preconditioner is also parallelizable.

The wave directions constitute a discretization of the unit sphere. The interpolation is actually two-dimensional, and as explained in Section 4.2 is split in two one-dimensional interpolations.

Because of the fact that the basis functions are vectors, the outgoing waves are also vectors. It is coincidental that the vectors are tangent to the airplane and the sphere.

Solving the linear system using GMRES consists of two phases: the matrix vector multiplication and the construction of the basis of the Krylov space. The matrix vector product can be parallelized and the construction of the Krylov basis is inherently sequential, so cannot be parallelized as a whole. Therefore, the focus of this thesis is the parallelization of the matrix vector multiplications of the MLFMA.

4.5. Message Passing Interface Parallelization of MLFMA

The Netherlands Aerospace Center uses a Message Passing Interface (MPI) parallelization method for the MLFMA. This means parallelization of the method on multiple CPU's. In Chapter 5, the differences between the use of an MPI for a multi core CPU machine and parallelization on Graphics Processing Units (GPU's) are explained.

To parallelize an algorithm, data should be divided over different Central Processing Units (CPU's). A problem that arises when working with the MPI parallelization method, is the fact that some parts of the MLFMA are suitable to parallelize over the groups and other parts of the MLFMA are suitable to parallelize over the wave directions. This means that a choice has to be made if the parallelization is performed over the groups or over the wave directions. In both cases data transfer is needed, because of the fact that not all data is present in the memory of a specific CPU.

A pure parallelization over the wave directions is inefficient during agglomeration on the fine levels, as there are but a few wave directions. A pure parallelization over the groups is inefficient on the coarse levels, as there are few groups on these levels. A mix between the two parallelizations is implemented, where parallelization over the groups is done on the fine levels, and over the wave directions on the coarse levels. If this approach needs to be extended to the GPU is one of the detailed research questions.

5

Graphics Processing Units

Graphics Processing Units (GPU's) can be used for parallel computing. Without parallel computing, a specific task can only start if the previous task has finished. On parallel computers more than one task can be performed at the same time, which can speed up the rate of performance considerably for problems that perform many of the same computations. Supercomputer performance can be obtained for relatively low costs. The algorithm must be specifically adapted (in a way which is possibly sub-optimal for a CPU) in order to realize acceleration on the GPU.

In this chapter background information about the GPU is given, such as information about threads, blocks and grids and different types of memory on the GPU. The difference between Message Passing Interface and GPU's is given and vector computers are compared to parallel computers. In Section 5.5 implementation methods for the GPU are discussed. Specifications for the Kepler K40 (GPU) and the Intel E5-2640 (CPU) finalize the chapter.

5.1. Background of Graphics Processing Units

For this thesis, CUDA (Compute Unified Device Architecture) from NVIDIA is used. The main programming language for the GPU is C, while the programming language of the MLFMA implementation at the NLR is Fortran. In order to make use of this already existing code, it is preferable to work with Fortran instead of C. This means that a CUDA Fortran compiler is needed to work with Fortran on the GPU. The compiler that is used is the Portland Fortran 16.1-0 (PGI): this is the only Portland compiler to compile Fortran code. This compiler does not work in combination with a relatively old GPU (Kepler K20), but only with a more recent GPU (Kepler K40).
For both the CPU (Intel Fortran 11.1) and the GPU the compiler of the manufacturer are used to ensure a good comparison.

A GPU is a Single Instruction Multiple Data (SIMD) computer, this means that multiple processors perform exactly the same instruction on multiple elements.

A Graphics Processing Unit (GPU) takes over tasks of a Central Processor Unit (CPU). Calculations are performed much faster on a GPU, because of the fact that it consists of thousands of small, efficient cores, while a CPU consists only of a few cores. The most intensive functions can be computed on a GPU, while other calculations can still be performed on a CPU.

In GPU's, different cores can perform a calculation at the same time. These different cores should perform the same operation for different data, so for example perform a multiplication with different input variables. Different operations cannot be performed in parallel.

GPU's can only be used for 'inner-loop' style codes, but fortunately different inner-loops can be divided over different blocks. In this case an outer-loop can be computed (partly) on a GPU.

5.1.1. Threads, Blocks and Grids

A GPU is made up of threads, blocks and grids. *Threads* calculate the same parallel computations that have to be performed, but on different parts of an array. A group of threads together is called a *block* and a group of blocks together is called a *grid*. One entire grid is handled by a single GPU chip. This chip is organized as a collection of *multiprocessors* (MPs) and those MPs are responsible for handling one or more blocks. It never happens that one block is divided over multiple MPs. Each MP is subdivided into multiple *stream processors* (SPs) and those SPs are responsible for one or more threads.

An overview of the threads, blocks and grids can be seen in Figure 5.1.

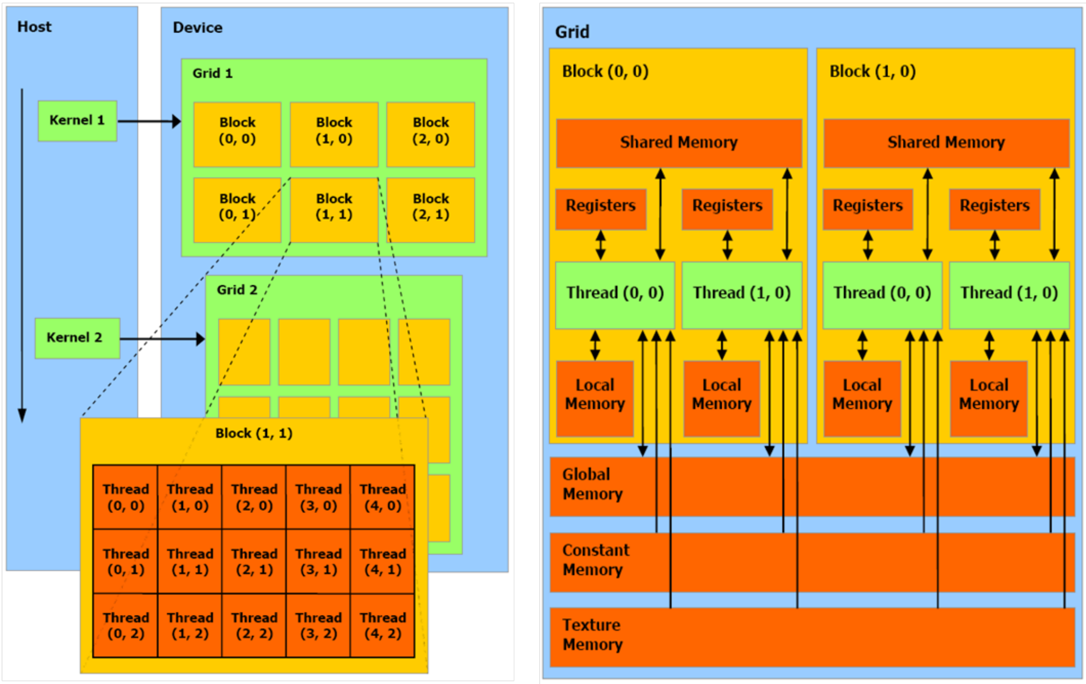

Figure 5.1: Overview of GPU architecture [31].

5.1.2. Types of Memory

There are different types of memory on a GPU and they all have advantages and disadvantages. To get optimal performance, it is necessary to make use of the right memory. Global memory, local memory, texture memory, constant memory, shared memory and register memory are different places to store data, which are briefly explained below. Besides the speed of the memory, also the scope and the lifetime of the memory are important to keep in mind. When threads need to communicate with each other, threads need to send and receive data. This communication needs an interconnection network via memory.

Register memory and *local* memory both last for the lifetime of the thread that wrote the data. The written data is only visible to this thread. The difference between the two memories is that the *local* memory reads the data slower compared to the *register* memory.
Shared memory makes it possible for threads within an MP to communicate and share data with each other. The *shared* memory lasts for the duration of the block and the data is visible to all threads in this block. It can happen that two processors read and write information on the same part of the memory. This is important to keep track of, because the order of the processes influences the answer.
Global memory also makes it possible for threads to communicate, but this memory is visible to all threads within the application and lasts for the duration of the host allocation. *Global* memory is also called *device* memory. In a multiprocessor, each thread can read and write to any location in the memory.

Constant memory and *texture* memory are both read only and therefore beneficial for special types of applications. Using one of those types of memories can reduce the required memory bandwidth when threads read the same location. On a GPU, *shared* memory is the type of memory which is used the most.
An overview of the different types of memory on a GPU can be seen in Figure 5.1.

The cache is a part of the memory between the main memory and the vector registers. A vector register operates on a vector processor and contains vectors. Data transfer between cache and vector is fast, but data transfer between cache and main memory is slow.

The amount of time required to move n data items depends on the latency or start-up time α and the incremental time per data item moved: β. Therefore a simple formula for the time it takes to move n data items is: $\alpha + \beta n$.
When a computer has a relatively high latency, it is useful to combine communications.

5.1.3. Data Transfer

For data transfer between system memory and graphics card memory and vice versa, pinned memory or non-pinned memory can be used. Pinned memory provides improved transfer speeds, but is much more expensive to allocate and deallocate compared to non-pinned memory. This means that pinned memory is useful if a lot of memory needs to be transferred, because then it will provide a performance advantage.

5.1.4. Single and Double Precision Processing

Both single and double precision floating-point formats are used on GPU's. The single precision format occupies 4 bytes (32 bits) in computer memory and can be used for computational expensive parts of the calculations. The double precision format occupies 8 bytes (64 bits) in computer memory and is preferred to have a lower aggregate error but is also slower compared to the single precision format.

5.2. Message Passing Interface Versus Graphics Processing Units

MPI is coarsely parallel, which means that various calculations can be performed by different computers. When MPI is used, all threads are independent and therefore need to communicate by sending messages. MPI can be used in C and Fortran and contains a rich set of routines.
GPU is fine parallel, which means that just exactly the same orders can be performed on the nodes. Therefore a structured way of memory use is desirable. On a GPU, threads can communicate with each other if the right memory is used. Remember that this memory is limited, so problems that exceed the memory size cannot be solved by just a single GPU. GPU's are convenient to use if in one problem several same calculations have to be performed.

5.3. Vector Computers Versus Parallel Computers

In the past, Vector Computers were used to parallelize computations before the invention of GPU's. For a Vector Computer, it is important to perform the loops and memory in a structured way. Vector operations are performed fast, but different structures are performed slower. The GPU also prefers structured memory.
Vector Computers contain vector registers, which can be used to store intermediate and final results. The vector registers are very costly, so only a small number of reals can be stored. When a vector is too large to store in the vector registers, the Vector Computer evaluates the calculation in different parts. GPU's can perform various calculations in parallel and do not need to perform calculations in different parts.
Vector Computers are mainly focused on performing vector operations and perform slower when working with different structures. For GPU's this difference does not matter that much: most important is the fact that the GPU can only perform the exact same operations in parallel.

5.4. Common Pitfalls and Tips for Graphics Processing Units

GPU's are already used extensively in practice to perform matrix vector multiplications. The main focus of this thesis is the computation of the matrix vector product for the MLFMA. It is checked if, and how this can be performed efficiently by GPU's and what the bottlenecks are.

It is important to know what happens exactly when working with GPU's. It can happen that algorithms only become slower when running them on GPU's and sometimes it is needed to perform more work on a GPU to achieve a faster calculation compared to a CPU.

A GPU can perform multiple computations at the same time, but it is important to keep in mind that those computations must be simple. In case of the MLFMA, the algorithm has to be split into several smaller sub-algorithms which can be performed on a GPU.

One of the possible causes that lead to longer computation times on a GPU compared to a CPU is the fact that sending information from the CPU to the GPU, and the other way around, is time-consuming. When for a specific problem few computations are performed in parallel on a GPU, it can happen that the GPU does not perform faster compared to a CPU. This is because one iteration on the GPU is not fast enough compared to one iteration on the CPU, to catch up the time lost for sending information to the GPU. Therefore, it is important to understand exactly what happens and send all data at the same time and perform several calculations in parallel, so that the time to send information is negligible.
It is important to check always if a speed-up is obtained, because it could happen that sending the data is more time-consuming compared to the time saved by parallel computing.

Another important thing to check is if the algorithm contains recursions. Matrix vector products can be performed perfectly in parallel, but if there is a recursion in the algorithm, calculations cannot be performed independently of each other and can therefore not be performed in parallel. An example of a recursion which cannot be performed in parallel is the following:

do $1 = 1, 10$
  $y(i) = b(i) - a(i)y(i-1)$
enddo

The operators $+$, $-$ and $*$ all count as one flop, while the division operator is more expensive. Therefore it is important to avoid this operator as much as possible. It is possible to use the operator once on the CPU ($h = 1/b$) and afterwards multiply the values on the GPU by h, which gives the same result.

Finally, the most important thing is the amount of data transfer. When too much data is transferred, the method does not perform faster on a GPU. The implementation strategy with as little data transfer as possible is the best one.

5.5. GPU Implementation

There are different ways of writing code for the GPU. Specific GPU code can be written, standard GPU routines can be called from the GPU and standard GPU routines can be called from the CPU.

Writing GPU code directly has to be done in the *device code*. The *host code* is the part that is performed by the CPU and from here a call to the *device code* can be done. In the *device code* all steps that have to be performed by the GPU are implemented. The *device code* is written for one single thread, but all different threads walk through this routine. An example of writing GPU code directly is filling a matrix and specific vectors in the host code on the CPU and send this matrix and the vectors to the GPU. A call to the kernel has to be done from the CPU, where the number of threads per block and the number of blocks per grid have to be defined. On the GPU kernel or *device code*, multiple computations and iterations can be performed. Back in the *host code*, data has to be copied back from the GPU to the CPU and the solution can be printed.

This way of implementation only works for small problems, because of the fact that the computations are performed by different threads within one block (with a maximum of approximately 512 threads per block).

This means that they all can read and write information in the shared memory. This type of memory is fast and lets threads within a block communicate with each other very easily. When the problem size becomes larger than 512, multiple blocks are needed to perform the computation in parallel. This means that communication between the threads in different blocks is needed. This is difficult to implement with limited knowledge about GPU's.

To circumvent this, standard Basic Linear Algebra Subprograms (BLAS) routines in CUDA can be used. This means that for example copies, summations or dot products are performed in a standard efficient way, while the GPU divides the computations over the threads by itself. Recently, it became possible to call those cuBLAS routines from the device code (dynamic parallelism), while earlier this could only be done from the host code.

The easiest way of working with a GPU is by only making use of standard cuSPARSE and cuBLAS routines. Information has to be sent once to the GPU and all instructions for the GPU can be given by the CPU. For this research this approach is used. The matrices and vectors are sent to the GPU memory once. After the computations on the GPU, the solution vector is sent back to the CPU once. This method is shown schematically in Figure 5.2.

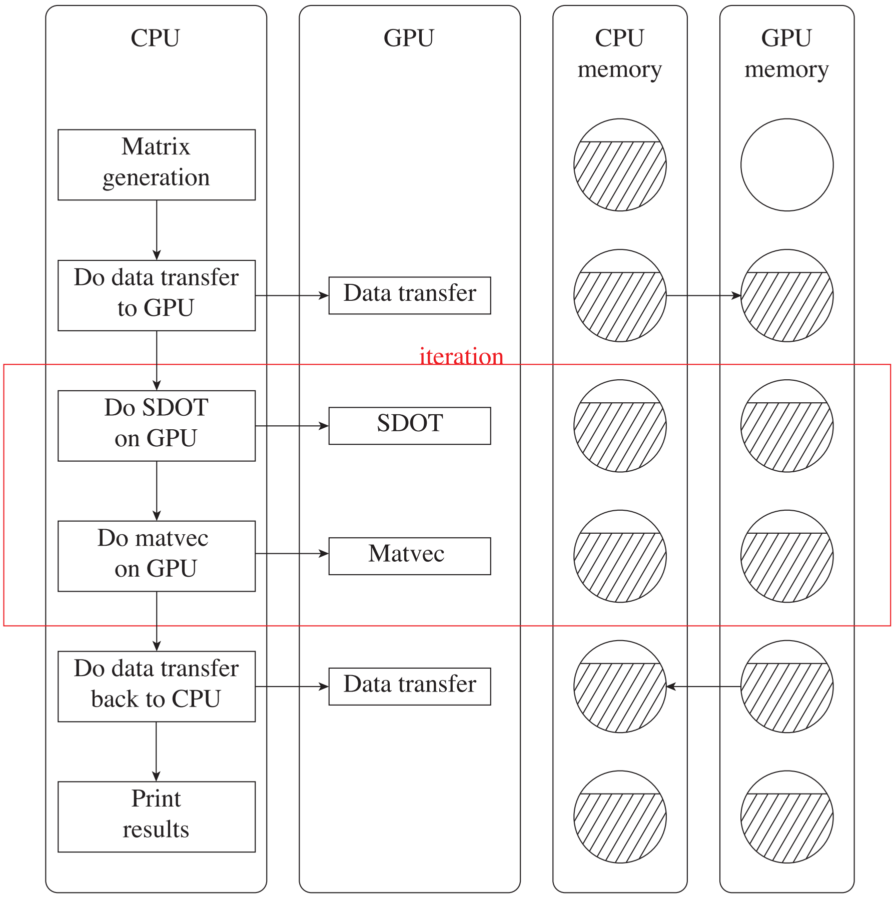

Figure 5.2: Overview of the calculations and communications on the CPU and GPU.

To make use of the sparse matrix vector multiplication routine on the GPU (part of cuSPARSE), the matrix has to be stored in a specific Sparse Matrix Storage Format. There are different Sparse Matrix Storage Formats. In this thesis the Compressed Row Storage (CRS) format is used, because of the fact that this format is used in the current MLFMA implementation for the CPU. In the CRS format, three vectors have to be stored: one vector containing the values of the nonzeros, one vector containing the column indices of the nonzeros and one vector containing the row offsets. Another format is Ellpack-Itpack [37]. For Ellpack-Itpack two matrices have to be set up: one matrix with the values of the nonzeros and one matrix with the number of the column indices. The number of columns in the matrices is equal to the number of entries per row. This is a good format to try in future research, because of the fact that it uses less memory in the MLFMA compared to the CRS format, under the assumption that the rows all have approximately the same length.

5.6. GPU and CPU Specifications

5.6.1. Kepler K40

For this thesis, a KEPLER with an NVIDIA TESLA GK110B Kepler K40 GPU Computing Card is used. Specifications of the Kepler K40 can be found in Table 5.1, where the Streaming Multiprocessor (SM) is the part on the GPU where processing elements are grouped together and the instructions are stored [34].

Released	November 2013
Processing elements	2,880
Streaming mult. processors	15
Processor clock (MHz)	745
Memory clock (GHz)	3.0
Memory bandwidth (GB/s)	288
Memory bus interface PCIe	3.0x16
Max. global memory (GB)	12,288
Shared memory (per SM)	48 kB
Registers (per block)	65,536
Computational speed single precision (GFLOP/sec)	4,290
Computational speed double precision (GFLOP/sec)	1,430

Table 5.1: Specifications of the Kepler K40 GPU.

If the GPU performs one operation per clock cycle, the computational speed is the number of cores multiplied by the processor clock: $2,880 \cdot 745 = 2145$ GFLOP/sec. The computational speed for single precision computations is exactly twice this number, which means that the GPU performs two operations per clock cycle for single precision computations.

Information about the compilation of a Fortran implementation of an algorithm on the GPU can be found in Appendix C.

5.6.2. Intel E5-2640

For this thesis, a single core of the Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz is used. Information about the compilation of a Fortran implementation of an algorithm on the CPU can be found in Appendix C.

6

The Application of the MLFMA on a GPU

The Multi Level Fast Multipole Algorithm consists of different parts that may or may not be suitable for parallelization on a GPU.The construction of the matrices V, A and T, the construction of the near field system matrix $\mathscr{L}'$ and the matrix vector multiplications in the MLFMA are expected to be suitable for parallelization on a GPU. In this chapter, a first analysis of those parts of the MLFMA on the GPU is given. Afterwards, results from literature are given and the decision for the focus of this thesis is explained.

6.1. First Analysis

The overview of the building blocks in Section 4.4 arouses various ideas for parallelizing the MLFMA on GPU's. The ideas are stated below.

- Constructing near field system matrix $\mathscr{L}'$ on GPU.
- Constructing matrices V, T and A on GPU.
- Global interpolation method instead of local interpolation method.
- Parallelizing MLFMA over both groups and wave directions.

The construction of the near field system matrix $\mathscr{L}'$ can be performed faster on GPU's by letting each thread of the GPU compute one matrix entry. The computations of the coefficients of the matrices V, T and A can probably be computed faster with the use of GPU's, because of the fact that the computations of the coefficients can be performed completely independently. Because of the use of e-powers, Hankel functions and Legendre polynomials, which are time consuming calculations, the fill can be performed faster with the use of GPU's by letting each thread of the GPU compute one matrix entry.

For the interpolation step, NLR makes use of Lagrange interpolation. This is a local interpolation method because only information of the neighbours is needed. It can be checked if a global interpolation method operates faster on GPU's [2].

The matrix vector multiplications of the MLFMA can be parallelized over the groups or over the wave directions. The challenge is the fact that one way is better suited on fine levels and the other is better suited on coarse levels. NLR makes use of a method which parallelizes over the groups at fine levels, while it parallelizes over the wave directions at coarser levels. A consequence of this is the fact that a lot of data transfer is needed during the switch between the two different ways of parallelization. Parallelization the whole algorithm over both the wave directions and the groups is an idea which came up immediately. This possibly results in a fast way of performing the algorithm, but there need to be enough threads and memory to compute everything in parallel at the same time.

6.2. Survey Existing Literature MLFMA and GPU's

The first ideas for parallelizing the MLFMA on GPU's are explained in the previous section. It is important to make use of publications to see which methods are performed in practice and how much speed-up they obtain.

There are various publications about the application of the Method of Moments on GPU's, about the Fast Multipole Method on GPU's for modelling particles and about the MLFMA with non-oscillatory kernels, but very few about the Multi Level Fast Multipole Algorithm with an oscillatory kernel e^{ikr}/r.
Multi Level Fast Multipole Algorithms for non-oscillatory kernels have different computational complexity compared to Multi Level Fast Multipole Algorithms for oscillatory kernels. The most important difference is that for non-oscillatory kernels the computational load decreases fast at coarser levels, whereas for oscillatory kernels the computational load remains the same (less groups but more wave directions). As the work is concentrated on the finest level, the MLFMA for non-oscillatory kernels is computationally comparable with the FMM for oscillatory kernels. As the focus of this thesis is on the Multi Level Fast Multipole Algorithm with an oscillatory kernel, literature for non-oscillatory kernels is skipped.

The publications about the MoM and the FMM can still be used: they can contain useful information about the parallelization of the composition of the matrices $\mathscr{L}$, V, T and A.

The fact that there are few publications about the specific MLFMA with an oscillatory kernel could mean that parallelizing the MLFMA on a GPU does not result in a large speed-up.

It is difficult to give the exact speed-up achieved in publications, because different GPU's cannot be compared because of their difference in characteristics. Nonetheless, an idea of the order of the acceleration is given for implementation strategies from publications.

In Table 6.1, different ways to accelerate the FMM or the MLFMA on GPU's are given. It can also be seen in which publications the different ways of acceleration are discussed. In this way it is clear if the manner is frequently used or if it is a new or not very useful manner. Afterwards, it is discussed if the different manners are useful for this research and for which components they can be used.

How?	Reference
Each thread computes one entry of the near-field interaction matrix	[13],[15],[18],[32],[33]
BiCGSTAB as solver	[13],[14]
Each thread is related to one basis function	[13]
Distribute data among nodes using a group-based partitioning scheme	[13],[15],[18]
New parallel algorithm	[16]
Use of pinned memory	[18]
Different or 'own' GPU clusters	[20]
Application of HPC clusters	[25]
'Compute on-the-fly' strategy	[26]

Table 6.1: Information about MoM, FMM or MLFMA acceleration on GPU's from publications.

6.2.1. Construction $\mathscr{L}'$

Following the reasoning presented by J. Guan, S. Yan and J.M. Jin in [13], for the near interaction matrix several threads are busy to carry out the assembly of a part of the matrix. The matrix fill can be performed in parallel by letting one thread compute one entry of the matrix.

In the work of Q. Nguyen, V. Dang, O. Kilie and E. El-Araby in [15], the same happens, but in this case the threads are grouped into blocks that are responsible for computing the elements of one row of $\mathscr{L}'$.

In [18], J. Guan, S. Yan and J.M. Jin obtain a speed-up of 124.5 when constructing the near field system matrix $\mathscr{L}'$ on a GPU instead of a CPU. Each thread computes a non-zero element of the near field system matrix and stores the results in the global memory.

6.2.2. Construction V, T, A

Following the reasoning presented by Q. Nguyen, V. Dang, O. Kilie and E. El-Araby in [15], the construction of the transfer matrix T works the same as the construction of $\mathscr{L}'$: each thread computes one entry of the matrix and threads are grouped into blocks that are responsible for computing the elements of one row of T.

The outgoing and incoming waves V and $\overline{V}^T$ are complex conjugates of each other, so just V needs to be constructed. The group-based partitioning scheme is used for the construction, which has to be done K times.

6.2.3. Solution of the Linear System

Following the reasoning presented in [13] and [14], it is important to have a preconditioner that can be parallelized efficiently on a GPU. The iterative method BiCGSTAB can be used together with preconditioners ILU0, Approximate Inverse (AI) or Jacobi. From the examples in [13] it can be seen that the ILU0 is not suitable for working with GPU's because of the forward and backward substitutions. The AI preconditioner is the cheapest preconditioner in the solution phase and the Jacobi is the cheapest preconditioner in the construction phase. It depends on the size of the example in the paper if the AI or the Jacobi preconditioner achieves more speed-up: larger problems achieve more speed-up using the AI preconditioner.

6.2.4. Matrix Vector Multiplication

According to the authors of [13], fast evaluation of the scattered fields can be achieved by letting each thread be related to one basis function. At a specific wave direction, each GPU thread computes the corresponding scattered field in parallel. One CPU thread superposes all the calculated scattered fields in series to avoid writing conflicts on the GPU. Unfortunately, this method does not work for the MLFMA because the MLFMA operates with different group sizes, which means that the wave directions are different on each level.

Following the reasoning presented in [13],[15] and [18], the distribution of data among nodes using a group-based partitioning scheme, operates well on GPU's. Unfortunately, the group sizes change in the MLFMA, so this way of parallel computing does not work for the MLFMA.

The parallelization of the far field interactions can be implemented with the use of pinned or global memory, stated by the authors in [18]. The global memory strategy stores the outgoing and incoming waves at each level on a single GPU. Therefore the computational efficiency is very high, because of the fact that no data transfer between the host and device is needed. Unfortunately, the global memory is not very large, so this limits the size of the problem that can be solved. The pinned memory strategy stores the results to the pinned memory on multiple GPU's after computing the outgoing and incoming waves. Larger problems can be solved because of the fact that the pinned memory is much larger compared to the global memory, but data communications between host and device become unavoidable. The global memory strategy obtains more speed-up, but cannot be used for large problems. It dependents on the size of the problem which memory strategy is most suitable.

The parallelization strategy of [18] divides different wave directions over different threads and lets one or several block(s) compute one group. Assuming all wave directions can be calculated by different threads on one GPU at the coarsest level, the parallelization at finer levels also fit on a single GPU. The size of a thread block is determined by the number of wave directions on the finest level. A number of blocks together represent a parent cube. The number of blocks representing a parent cube grows for coarser levels. This method only reaches a speed-up of 2.9 on a GPU compared to the computations on a CPU.

6.2.5. General

According to the authors of [16], a new parallelizable algorithm generates the FMM data structure in $O(N)$ time with complexity $O(N)$. Unfortunately, the algorithm is for the Fast Multipole Method for particles, which operates differently compared to the Fast Multipole Method for oscillatory kernels.

Following the reasoning presented in [20], GPU clusters are important in the future for solving problems even faster. Northrop Grumman invested in the creation of its own GPU cluster. This appears to be an advertising story and is not useful for this thesis.

In [25], a High Performance Computing (HPC) cluster is used to solve large scale scattering problems. The HPC clusters can overcome the memory limitation of GPU's and take advantage of the conventional CPU based cluster to achieve more acceleration.

For the MLFMA, more data communications between CPU and GPU are required when the problem size increases, which results in a reduction of the computational efficiency. Therefore, [26] uses a 'compute on-the-fly' strategy, where the whole system matrix is not computed and stored before the iterative solution. The entries are computed whenever they are needed in the matrix vector multiplications. The number of recalculations depend on the number of iterations. In this case, the second-kind Fredholm integral equation is used because of its optimal iterative convergence.

6.3. Possible Methods and Bottlenecks

After the first analysis and reading literature about the application of the FMM and the MLFMA on GPU's, some options are available for a possible speed-up of the MLFMA on a GPU. The options from literature and from the analysis are explained below.

The previous section was divided in three different parts of the MLFMA that can possibly be accelerated on a GPU: the construction of $\mathscr{L}'$, the construction of V, T and I and the matrix vector multiplications. In some publications, the construction of matrices is parallelized by letting each thread compute one matrix entry. Sometimes these threads are grouped into blocks that are responsible for computing the elements of one row of $\mathscr{L}'$.

The construction of the matrices V, T and I can be performed in the same way as the construction of matrix $\mathscr{L}'$. In various publications, matrices V and T are filled in parallel for the groups for just a single wave direction. This means that the matrix construction has to be done K times. When the parallelization is performed over the wave directions and the groups at the same time, matrices V and T can be constructed directly. This was not discussed in the publications, but is worth trying.

Parallelizing over the wave directions and using the pinned memory can speed-up the algorithm. For this strategy, there need to be enough threads to be able to parallelize over the wave directions on the coarsest level.

Another strategy from literature is to recalculate matrices when they are needed (compute on-the-fly strategy). This results in less data transfer, but is also inefficient because of the large number of recalculations for large problems.

Choosing the suitable type of memory for an algorithm is difficult when working with GPU's: memory is limited and there are several different types of memory. Shared memory is the type of memory that is used the most on GPU's.

There is nothing discussed about local or global interpolation methods in publications used for this thesis. It is still worth comparing a global interpolation method to the local interpolation method (Lagrange interpolation).

Unfortunately, it is not possible to try all different approaches, as this is outside the scope of this thesis. NLR knows from practice that the matrix vector multiplications of the MLFMA are the most time consuming part of the algorithm. Therefore, the focus in this thesis is on the acceleration of matrix vector multiplications of the MLFMA on the GPU.

7

Model Problem for the MLFMA

To get used to utilizing a GPU, computations for a model problem are performed in parallel on a GPU. As a model problem the discretized Poisson equation is used, which is solved iteratively using the Conjugate Gradient method. In order to introduce the design considerations for GPU implementations, this chapter considers an easy model problem. The model problem shall have similar computational characteristics as the MLFMA algorithm. A Poisson solver is chosen since it is a sparse, linear problem which can be solved iteratively, just like the MLFMA.

7.1. Poisson Equation

The two-dimensional Poisson equation is the following:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)u(x,y) = f(x,y) \ \in \Omega, \tag{7.1}$$

where Ω is a two-dimensional domain. In this example homogeneous Dirichlet boundary conditions are imposed and u meets these conditions.

In this example, the exact function u is equal to

$$u(x,y) = sin^2(\pi x) sin^2(\pi y), \tag{7.2}$$

which means that the Laplacian of this solution is given by:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)u = 2\pi^2 cos(2\pi x) sin^2(\pi y) + 2\pi^2 sin^2(\pi x) cos(2\pi y). \tag{7.3}$$

The Poisson equation is discretized, based on a standard second order finite difference approximation of the Laplacian on a five point stencil.

7.2. Conjugate Gradient Solver

Iteratively solving the discretized Poisson equation can be done using the Conjugate Gradient method, because the resulting matrix is symmetric positive definite. For this method, in each iteration the new values of $u_j = u(x_j)$, $\boldsymbol{p}$, and $\boldsymbol{r}$ are computed, which are the solution vector, the search direction vector and the residual vector, respectively. u, p and r are elements in the set R. At the end of each iteration, the solution vector is compared to the exact solution vector to check if the method is converged.

The algorithm of the Conjugate Gradient method starts with the computation of the following initial values for the search direction $\boldsymbol{p}$ and the residual $\boldsymbol{r}$: $\boldsymbol{p}_{(0)} = \boldsymbol{r}_{(0)} = \boldsymbol{b} - A\boldsymbol{u}_{(0)}$, where $\boldsymbol{b}$ is the right-hand side in $A\boldsymbol{x} = \boldsymbol{b}$ and $\boldsymbol{u}$ is the final solution. In each iteration i, the following is computed (until the solution vector is sufficiently close to the exact solution):

$$\begin{aligned}
\alpha_{(i)} &= \frac{\boldsymbol{r}_{(i)}^T \boldsymbol{r}_{(i)}}{\boldsymbol{p}_{(i)}^T A \boldsymbol{p}_{(i)}} \\
\boldsymbol{u}_{(i+1)} &= \boldsymbol{u}_{(i)} + \alpha_{(i)} \boldsymbol{p}_{(i)} \\
\boldsymbol{r}_{(i+1)} &= \boldsymbol{r}_{(i)} - \alpha_{(i)} A \boldsymbol{p}_{(i)} \\
\beta_{(i+1)} &= \frac{\boldsymbol{r}_{(i+1)}^T \boldsymbol{r}_{(i+1)}}{\boldsymbol{r}_{(i)}^T \boldsymbol{r}_{(i)}} \\
\boldsymbol{p}_{(i+1)} &= \boldsymbol{r}_{(i+1)} + \beta_{(i+1)} \boldsymbol{p}_{(i)}.
\end{aligned} \tag{7.4}$$

7.3. Implementation

In Section 5.5 it is explained that writing GPU code for large problems directly is difficult when limited knowledge about GPU's is available. For small problems, where all computations can be performed by threads within one block, it is less difficult to write the host and device code. It is also explained that cuBLAS and cuSPARSE routines are efficient and work faster than writing GPU code directly. To support this remark, in this chapter the device code implementation and the cuBLAS and cuSPARSE implementation are compared for small problem sizes of the model problem.

7.3.1. Device Code Implementation

In the device code implementation, no standard GPU routines are used. The device code is written directly. In the device code implementation, the right-hand side, the exact solution vector, the dense matrix and the initial residual, solution and search direction vectors are computed on the host code (or CPU). After this, the block dimension and grid dimension are set (in this case they are both equal to one) and the initial residual, search direction and solution vectors are sent to the GPU. Also the dense matrix is sent to the GPU. A call to the Poisson kernel, where the Conjugate Gradient method is used to solve the Poisson equation iteratively, is done by the CPU. The device code is written for the computations of one thread, but multiple threads will perform those calculations. One thread computes the multiplication of one row of the matrix by the search direction vector and computes one element in dot products, scalar-vector multiplications, vector-adds and copies. Before threads share information with each other, a *syncthreads* call has to be done, so that the threads only share information if all threads have finished the computation. In each iteration the next solution vector, residual and search direction are computed. During the iterations, data is saved in the memory of the GPU. As soon as the method is converged, the final solution vector is copied back to the CPU and can be printed on the CPU. The implementation in Fortran can be found in Appendix D.

7.3.2. cuBLAS and cuSPARSE Implementation

In the cuBLAS and cuSPARSE implementation device code is not written directly, but standard efficient cuBLAS and cuSPARSE routines are used. In the cuBLAS and cuSPARSE implementation, the right-hand side, the exact solution vector, the sparse matrix and the initial residual, solution and search direction vectors are computed on the CPU. After this, the initial residual, search direction and solution vectors are sent to the GPU. Also the vectors for the sparse CRS format are sent to the GPU. The Conjugate Gradient method is used to solve the Poisson equation iteratively. For the Conjugate Gradient method, cuSPARSE is used for the computation of the matrix vector product and cuBLAS is used for dot products, scalar-vector multiplications, vector-adds and copies in the algorithm. The cuSPARSE and cuBLAS routines are called from the CPU and performed by the GPU. The GPU divides the data over the threads and blocks in an efficient way. In each iteration the next solution vector, residual and search direction are computed. During the iterations, data is saved in the memory of the GPU. As soon as the method is converged, the final solution vector is copied back to the CPU and can be printed on the CPU.

The implementation in Fortran can be found in Appendix D. It can be seen that the division of two scalars is performed by the CPU. This is possible, because of the fact that sending just a single scalar back to the CPU is negligibly compared to the computing times.

For the CPU implementation the same computations are performed, but information does not have to be sent and instead of cuBLAS and cuSPARSE routines, BLAS and SPARSE routines from the MKL library of Intel are used.

7.4. Device Code and cuBLAS and cuSPARSE Results

The device code implementation and the cuBLAS and cuSPARSE implementation for the GPU are compared for small problem sizes of the model problem. In Section 5.6 information about the hardware of the GPU and CPU is given.

Communication times (time needed to send date to the GPU) and calculation times (time needed to perform calculations) are important for a GPU algorithm, together this is called the computation time. For a CPU the computation time is equal to the calculation time, because of the fact that data does not have to be sent to the GPU. For several small problem sizes for the model problem, 500 iterations are performed for the device code implementation and the cuBLAS and cuSPARSE implementation. The results are given in Table 7.1.

Unknowns	Time device code (sec)	Time cuBLAS and cuSPARSE (sec)
64	2.370	0.309
324	2.460	0.327
784	2.619	0.332
1,444	2.991	0.351
2,304	3.303	0.391

Table 7.1: Computation times device code implementation and cuBLAS and cuSPARSE implementation 2D Poisson model problem.

The cuBLAS and cuSPARSE implementation, that makes use of the standard cuBLAS and cuSPARSE routines, is approximately 8 times faster compared to the device code implementation that is written directly. Note that in the device code implementation the matrix vector multiplication is not sparse. This means that $\frac{N^2}{5N}$ more calculations have to be performed for the device code implementation, compared to the cuBLAS and cuSPARSE implementation. This is one of the reasons why the device code implementation is 8 times slower compared to the cuBLAS and cuSPARSE implementation. The data in the dense matrix in the device code implementation is on the other hand easier to read for the GPU, compared to the data of the sparse matrix in the cuBLAS and cuSPARSE implementation. Because of these reasons, it is difficult to compare the two implementations fairly.

7.5. Double Precision Results

The device code implementation can only be used for small problem sizes of the model problem and the cuBLAS and cuSPARSE implementation is significantly faster for small test problems, compared to the device code implementation. Therefore, the computations in this chapter are performed for the cuBLAS and cuSPARSE implementation.

7.5.1. cuBLAS and cuSPARSE Results

For different problem sizes of the model problem, the calculation times for one iteration on the CPU and one iteration on the GPU are computed. In this way the maximum speed-up can be computed. This speed-up can be obtained if enough iterations are performed on the GPU, so that the time needed to send information back and forth is negligibly small. The results can be seen in Table 7.2.

Unknowns	CPU (sec)	Send information to GPU	GPU (sec)	Send information to CPU	Max acceleration	Number of iterations
248,004	$2.800 \cdot 10^{-3}$	0.287	$4.989 \cdot 10^{-4}$	$5.07 \cdot 10^{-4}$	5.5	20,000
996,004	$1.546 \cdot 10^{-2}$	0.302	$1.626 \cdot 10^{-3}$	$1.62 \cdot 10^{-3}$	9.5	18,000
2,244,004	$3.931 \cdot 10^{-2}$	0.320	$3.506 \cdot 10^{-3}$	$3.51 \cdot 10^{-3}$	11.2	17,000
3,992,004	$7.147 \cdot 10^{-2}$	0.369	$6.193 \cdot 10^{-3}$	$6.16 \cdot 10^{-3}$	11.5	16,000
6,240,004	0.113	0.418	$9.809 \cdot 10^{-3}$	$9.55 \cdot 10^{-3}$	11.5	15,000
8,988,004	0.162	0.449	$1.388 \cdot 10^{-2}$	$1.37 \cdot 10^{-2}$	11.7	13,000
12,236,004	0.220	0.463	$1.875 \cdot 10^{-2}$	$1.87 \cdot 10^{-2}$	11.7	12,000
15,984,004	0.287	0.518	$2.422 \cdot 10^{-2}$	$2.44 \cdot 10^{-2}$	11.8	11,000
20,232,004	0.364	0.573	$3.071 \cdot 10^{-2}$	$3.08 \cdot 10^{-2}$	11.9	10,000
24,980,004	0.450	0.643	$3.795 \cdot 10^{-2}$	$3.81 \cdot 10^{-2}$	11.9	20,000
30,228,004	0.544	0.872	$4.590 \cdot 10^{-2}$	$4.61 \cdot 10^{-2}$	11.9	30,000

Table 7.2: Double precision results 2D Poisson model problem on CPU and GPU.

Small problems need to perform more iterations to obtain the maximum speed-up compared to larger problems. This is because of the fact that for small problems, less calculations can be performed in parallel, so more iterations have to be done to have a negligibly small communication time. In Table 7.2 it can be seen that the number of iterations needed to obtain the maximum acceleration does not decrease for the two largest problem sizes. The number of iterations even grows, which is possibly the case because of the fact that the bandwidth of the GPU becomes full.

The plot of the accelerations on the GPU for different problem sizes can be seen in Figure 7.1.

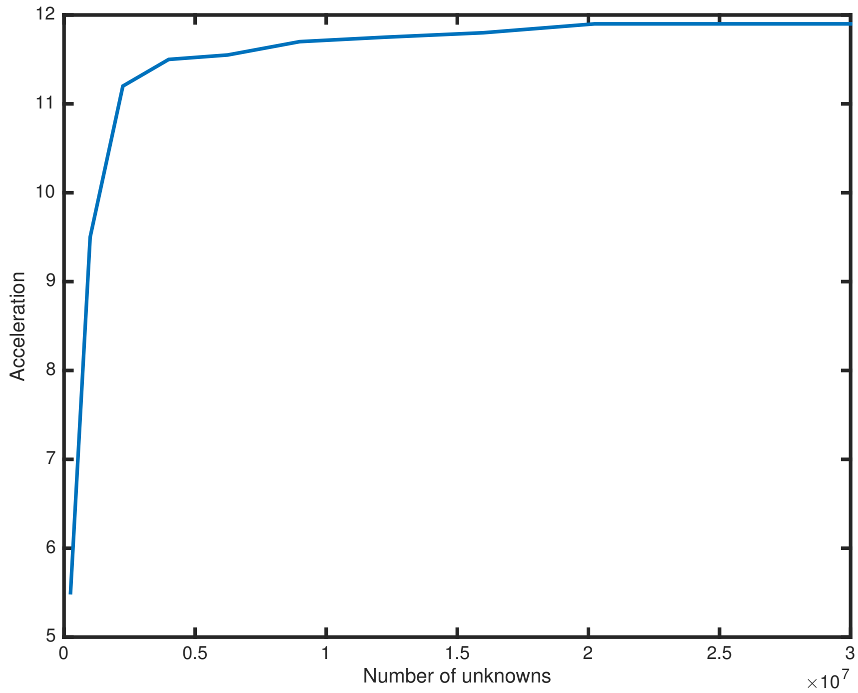

Figure 7.1: GPU acceleration compared to the CPU plotted against number of unknowns for the 2D Poisson model problem in double precision.

The GPU algorithm is up to approximately 11.9 times faster compared to the CPU algorithm (11.9 appears to be the asymptotic value). This maximum is reached by problems with more than 20,232,004 unknowns. If the bandwidth of the GPU is fully used, larger problems cannot obtain more speed-up. In the next section the computational efficiency and memory efficiency are computed to see if the bandwidth is really full and to explain this maximum speed-up.

To show for which problem size the algorithm is faster on the GPU compared to the CPU, the data in Table 7.3 is plotted in Figure 7.2 by making a zoom in the plot with the computation time against the number of unknowns for larger problems.

Unknowns	Time CPU (sec)	Time GPU (sec)
2,304	$1.14 \cdot 10^{-2}$	0.326
9,604	$4.73 \cdot 10^{-2}$	0.338
21,904	0.117	0.355
39,204	0.209	0.369
61,504	0.328	0.381
88,804	0.470	0.404

Table 7.3: Double precision results 2D Poisson model problem on CPU and GPU for small problem sizes.

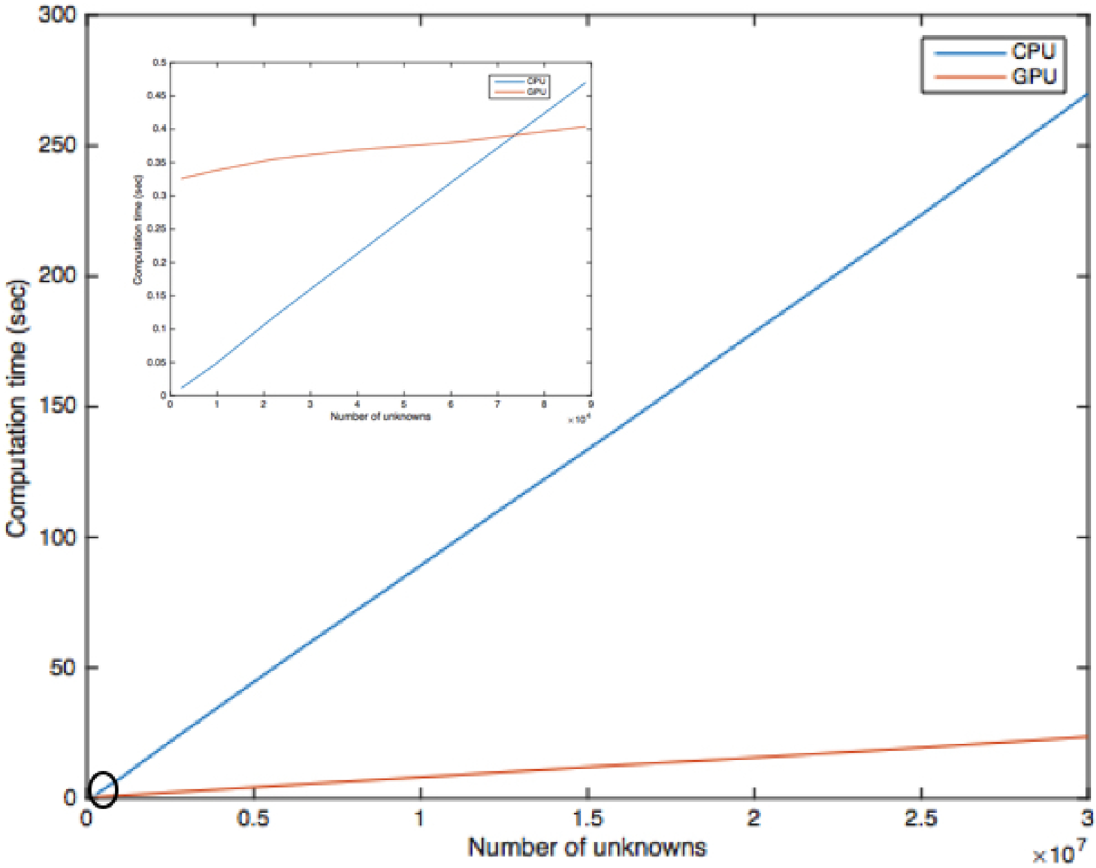

Figure 7.2: CPU and GPU computation times plotted against number of unknowns for 2D Poisson model problem in double precision.

The algorithm is faster on the GPU for problem sizes with approximately 75,000 unknowns. For small problems, not many computations can be performed in parallel, so the acceleration on the GPU is not enough to compensate the lost time for sending information to the GPU. For larger problems more computations can be performed in parallel, so the information that is sent is negligibly small.

7.5.2. Computational Efficiency and Memory Efficiency

It can be checked if those cuSPARSE and cuBLAS routines are indeed working efficiently by calculating the computational speed (in GFLOP/sec) and the memory bandwidth (in GB/sec) for the algorithm and compare this with the specifications of the GPU and CPU. After that, it can be concluded whether 10% or 95% of the hardware capabilities is used and whether more time should be spend on self-tuning. Self-tuning is dividing the computations over the threads by yourself, so that the algorithm operates efficiently.

In Table 7.4, for each operation it is computed how many bytes are needed for one iteration. The computations are for a problem size of 20,232,004 because of the fact that at this point the maximum speed-up is achieved. The number of rows (and columns) are $c = 20,232,004$ and the number of nonzeros is $nnz = 101,142,028$. The computations are performed in double precision and integers in 4 bytes.

Step	**Calculation**	**Bytes**
Matrix vector product	$nnz \cdot 4 + nnz \cdot 8 + (c+1) \cdot 4 + 2 \cdot c \cdot 8$	$1.618 \cdot 10^9$
Dot product	$2 \cdot c \cdot 8$	$3.237 \cdot 10^8$
Dot product	$2 \cdot c \cdot 8$	$3.237 \cdot 10^8$
Daxpy	$3 \cdot c \cdot 8$	$4.856 \cdot 10^8$
Dscal	$2 \cdot c \cdot 8$	$3.237 \cdot 10^8$
Daxpy	$3 \cdot c \cdot 8$	$4.856 \cdot 10^8$
Dot product	$2 \cdot c \cdot 8$	$3.237 \cdot 10^8$
Dcopy	$2 \cdot c \cdot 8$	$3.237 \cdot 10^8$
Dcopy	$2 \cdot c \cdot 8$	$3.237 \cdot 10^8$
Daxpy	$3 \cdot c \cdot 8$	$4.856 \cdot 10^8$

Table 7.4: Memory bandwidth calculations 2D Poisson model problem in double precision for a problem size of 20,232,004.

This means that in total $5.02 \cdot 10^9$ bytes = 5.02 GB is used in one iteration on the GPU. In Table 7.2 it can be seen that one iteration on the GPU for 20,232,004 unknowns takes $3.07 \cdot 10^{-2}$ seconds. It can be concluded that the memory bandwidth for this problem is 163 GB/sec.

For the computational speed the same table is created in Table 7.5. Again, the computations are for a problem size of 20,232,004.

Step	**Calculation**	**FLOPs**
Matrix vector product	$nnz + nnz - c$	$1.821 \cdot 10^8$
Dot product	$c + c - 1$	$4.046 \cdot 10^7$
Dot product	$c + c - 1$	$4.046 \cdot 10^7$
Daxpy	$c + c$	$4.046 \cdot 10^7$
Dscal	c	$2.023 \cdot 10^7$
Daxpy	$c + c$	$4.046 \cdot 10^7$
Dot product	$c + c - 1$	$4.046 \cdot 10^7$
Daxpy	$c + c$	$4.046 \cdot 10^7$

Table 7.5: Computational speed calculations 2D Poisson model problem in double precision for a problem size of 20,232,004.

This means that in total $4.451 \cdot 10^8$ FLOPs = 0.445 GFLOPs are performed in $3.07 \cdot 10^{-2}$ seconds, so the computational speed is 14.5 GFLOP/sec. Note that communication time and calculation time cannot be separated, so the computed speeds are lower bounds.

The Kepler K40 GK110B has a maximum memory bandwidth of 288 GB/sec and a maximum computational speed of 1430 GFLOP/sec for double precision computations. This means that the memory efficiency is 60% and the computational efficiency is less than 1%.

In most problems the memory bandwidth or the computational speed is 100%. If the memory bandwidth is 100%, information cannot be made faster available for GPU computations, which means that it is not possible to do more computations per second. If the computational speed is 100%, the machine operates as fast as it can, so it does not make sense to have information faster available for computations. This means that the memory bandwidth is less than 100%.

To check if the pgfortran compiler can indeed achieve the maximum memory bandwidth of 288 GB/sec, a very simple axpy (one addition and one multiplication) algorithm is written. For this algorithm, 3 vectors of the size of the problem have to be sent and in the calculations 2 times the size of the problem is the number of used FLOPs. This very small example gives results between 207 GB/sec and 215 GB/sec for maximum bandwidth, which is approximately 75% of the maximum memory bandwidth. The reason that this is not 100% is the fact that communication time and calculation time cannot be separated. Profiling can be used to see when the GPU has to wait for information, but the Portland compiler only allows profiling on the CPU. It can be concluded that the maximum bandwidth is reached when the computed bandwidth is approximately 75% of the total memory bandwidth.

The bandwidth of the GPU is 60% of the total memory bandwidth, while the bandwidth is full if approximately 75% of the total memory bandwidth is used. The standard cuSPARSE and cuBLAS routines potentially transfer data from global memory to shared memory and the other way around to make the computations as efficient as possible. This data transfer is not included in the calculations in this section, which means that in the computation more data is transferred. Therefore, it can be concluded that the memory bandwidth is approximately 100%. Because of the fact that the bandwidth is fully used, the computational efficiency is less than 1%, since computations cannot be started until the information has arrived. Also for larger problem sizes, computations cannot be performed faster and therefore no more speed-up is obtained. If multiple GPU's are used, the memory bandwidth will become higher and so the speed-up can potentially increase.

For the CPU (Intel E5-2640 v3) the maximum memory bandwidth is 59 GB/sec and the maximum computational speed is 10.4 GFLOP/sec. In the CPU algorithm, for the sparse matrix vector multiplication, an extra vector of length c has to be stored. This means that the total GB needed is 5.18. The algorithm on the CPU has a memory bandwidth of $\frac{5.18}{0.361} = 14.35$ GB/sec. This is approximately 24% of the maximum memory bandwidth of the CPU. The computational speed of the algorithm on the CPU is $\frac{0.445}{0.361} = 1.23$ GFLOP/sec. This is approximately 12% of the maximum computational speed of the CPU, so the computational speed is not maximal because of the fact that the memory bandwidth is getting fuller.

The computations in this section are performed in single precision in Section 7.6. For GPU computations in single precision, it is expected that data is available twice as fast, so it is expected that the elapsed time is halved. For the GPU computations in double precision the computational speed is less than 1%. Therefore, the speed-up of the computations in single precision are expected to be negligible. For CPU computations in single precision, it is expected that computations are twice as fast: because of the fact that the bandwidth is not full, calculations can be performed twice as fast in single precision. It is expected that GPU computations are twice as fast in single precision, compared to double precision computations. CPU computations are also expected to be two times faster in single precision, compared with double precision computations. This means that the speed-up of the GPU, compared to the CPU, is approximately the same for single precision computations, compared to double precision computations. The results for single precision computations can be found in Section 7.6.

7.5.3. Computation Times of Specific Parts

To know which parts of the model problem on the GPU and CPU are most time consuming, the calculation and communication times of the matrix fill, sending information to the GPU, the iterations and sending information back to the CPU are computed separately.

Tables 7.6 and 7.7 show the calculation and communication times in different parts of the algorithm for the CPU and the GPU, respectively.

Unknowns	Construct matrix	Iteration
248,004	$1.17 \cdot 10^{-2}$	$2.77 \cdot 10^{-3}$
996,004	$4.43 \cdot 10^{-2}$	$1.54 \cdot 10^{-2}$
2,244,004	$9.87 \cdot 10^{-2}$	$3.89 \cdot 10^{-2}$
3,992,004	0.175	$7.10 \cdot 10^{-2}$
6,240,004	0.267	0.112
8,988,004	0.376	0.160
12,236,004	0.509	0.219
15,984,004	0.664	0.285
20,232,004	0.842	0.361
24,980,004	1.031	0.447
30,228,004	1.237	0.540

Table 7.6: Double precision calculation and communication times in different parts of the 2D Poisson model problem on the CPU.

Unknowns	Construct matrix	Send information to GPU	Iteration	Send information back to CPU
248,004	$1.02 \cdot 10^{-2}$	0.287	$4.96 \cdot 10^{-4}$	$5.07 \cdot 10^{-4}$
996,004	$3.86 \cdot 10^{-2}$	0.302	$1.62 \cdot 10^{-3}$	$1.62 \cdot 10^{-3}$
2,244,004	$8.50 \cdot 10^{-2}$	0.320	$3.51 \cdot 10^{-3}$	$3.51 \cdot 10^{-3}$
3,992,004	0.154	0.369	$6.19 \cdot 10^{-3}$	$6.16 \cdot 10^{-3}$
6,240,004	0.242	0.418	$9.81 \cdot 10^{-3}$	$9.55 \cdot 10^{-3}$
8,988,004	0.333	0.449	$1.39 \cdot 10^{-2}$	$1.37 \cdot 10^{-2}$
12,236,004	0.454	0.463	$1.88 \cdot 10^{-2}$	$1.87 \cdot 10^{-2}$
15,984,004	0.571	0.518	$2.42 \cdot 10^{-2}$	$2.44 \cdot 10^{-2}$
20,232,004	0.748	0.573	$3.07 \cdot 10^{-2}$	$3.08 \cdot 10^{-2}$
24,980,004	0.931	0.643	$3.79 \cdot 10^{-2}$	$3.81 \cdot 10^{-2}$
30,228,004	1.130	0.872	$4.62 \cdot 10^{-2}$	$4.61 \cdot 10^{-2}$

Table 7.7: Double precision calculation and communication times in different parts of the 2D Poisson model problem on the GPU.

Sending information to the GPU takes much more time compared to sending information back to the CPU. This is because information about all matrices have to be sent to the GPU, while just the solution vector has to be sent back to the CPU. It can be seen that one iteration on the GPU takes approximately the same time as sending the solution vector back to the CPU.

A histogram makes it easier to compare the calculation and communication times for the different parts of the model problem. The time for constructing the matrix, sending information to the GPU (in case of the GPU), one iteration and sending information back to the CPU (in case of the GPU) are plotted in different stacks in Figure 7.3.

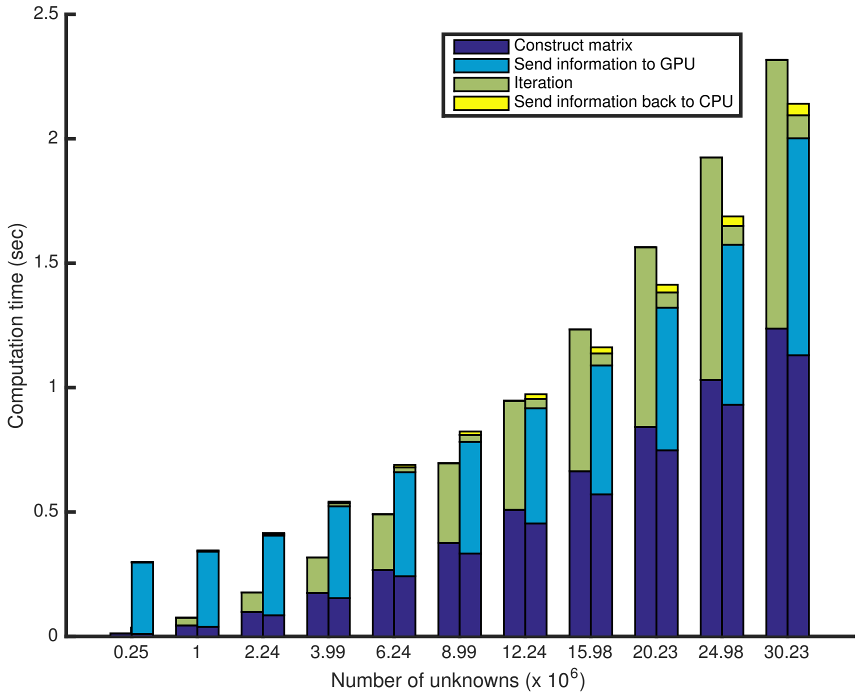

Figure 7.3: Double precision calculation and communication times in different parts plotted against number of unknowns for the 2D Poisson model problem for one iteration on the CPU and GPU.

It would be expected that constructing the matrix would take the same time in the CPU algorithm and the GPU algorithm, because both are computed by a CPU. The construction of the matrix takes less time on the GPU. The Portland compiler appears to perform computations on the CPU slightly faster compared to the Intel compiler.

It can be seen that sending information back and forth between the CPU and the GPU takes quite a lot of time: much more than one iteration on the GPU. Therefore a general GPU algorithm becomes faster after some more iterations. If two iterations are performed, the GPU is already faster than the CPU for large problem sizes.

7.6. Single Precision Results

After performing double precision computations for the model problem, curiosity for computations in single precision grows. Poisson computations are less accurate in single precision compared to double precision, but computations on the CPU and the GPU are faster compared to double precision computations. Possibly a larger speed-up of the GPU compared to the CPU can be obtained.

7.6.1. cuBLAS and cuSPARSE Results

For different problem sizes for the model problem, the calculation times for one iteration on the CPU and the GPU are computed. In this way the maximum speed-up can be computed. This speed-up can be obtained if enough iterations are performed on the GPU, so that the time needed to send information back and forth is negligibly small. The results can be seen in Table 7.8.

Unknowns	CPU (sec)	GPU (sec)	Max acceleration	Number of iterations
248,004	$1.992 \cdot 10^{-3}$	$3.730 \cdot 10^{-4}$	5.3	20,000
996,004	$9.519 \cdot 10^{-3}$	$1.148 \cdot 10^{-3}$	8.3	19,000
2,244,004	$2.401 \cdot 10^{-2}$	$2.435 \cdot 10^{-3}$	9.9	18,000
3,992,004	$4.543 \cdot 10^{-2}$	$4.219 \cdot 10^{-3}$	10.8	17,000
6,240,004	$7.225 \cdot 10^{-2}$	$6.472 \cdot 10^{-3}$	11.2	16,000
8,988,004	0.105	$9.353 \cdot 10^{-3}$	11.3	15,000
12,236,004	0.143	$1.267 \cdot 10^{-2}$	11.3	14,000
15,984,004	0.186	$1.650 \cdot 10^{-2}$	11.3	13,000
20,232,004	0.235	$2.085 \cdot 10^{-2}$	11.3	12,000
24,980,004	0.294	$2.576 \cdot 10^{-2}$	11.4	11,000
30,228,004	0.354	$3.115 \cdot 10^{-2}$	11.4	10,000

Table 7.8: Single precision results 2D Poisson model problem on CPU and GPU.

Small problems need again more iterations to obtain the maximum speed-up. Plotting the accelerations on the GPU compared to the CPU for different problem sizes gives the graph in Figure 7.4.

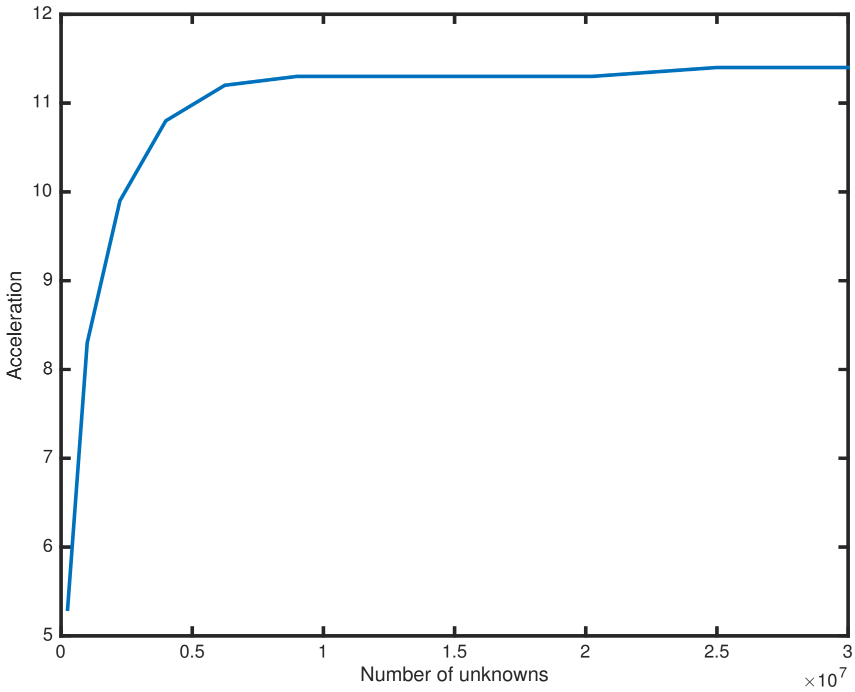

Figure 7.4: GPU acceleration compared to the CPU plotted against number of unknowns for the 2D Poisson model problem in single precision.

The GPU algorithm is approximately 11.4 times faster compared to the CPU algorithm. The CPU algorithm is approximately 1.6 times faster for single precision computations compared to double precision computations. The GPU algorithm is approximately 1.4 times faster for single precision computations compared to double precision computations.

7.6.2. Computational Efficiency and Memory Efficiency

For those single precision computations, the computational speed and the memory bandwidth are computed. In Table 7.9, for each operation it is computed how many bytes are needed for one iteration. Note that the computations are for a problem size of 24,980,004 instead of a problem size of 20,232,004 in the computational efficiency and memory efficiency computations for double precision computations in Section 7.5.2. The efficiency computations for single precision are for a problem size of 24,980,004, because of the fact that at this point the maximum speed-up is achieved for the first time.

The number of rows (and columns) are $c = 24,980,004$ and the number of nonzeros is $nnz = 124,880,028$. The computations are performed in single precision.

Step	**Calculation**	**Bytes**
Matrix vector product	$2 \cdot nnz \cdot 4 + (c+1) \cdot 4 + 2 \cdot c \cdot 4$	$1.299 \cdot 10^9$
Dot product	$2 \cdot c \cdot 4$	$1.998 \cdot 10^8$
Dot product	$2 \cdot c \cdot 4$	$1.998 \cdot 10^8$
Saxpy	$3 \cdot c \cdot 4$	$2.998 \cdot 10^8$
Sscal	$2 \cdot c \cdot 4$	$1.998 \cdot 10^8$
Saxpy	$3 \cdot c \cdot 4$	$2.998 \cdot 10^8$
Dot product	$2 \cdot c \cdot 4$	$1.998 \cdot 10^8$
Scopy	$2 \cdot c \cdot 4$	$1.998 \cdot 10^8$
Scopy	$2 \cdot c \cdot 4$	$1.998 \cdot 10^8$
Daxpy	$3 \cdot c \cdot 4$	$2.998 \cdot 10^8$

Table 7.9: Memory bandwidth calculations 2D Poisson model problem in single precision for a problem size of 24,980,004.

This means that in total $3.40 \cdot 10^9$ bytes = 3.40 GB is used. This is not half of the 5.02 GB used for double precision computations. This is because of the fact that for the CRS format for sparse matrix vector multiplications, the number of nonzeros per row and the column number are stored. These values are integers and therefore use 4 bytes. In single precision those values also use 4 bytes, so their memory is not halved. For single precision computations, 3.40 GB is used in $2.58 \cdot 10^{-2}$ seconds, so the memory bandwidth is 131.78 GB/sec.

For the computational speed the same table is created in 7.10. Again, the computations are for a problem size of 24,980,004.

Step	**Calculation**	**FLOPs**
Matrix vector product	$nnz + nnz - c$	$2.248 \cdot 10^8$
Dot product	$c + c - 1$	$4.996 \cdot 10^7$
Dot product	$c + c - 1$	$4.996 \cdot 10^7$
Saxpy	$c + c$	$4.996 \cdot 10^7$
Sscal	c	$2.498 \cdot 10^7$
Saxpy	$c + c$	$4.996 \cdot 10^7$
Dot product	$c + c - 1$	$4.996 \cdot 10^7$
Saxpy	$c + c$	$4.996 \cdot 10^7$

Table 7.10: Computational speed calculations 2D Poisson model problem in single precision for a problem size of 24,980,004.

This means that in total $5.495 \cdot 10^8$ flops = 0.550 GFLOPs are performed in $2.58 \cdot 10^{-2}$ seconds, so the computational speed is 21.30 GFLOP/sec.

On the GPU, the memory bandwidth for the algorithm is 131.78 GB/sec of a maximum of 288 GB/sec, which is almost 50%. This means that the bandwidth is quite high for a problem with 24,980,004 unknowns. The computational speed is 21.30 GFLOP/sec of a maximum of 4290 GFLOP/sec. This means that the computations are performed faster compared to the computations in double precision, but the computational efficiency is still less than 1%.

On the CPU, the memory bandwidth is $\frac{3.50}{0.292} = 11.99$ GB/sec (because of the extra vector). This is approximately 20% of the maximum memory bandwidth of the CPU. The computational speed is $\frac{0.550}{0.292} = 1.88$ GFLOP/sec. This is approximately 18% of the maximum computational speed of the CPU. These results are as expected, because of the fact that the memory bandwidth of the CPU was not full, so computations are not much faster now (they are just performed after each other and when the bandwidth is not full, computations are only performed faster because of the precision).

7.6.3. Computation Times of Specific Parts

To know which parts of the model problem are most time consuming, the calculation and communication times of the matrix fill, sending information to the GPU, the iterations and sending information back to the CPU are computed separately. Tables 7.11 and 7.12 show the calculation and communication times in different parts of the algorithm for the CPU and the GPU, respectively.

Unknowns	**Construct matrix**	**Iteration**
248,004	$9.947 \cdot 10^{-3}$	$1.99 \cdot 10^{-3}$
996,004	$3.684 \cdot 10^{-2}$	$9.59 \cdot 10^{-3}$
2,244,004	$8.094 \cdot 10^{-2}$	$2.40 \cdot 10^{-2}$
3,992,004	0.144	$4.54 \cdot 10^{-2}$
6,240,004	0.222	$7.19 \cdot 10^{-2}$
8,988,004	0.316	0.105
12,236,004	0.422	0.143
15,984,004	0.548	0.185
20,232,004	0.688	0.235
24,980,004	0.850	0.292
30,228,004	1.021	0.351

Table 7.11: Single precision calculation and communication times in different parts of the 2D Poisson model problem on the CPU.

Unknowns	**Construct matrix**	**Send information to GPU**	**Iteration**	**Send information back to CPU**
248,004	$6.96 \cdot 10^{-3}$	0.286	$3.73 \cdot 10^{-4}$	$3.25 \cdot 10^{-4}$
996,004	$2.48 \cdot 10^{-2}$	0.291	$1.14 \cdot 10^{-3}$	$8.80 \cdot 10^{-4}$
2,244,004	$5.66 \cdot 10^{-2}$	0.340	$2.44 \cdot 10^{-3}$	$1.85 \cdot 10^{-3}$
3,992,004	$9.85 \cdot 10^{-2}$	0.313	$4.21 \cdot 10^{-3}$	$3.21 \cdot 10^{-3}$
6,240,004	0.148	0.352	$6.52 \cdot 10^{-3}$	$4.93 \cdot 10^{-3}$
8,988,004	0.211	0.390	$9.36 \cdot 10^{-3}$	$7.07 \cdot 10^{-3}$
12,236,004	0.302	0.450	$1.27 \cdot 10^{-2}$	$9.61 \cdot 10^{-3}$
15,984,004	0.360	0.423	$1.65 \cdot 10^{-2}$	$1.25 \cdot 10^{-2}$
20,232,004	0.476	0.522	$2.10 \cdot 10^{-2}$	$1.58 \cdot 10^{-2}$
24,980,004	0.585	0.591	$2.58 \cdot 10^{-2}$	$1.95 \cdot 10^{-2}$
30,228,004	0.711	0.671	$3.11 \cdot 10^{-2}$	$2.36 \cdot 10^{-2}$

Table 7.12: Single precision calculation and communication times in different parts of the 2D Poisson model problem on the GPU.

The time for constructing the matrix, sending information to the GPU (in case of the GPU), one iteration and sending information back to the CPU (in case of the GPU) are plotted in different stacks in Figure 7.5.

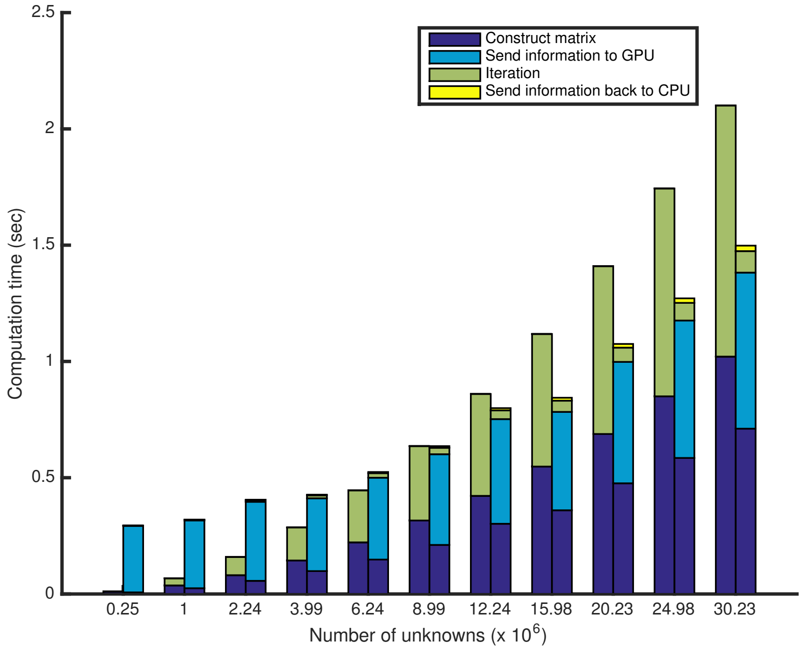

Figure 7.5: Single precision calculation and communication times in different parts plotted against number of unknowns for the 2D Poisson model problem for one iteration on the CPU and GPU.

One iteration on the GPU is much faster than one iteration on the CPU. Therefore, the GPU algorithm is already faster than the CPU for large problem sizes after two iterations.

7.7. Summary

Sending information from the CPU to the GPU is a time consuming operation (compared to computations). Therefore, for small problems the CPU operates mostly faster, compared to the GPU. For the 2D Poisson model problem in double precision discussed in this chapter, the GPU algorithm operates faster compared to the CPU algorithm for problem sizes larger than 75,000. The Poisson solver in double precision on the GPU for a problem size of 24,980,004, accelerates the algorithm approximately 11.5 times. This maximum can be explained with the use of computational speed and memory bandwidth: the bandwidth of the GPU is fully used, therefore computations cannot be performed faster. In a histogram with the calculation and communication times of the different parts of the model problem, it can be seen that the larger the problem, the faster the problem operates on the GPU. For this problem this is already seen after two iterations.

It is expected that single precision computations on the CPU are twice as fast compared to double precision computations. The GPU computations are also expected to be twice as fast. Because of the fact that the memory bandwidth is fully used for the double precision computations, the acceleration obtained for computations in single precision is negligibly small. This is because of the fact that all time is spent on communication: the acceleration in the computations are not noticed. Less data has to be sent to the GPU, so information is twice as fast available for computations. This means that the total speed-up of the GPU compared to the CPU is approximately the same as for double precision computations. The results are a bit different than expected: the CPU computations are approximately 1.6 times faster compared to computations in double precision and GPU computations are approximately 1.4 times faster compared to computations in double precision. The speed-up on the GPU compared to the CPU is approximately the same for single precision computations compared to double precision computations.

8

MLFMA Experiments

The focus of this thesis is the matrix vector multiplication of the MLFMA on a GPU. In the previous section computations are performed for a model problem. The sparse matrices of the MLFMA depend on degrees of freedom, wave directions and groups. Therefore, performing the matrix vector multiplications on a GPU is more complex for the MLFMA compared to the matrix vector multiplications of the Poisson solver. As a first step, a pure matrix vector multiplication algorithm is implemented.

This chapter explains the current implementation of the MLFMA in Shako, dismisses different implementations of the matrix vector multiplications on the GPU and measures computing times for three test problems. Finally results are discussed and conclusions are drawn.

8.1. Shako Implementation MLFMA

The MLFMA implementation of the NLR in Shako stores one real interpolation matrix for all groups. This means that the interpolation is implemented as matrix vector multiplications per group. The transfer operators are stored as function of the path between two groups, with a maximum of 316 paths per level. The transfer is not implemented as matrix vector multiplication, but in a memory efficient way.

In the GPU implementation, the different operators T, A and V (4.4) are stored as matrices. At first, the matrix vector multiplications are performed with large matrices: for all groups and all wave directions at the same time. Therefore, the three test problems probably use more memory on the GPU than in Shako, because of the fact that the transfer matrix has to be stored on the GPU.
In the next section, the baseline GPU implementation for the MLFMA is explained.

8.2. GPU Implementation MLFMA

On the GPU, all matrix vector multiplications are implemented with cuSPARSE. The matrices are not filled on the GPU, but read from files created by Shako on the CPU. These files contain information about the near matrix, the theta interpolation matrix, phi interpolation matrix and shift matrix for each level (together the interpolation) and the transfer matrix for each level. These files also provide information on the degrees of freedom, the wave direction, the group and the value for each non zero element in the matrix. The files also show the total number of nonzero elements and the total number of wave directions, groups and/or degrees of freedom.

With the use of this information and the knowledge about the building blocks of the MLFMA, the number of rows, the number of columns and the number of nonzeros have to be saved for each matrix. As explained before, for the sparse matrix vector multiplication the matrix is stored in the CRS format. This implies that the values and the column indices of the nonzeros have to be saved as well as the row offsets.

Because of the fact that those matrices depend on the wave directions and the groups, it is important to order the matrices and vectors in the same way. First, all wave directions are stored for the first group, then for the second group, until the wave directions within all groups at that level are stored in the matrix.

The conjugate transpose of matrix V and the transpose of the theta interpolation matrix, the phi interpolation matrix and the shift matrix are computed and stored in the CRS format for sparse matrices. This requires more memory on the GPU than computing the transpose on the GPU, but unfortunately the cuSPARSE routine that calculates the conjugate transpose directly during the call of the sparse matrix vector multiplication does not work.

The theta interpolation matrix and phi interpolation matrix are real matrices, while the rest of the matrices are complex. Unfortunately, it is not possible to multiply the real and the imaginary part of the complex vector separately by the real matrices on the GPU directly. This is only possible if the vector is sent back to the CPU, split in a real and a complex vector and sent back to the GPU. Because of the fact that sending information is most time consuming on the GPU this is not a referred way of performing this calculation. This means that the theta interpolation matrix and the phi interpolation matrix are stored as complex matrices and additional (useless) calculations are performed.

As explained in Section 4.1, the outgoing waves contain two components and the interpolation contains three components. This means for the implementation that 5 component vectors have to be stored in the GPU memory. Another option is to perform all computations with three components. This results in some extra work, but also saves memory on the GPU. The bottleneck for the GPU is the memory storage, so the implementation with three components in all calculations is used in this thesis.

The vectors with the specifications of the matrices and the starting vector are sent to the GPU and the correct settings for the GPU are applied. Now, everything is ready to start the matrix vector multiplications.
For the vector with three components, matrix matrix multiplications are performed instead of three separate matrix vector multiplications. The second matrix is the matrix with three columns: the vectors for the three components.

In the MLFMA, first the near interactions are calculated by multiplying the near matrix by the starting vector. This solution is added to the final solution of the far field interactions.
For the far interactions, first the waves are expanded by multiplying matrix V by the starting vector. This matrix V has three components, which means that this matrix vector multiplication has to be performed three times and gives a vector with three components.
The interpolation brings the outgoing waves to coarser levels, which means that this interpolation has to be performed multiple times until arrived at level $l = 2$. In the first iteration, the theta interpolation matrix is multiplied by the solution vector of Vx. After this, interpolation matrix phi is multiplied by the result and finally the shift matrix is multiplied by this result. After each iteration, the solution vector is saved, so that this vector can be used for the transfer at each level.
The transfer matrix for each level is multiplied by the result of the interpolation to that level.
From level $l = 2$ the incoming waves are anterpolated to the finest level, but at each level, the result of the transfer matrix matrix multiplication is added to the result of the anterpolation at that level.
Finally, the incoming waves are calculated by multiplying the conjugate transpose of matrix V by the sum of the solution of the final anterpolation and the transpose at the finest level. This matrix consists of three components, which means that one solution vector is obtained after taking the inner product of three separate solution vectors for the different components.

To get a better idea of the CPU and GPU performances and the maximum speed-up of the GPU, the MLFMA matrix vector multiplications of the MLFMA are applied multiple times. So, no iterative solver is implemented.

The GPU sends back the final vector $A\boldsymbol{x}$ to the CPU to print this final result on the CPU.

8.3. Expected Gain in Performance

The implementation method in the previous section required several decisions. To see if those decisions are optimal, the computation times of various implementations are compared. In Table 8.1 the names of different implementation methods are stated.

Implementation method	Explanation
Baseline	Implementation method explained in Section 8.2.
Onlyintreal	Storing interpolation operators as real matrices.
Reals	Performing matrix vector products in real arithmetic.
Intsmall	Storing interpolation operators as small matrices.
Groupinwave	Storing matrices in 'group within wave' structure.

Table 8.1: Different implementation methods for the matrix vector multiplications of the MLFMA.

In the *baseline* implementation, all matrices are saved as complex matrices, while in principle the theta and phi interpolation matrices are real. In the *onlyintreal* implementation the theta and phi interpolation matrices are saved as real matrices. Comparing those implementations it can be seen if the GPU prefers to work with more (useless) data or prefers to change the outgoing waves from complex to real and the other way around.
It is expected that the calculations on the GPU work faster if all matrices are stored as complex matrices. This is because when part of the matrices are stored as real matrices, the complex vector has to be sent back to the CPU to separate the complex vector in a real and an imaginary part. Sending information back and forth is the most time consuming part when working with a GPU. If the matrices are stored as complex matrices, some more work has to be done on the GPU, but this is probably much faster compared to sending the vector back and forth to split it in real/imaginary parts and another time to make the vector complex again. It is expected that the CPU calculations are just a bit slower because of the fact that the complex vector has to be split in a real and an imaginary part and after the calculations be joined together to one complex vector.

To prevent the change of the outgoing waves from complex to real, all matrices are saved as real matrices (meaning the complex matrices are stored as two real matrices) in the *reals* implementation. Separate real matrix vector multiplications for the real and the imaginary parts of the complex matrices are calculated.
In this case, vectors do not have to be sent back to the CPU and for the theta and phi interpolation matrices there are no useless calculations. But, more code has to be written for the complex matrices: the real part of the matrix multiplied by the real part of the vector minus the imaginary part of the matrix multiplied by the imaginary part of the vector gives the next real vector and the real part of the matrix multiplied by the imaginary part of the vector plus the imaginary part of the matrix multiplied by the real part of the vector gives the next imaginary vector. In the complex matrix matrix multiplications in cuSPARSE this happens in an efficient way. It is expected that a direct implementation takes more time compared to using the optimal cuSPARSE routine for complex matrix matrix multiplications (and performing some useless calculations for the theta and phi interpolation matrices). Also on the CPU this is expected to perform a bit slower, because of the fact that the optimal sparse matrix routine for complex matrix matrix multiplications is expected to work faster.

In the theta and phi interpolation matrices, the wave directions are the same for each group. Therefore, it is possible to store just the wave directions for one group in a smaller matrix, which saves memory storage. This is done in the *intsmall* implementation to see if this results in approximately the same computation time compared to the *baseline* implementation.
Storing the total theta and phi interpolation matrices is a waste of memory, because of the fact that the wave directions are the same for each group. Storing smaller theta and phi interpolation matrices saves memory, but is expected to makes the calculations slower. This is because of the fact that less calculations can be performed in parallel on the GPU. The GPU wants to work with as much data as possible at the same time, so this would probably give a slower computation time of the algorithm on the GPU. The calculations are expected to perform a bit slower on the CPU because of the fact that the CPU prefers to work with larger data sets.

In the *baseline* implementation, wave directions are stored within the groups. This can also be done the other way around, by storing the groups within the wave directions. This is done in the *groupinwave* implementation. It is expected that storing the groups within the wave directions operates slower compared to storing the wave directions within the groups. This is because of the fact that the interpolation matrices lose their structure, which probably makes the calculations slower. Also for matrix V the matrix is more structured when the wave directions are stored within the groups, which is expected to work faster compared to an unstructured matrix. This process is also expected to perform slower on the CPU, but less so than on the GPU.

Furthermore, it is expected that the interpolation multiplication obtains more speed-up on the GPU compared to the transfer multiplication. This is because of the fact that the interpolation matrix is locally ordered with six entries on one row and the transfer matrix is unordered. The GPU is expected to work slower with unstructured matrices compared to the CPU.

Summarizing the expectations:

- Theta and phi interpolation matrices saved as complex matrices are considerably faster compared to theta and phi saved as real interpolation matrices or compared to all matrices and vectors saved as reals.
- Theta and phi interpolation matrices saved as smaller matrices are slower compared to storing the whole theta and phi interpolation matrices.
- Storing wave directions within groups is faster compared to storing groups within wave directions.
- Interpolation matrix obtains more speed-up on GPU compared to the transfer matrix.

8.4. Setting Information

To compare CPU and GPU performances for the MLFMA, calculation and communication times are determined for the specific part where the GPU can accelerate the algorithm. This means that on the CPU the calculation time of the iterations is calculated and on the GPU the calculation time for the iterations and the communication time to send information back and forth between the CPU and the GPU are calculated.

In the next sections computation times of different implementation methods are compared to see which implementation method obtains the maximum speed-up compared to the CPU implementation. To test this, different test problems are used. In Table 8.2 the problem sizes of the three test problems are given.

Name	Problem size
Test 1	5256
Test 2	21,453
Test 3	79,638

Table 8.2: Problem sizes of the three test problems.

8.5. Test 1: Small Test Problem for MLFMA with 4 Levels and 5,000 Unknowns

In test 1 a simplification of the real MLFMA is performed for a small problem with 5256 unknowns, where the matrices are read and matrix vector multiplications are calculated to see if this yields an acceleration on the GPU. The problem has maximum level $l_{max} = 4$ and minimum level $l_{min} = 2$ and 5,256 unknowns or degrees of freedom. At level $l = 4$ there are 436 groups and 98 wave directions, at level $l = 3$ there are 112 groups and 242 wave directions and at level $l = 2$ there are 28 groups and 648 wave directions. As starting vector the 333th unit vector is used. All calculations are performed in single precision.

8.5.1. Memory Use

The GPU memory contains information about the matrices and solution vectors of each matrix vector multiplication. The matrix sizes are of great importance for the computation of the amount of used memory. In Table 8.3 the number of rows, number of columns and number of non zeros are given for each matrix of the MLFMA for test 1.

Matrix	nrow	ncol	nnz	Memory use (MB)
Near matrix	5,256	5,256	853,186	10.3
V_1	42,728	5,256	515,088	6.4
$\text{Theta}_{l=3}$	67,144	42,728	402,864	
$\text{Phi}_{l=3}$	105,512	67,144	633,072	
$\text{Shift}_{l=3}$	27,104	105,512	105,512	24.3
$\text{Theta}_{l=2}$	44,352	27,104	266,112	
$\text{Phi}_{l=2}$	72,576	44,352	435,456	
$\text{Shift}_{l=2}$	18,144	72,576	72,576	
$T_{l=4}$	42,728	42,728	1,770,272	
$T_{l=3}$	27,104	27,104	1,294,700	40.4
$T_{l=2}$	18,144	18,144	276,048	
$\text{Shift}_{l=3}$	72,576	18,144	72,576	
$\text{Phi}_{l=3}$	44,352	72,576	435,456	
$\text{Theta}_{l=3}$	27,104	44,352	206,112	30.4
$\text{Shift}_{l=4}$	105,512	27,104	105,512	
$\text{Phi}_{l=4}$	67,144	105,512	633,072	
$\text{Theta}_{l=4}$	42,728	67,144	402,864	
V_2	5,256	42,728	515,088	6.2

Table 8.3: Information about the matrices of the MLFMA for test 1.

The matrices and vectors of test 1 use in total 149,867,848 bytes = 0.150 GB memory. The GPU has a total memory of 12 GB, so approximately 1% of the total memory is used.

The Transfer matrices use most of the GPU memory. Also the interpolation and anterpolation matrices use a large amount of memory.

8.5.2. Baseline Implementation

For different numbers of iterations, computation times of the CPU and the GPU are compared. On the CPU the calculation time of the MLFMA matrix vector multiplications is computed and on the GPU the communication time and the calculation time of the MLFMA matrix vector multiplications are computed. The computation times can be seen in Table 8.4.

Iterations	Time CPU (sec)	Time GPU (sec)	Acceleration
500	18.390	4.072	4.5
1,000	36.711	5.278	7.0
1,500	55.142	6.470	8.5
2,000	73.654	7.694	9.6
2,500	92.750	8.919	10.4
3,000	110.412	10.117	10.9
3,500	129.246	11.354	11.4
4,000	148.061	12.577	11.8
4,500	165.912	13.799	12.0
5,000	184.186	14.995	12.3
10,000	368.372	27.216	13.5
50,000	1,841.860	124.289	14.8
100,000	3,683.720	245.659	15.0
150,000	5,525.580	368.029	15.1

Table 8.4: CPU and GPU computation times for the MLFMA baseline implementation for test 1.

The calculation time for one iteration on the CPU is approximately $3.671 \cdot 10^{-2}$ seconds and the calculation time for one iteration on the GPU is approximately $2.427 \cdot 10^{-3}$ seconds. This means that the maximum speed-up is 15 if enough iterations are performed. Sending the matrices to the GPU takes approximately 2.856 seconds and sending the vector $A\boldsymbol{x}$ back to the CPU takes approximately $6.535 \cdot 10^{-2}$ seconds. This time for sending information becomes negligibly small if several iterations are performed.

For a real problem in Shako, the GMRES method needs approximately 100 iterations to converge. For real life problems the algorithm needs to be solved for multiple, let's say 100, right hand sides. This means that the matrices can stay in the GPU memory, but the result has to be sent back to the CPU after 100 iterations for one right hand side. In total 10,000 iterations are performed and the solution vector has to be sent back to the CPU 100 times. This means that the communication time becomes $100 \cdot 6.535 \cdot 10^{-2} + 2.856 = 9.391$ seconds. The speed-up on the GPU for 10,000 iterations for this problem is 10.9, which is still significantly.

The computation times on the CPU and GPU and the speed-up for different numbers of iterations are given in Figure 8.1 and 8.2.

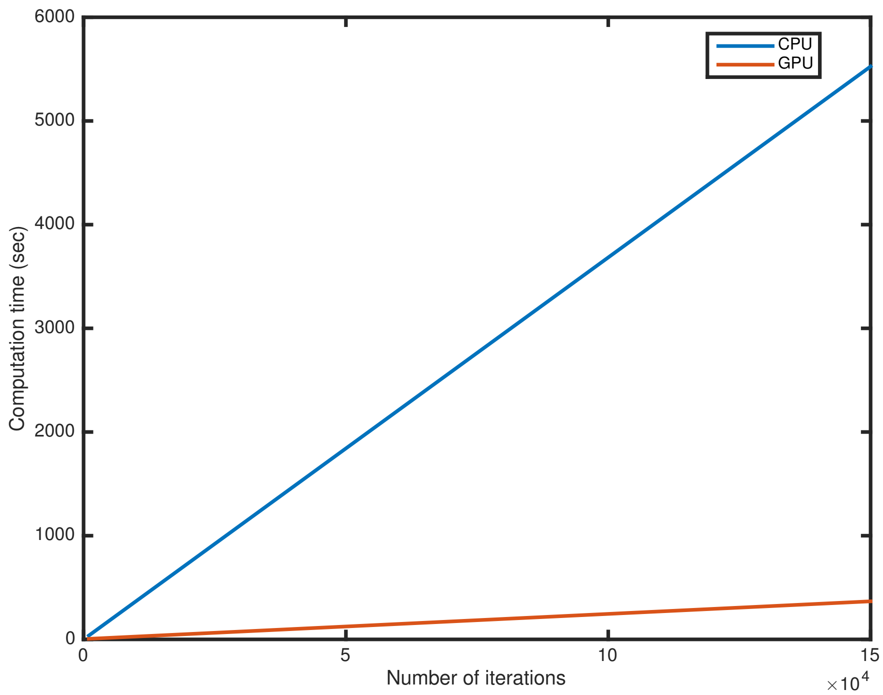

Figure 8.1: CPU and GPU computation times plotted against the number of iterations for the MLFMA baseline implementation for test 1.

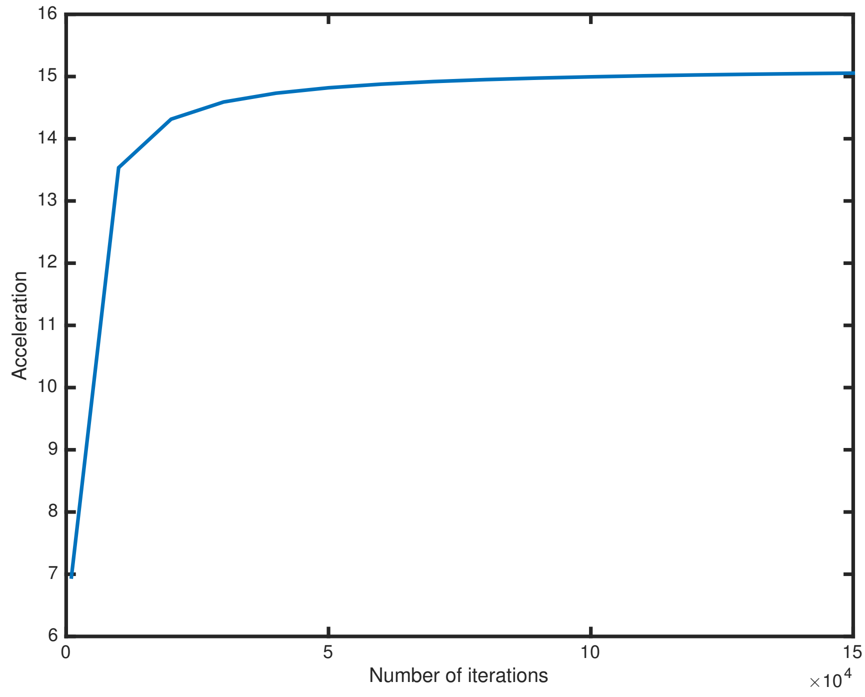

Figure 8.2: GPU acceleration compared to the CPU plotted against the number of iterations for the MLFMA baseline implementation for test 1.

After a number of approximately 75,000 iterations, the speed-up has a maximum value of 15. In real problems, not so many iterations are performed.

8.5.3. Computational Efficiency and Memory Efficiency

The computational efficiency and memory efficiency give insights in the calculations and help to explain the computation times. In this MLFMA algorithm, only complex matrix vector or matrix matrix multiplications are performed and all calculations are performed in single precision. This means that for each matrix vector product, 2 vectors of length nnz (the value of the non zero and the column index), 2 vectors of length $nrow$ (the row offsets and the solution vector) and one vector of length $ncol$ (the vector for calculations) are sent. This gives the following formula: $nnz \cdot 2 \cdot 4 + nrow \cdot 2 \cdot 4 + ncol \cdot 4$ bytes, where nnz is the number of non zeros for the specific matrix, $nrow$ is the number of rows for the specific matrix and $ncol$ is the number of columns for the specific matrix. For matrix matrix multiplications, the three vectors for the components and the three solution vectors are sent: $nnz \cdot 2 \cdot 4 + nrow \cdot 4 + nrow \cdot 3 \cdot 4 + ncol \cdot 3 \cdot 4$ bytes. Also some copies and caxpy's are performed in the calculations, those calculations need $2 \cdot ncol \cdot 4$ bytes and $3 \cdot ncol \cdot 4$ bytes, respectively.

The numbers of rows, columns and non zeros can be found in Table 8.3 in Section 8.5.1.

Calculating the number of bytes needed for one iteration in the *baseline* implementation gives $115,508,288$ bytes = 0.116 GB. One iteration of the *baseline* implementation takes 0.00243 seconds, so the memory bandwidth for this algorithm on the GPU is 47.53 GB/sec. This is approximately 16.5% of the maximum memory bandwidth, so the bandwidth is not full.
In the CPU algorithm, for each matrix vector or matrix matrix product, an extra vector of length $nrow$ is needed. Therefore, the total number of bytes needed is 0.116 GB. One iteration of the *baseline* implementation takes 0.0368 seconds, so the memory bandwidth for this algorithm on the CPU is 3.16 GB/sec. This is approximately 5% of the maximum memory bandwidth, so the bandwidth is not full.

For the computational speed computations, each complex matrix vector multiplication uses $(nnz + nnz - nrow) \cdot 4$ FLOPs. Each matrix matrix multiplication uses $(nnz + nnz - nrow) \cdot 4 \cdot 3$ FLOPs, because of the three components. The caxpy cuBLAS routine uses $(c + c) \cdot 4$ FLOPs. The total number of GFLOPs is 0.195 in 0.00243 seconds on the GPU algorithm. This means that the computational speed is 80.12 GFLOP/sec, which is approximately 2% of the maximum computational speed of the GPU.

The computational speed for the algorithm on the CPU is $\frac{0.195}{0.037} = 5.29$ GFLOP/sec, which is approximately 51% of the total computational speed of the CPU. This means that the computations on the CPU are performed quite fast.

8.5.4. Storing the Interpolation Operators as Real Matrices

The only difference of the *onlyintreal* implementation compared to the *baseline* implementation is the fact that the theta and phi interpolation matrices are stored as real matrices. This means that less computations have to be performed compared to the *baseline* implementation where the theta and phi interpolation matrices are stored as complex matrices (while they are real). For the *onlyintreal* GPU implementation, the complex solution vector has to be sent back to the CPU in each iteration to split the vector in a real part and an imaginary part.

In Table 8.5 the computation times of the CPU and the GPU for the *onlyintreal* implementation can be seen.

Iterations	Time CPU (sec)	Time GPU (sec)
500	26.775	65.620
1,000	53.574	110.686
1,500	80.370	167.878
2,000	107.119	223.073
2,500	132.813	283.910
3,000	161.061	343.629
3,500	187.425	406.968
4,000	215.001	470.737

Table 8.5: CPU and GPU computation times for the MLFMA onlyintreal implementation for test 1.

Comparing these results with the *baseline* implementation gives the plot in Figure 8.3.

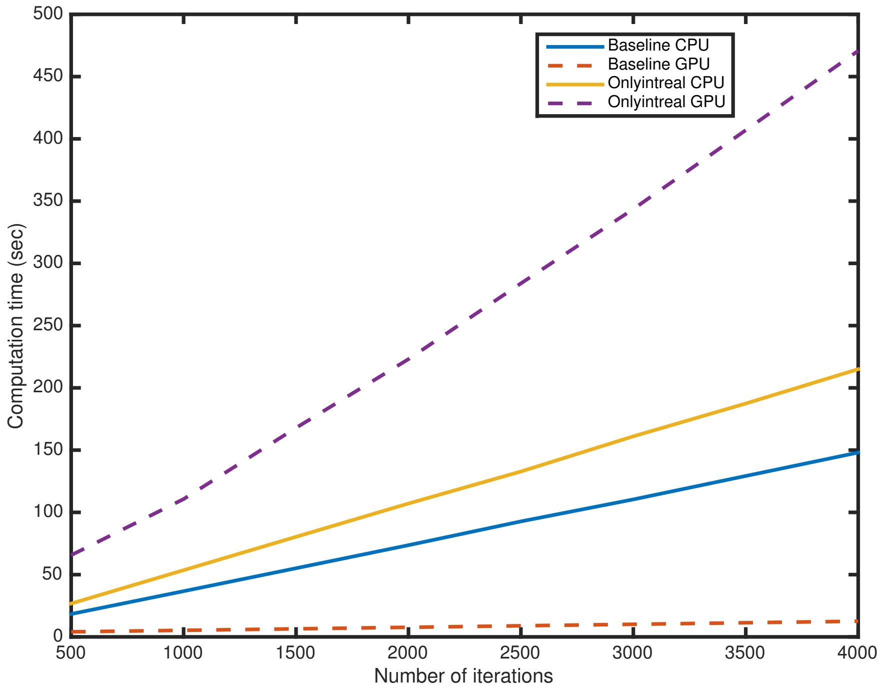

Figure 8.3: CPU and GPU computation times plotted against the number of iterations for the MLFMA baseline implementation and the MLFMA onlyintreal implementation for test 1.

The computation time for the CPU algorithm is a bit slower for the *onlyintreal* implementation. This is because of the fact that complex vectors are split in a real and an imaginary part and this takes some time. The computation time on the GPU is approximately 37 times slower, because of the fact that in each iteration, the complex vector is sent back to the CPU to split the vector in a real and an imaginary part. Sending information is the bottleneck on the GPU and very time consuming. Therefore this *onlyintreal* implementation is much slower compared to the *baseline* implementation on the GPU.

8.5.5. Performing Matrix Vector Products in Real Arithmetic

In the *reals* implementation, all vectors and matrices are stored as reals. This means that for complex matrices, two real matrices are stored: one for the real part and one for the imaginary part. The *reals* implementation and the *baseline* implementation perform exactly the same computation, but the *baseline* implementation uses a direct complex cuBLAS routine, while the *reals* implementation uses four real cuBLAS routines.

In Table 8.6 the computation times on the CPU and the GPU for the *reals* implementation can be seen.

Iterations	Time CPU (sec)	Time GPU (sec)
500	39.627	5.925
1,000	78.527	8.957
1,500	119.013	12.001
2,000	157.054	15.021
2,500	196.681	18.061
3,000	235.581	21.114
3,500	275.208	24.120
4,000	314.108	27.124

Table 8.6: CPU and GPU computation times for the MLFMA reals implementation for test 1.

Comparing these results with the *baseline* implementation gives the plot in Figure 8.4.

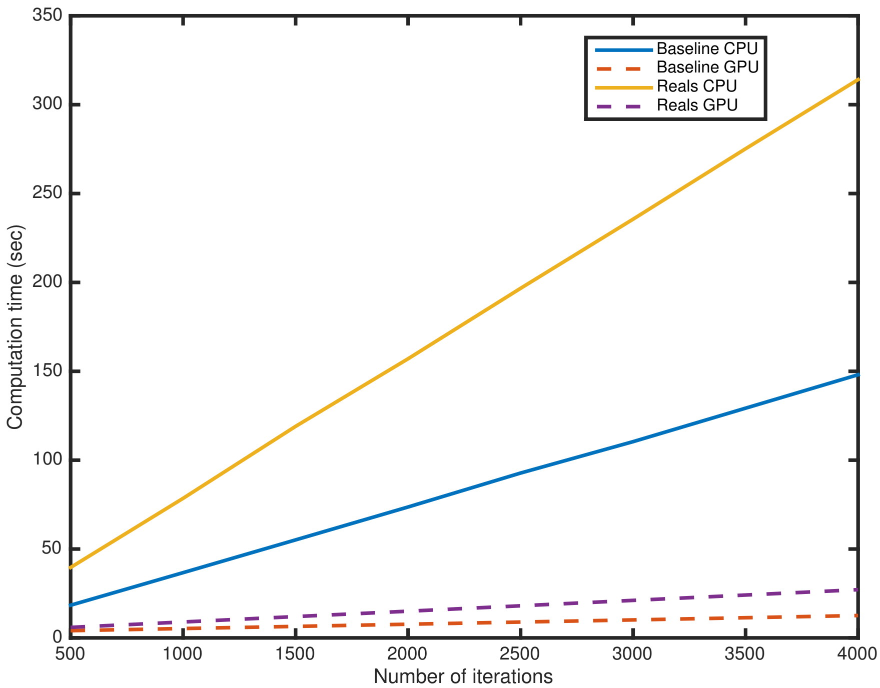

Figure 8.4: CPU and GPU computation times plotted against the number of iterations for the MLFMA baseline implementation and the MLFMA reals implementation for test 1.

The computation times for both the CPU and the GPU are approximately two times slower for the *reals* implementation compared to the *baseline* implementation. This is because of the fact that the computations in the complex cuBLAS routines are performed in the an efficient way, while computing multiple real cuBLAS routines are not that efficient. In the complex cuBLAS routine, the computations are divided more efficiently over the threads and blocks of the GPU compared to separate real cuBLAS routines, because of the fact that more computations can be performed in parallel.

8.5.6. Storing the Interpolation Operators as Small Matrices

In the *intsmall* implementation, information about the wave directions in the theta and phi interpolation matrices is saved for just a single group. This is possible, because of the fact that for each group the wave directions are the same. This results in smaller theta and phi interpolation matrices, which saves memory storage and data transport.

In Table 8.7 the computation times of the CPU and the GPU for the *intsmall* implementation can be seen.

Iterations	**Time CPU (sec)**	**Time GPU (sec)**
500	21.968	32.468
1,000	44.058	64.656
1,500	65.980	96.844
2,000	88.232	129.031
2,500	110.210	160.039
3,000	132.296	191.047
3,500	154.264	222.132
4,000	176.464	253.216

Table 8.7: CPU and GPU computation times for the MLFMA intsmall implementation for test 1.

In the *intsmall* implementation, it is not possible to make use of matrix matrix multiplications for the three components, because of the fact that only a part of each component is used for a multiplication. To make a fair comparison, the results of the *intsmall* implementations are compared to the *baseline* implementation with matrix vector multiplications instead of matrix matrix multiplications for the theta and phi interpolation matrices (which implementation times are approximately 1% higher on the CPU and 25% higher on the GPU). This gives the plot in Figure 8.5.

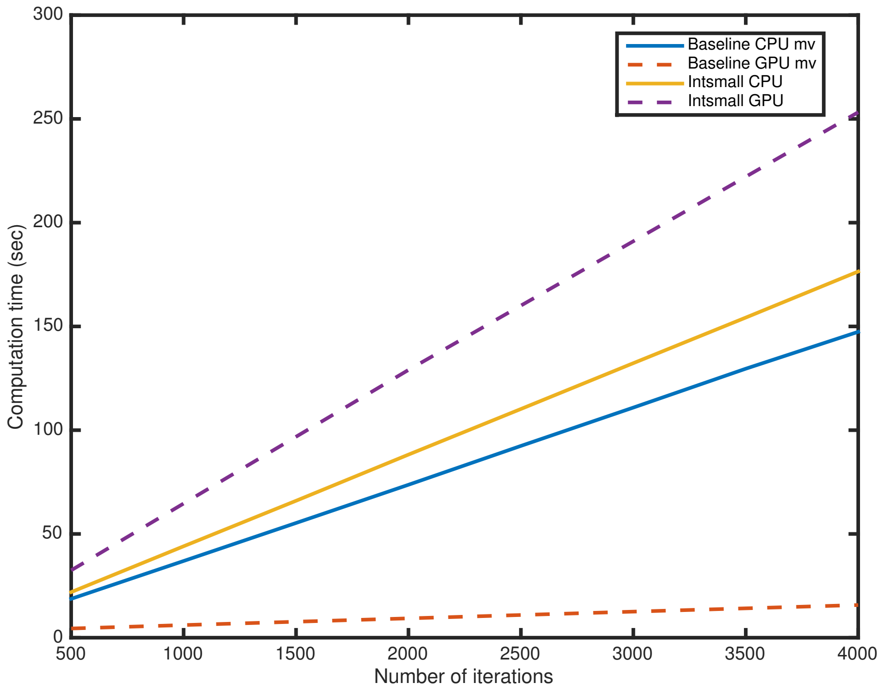

Figure 8.5: CPU and GPU computation times plotted against the number of iterations for the MLFMA baseline implementation with matrix vector multiplications and the MLFMA intsmall implementation for test 1.

Both computation times on the CPU and the GPU for the *intsmall* implementation are slower compared to the *baseline* implementation with matrix vector products for the theta and the phi interpolation matrices. For the CPU algorithm the computation times are only 1.2 times slower, but for the GPU algorithm they are 16 times slower. Here, the characteristics of the GPU are shown very clearly: it achieves the significant speed-ups because of the parallel computations. In this *intsmall* implementation, relatively small matrix vector multiplications are computed, so not much can be performed in parallel. The CPU also prefers to work with large data sets, but the CPU is clearly less affected by working with less data at the same time.

8.5.7. Storing Matrices in Group Within Wave Structure

In the *groupinwave* implementation, matrices are filled in a way that groups are stored within the wave directions. In the *baseline* implementation the matrices are filled in a way that the wave directions are stored within the groups.

In Table 8.8 the computation times of the CPU and the GPU for the *groupinwave* implementation can be seen.

Iterations	**Time CPU (sec)**	**Time GPU (sec)**
500	16.591	4.181
1,000	33.176	5.394
1,500	49.758	6.608
2,000	66.141	7.821
2,500	82.732	9.036
3,000	99.317	10.250
3,500	115.908	11.489
4,000	132.282	12.728

Table 8.8: CPU and GPU computation times for the MLFMA groupinwave implementation for test 1.

Comparing these results with the *baseline* implementation gives the plot in Figure 8.6.

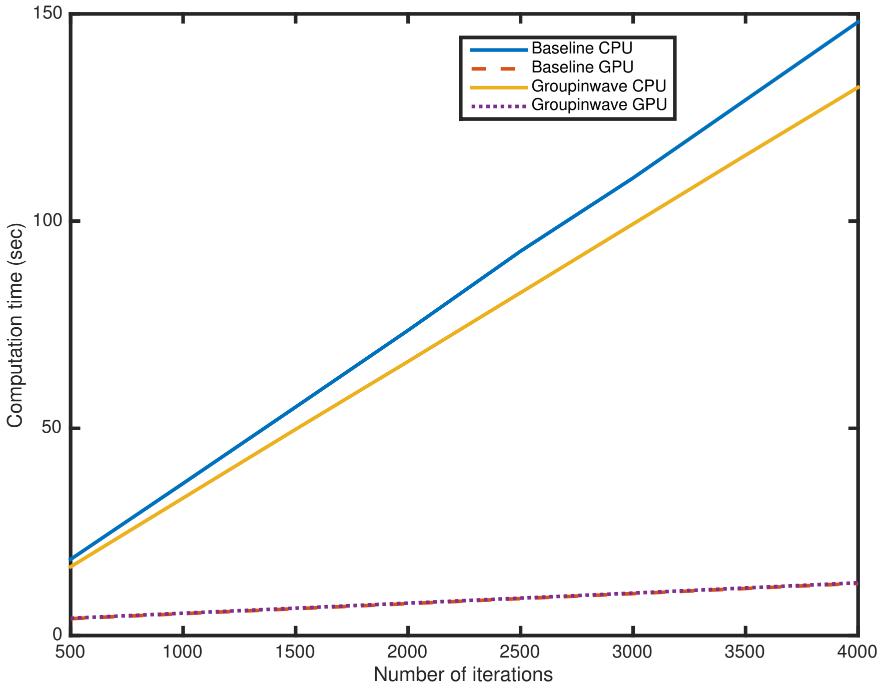

Figure 8.6: CPU and GPU computation times plotted against the number of iterations for the MLFMA baseline implementation and the MLFMA groupinwave implementation for test 1.

The computation times on the CPU and the GPU for the *groupinwave* implementation are other than expected: the *groupinwave* GPU implementation has the same computation time as the *baseline* implementation for the GPU. This means that the GPU does not work slower when unstructured matrix vector multiplications are performed. The *groupinwave* CPU implementation is even 10% faster compared to the *baseline* implementation. It appears this group in wave structure operates even better on the CPU. It is possible that the difference is so small because of the fact that the structure is only for a small number of elements.

8.5.8. Comparison of Implementations

An overview of the speed-ups of the different implementation methods can be seen in Table 8.9.

Implementation method	Acceleration
Baseline	15.1
Onlyintreal	0.40
Reals	13.1
Intsmall	0.71
Groupinwave	13.3

Table 8.9: GPU acceleration compared to the CPU for different implementation methods for test 1.

The *baseline* implementation obtains, as expected, the largest acceleration.

8.5.9. Parts of the Baseline Implementation

It is important to know which parts of the *baseline* implementation take the most time. Therefore the calculation times of different parts of the *baseline* implementation on the CPU and the GPU are calculated for one iteration in Table 8.10.

Performance	Time CPU (sec)	Time GPU (sec)	Acceleration
Near matrix	$9.382 \cdot 10^{-4}$	$1.349 \cdot 10^{-5}$	70
Expand waves	$1.981 \cdot 10^{-3}$	$2.922 \cdot 10^{-4}$	6.8
Interpolation to level 3	$5.862 \cdot 10^{-3}$	$6.407 \cdot 10^{-4}$	9.1
Interpolation to level 2	$3.943 \cdot 10^{-3}$	$8.730 \cdot 10^{-5}$	45
Transfer at level 4	$5.585 \cdot 10^{-3}$	$1.198 \cdot 10^{-5}$	466
Transfer at level 3	$6.822 \cdot 10^{-3}$	$3.461 \cdot 10^{-5}$	197
Transfer at level 2	$1.008 \cdot 10^{-3}$	$9.996 \cdot 10^{-5}$	10
Anterpolation to level 3	$3.501 \cdot 10^{-3}$	$3.496 \cdot 10^{-4}$	10
Anterpolation to level 4	$5.325 \cdot 10^{-3}$	$1.915 \cdot 10^{-4}$	28
Incoming waves	$1.644 \cdot 10^{-6}$	$9.890 \cdot 10^{-8}$	18

Table 8.10: CPU and GPU calculation times in different parts of the MLFMA baseline implementation for test 1.

The interpolation matrix vector multiplications on the GPU are not accelerated more compared to the transfer matrix vector multiplications. This shows again that the structure does not matter for the calculations on the GPU. It is very clearly that the transfer matrix vector multiplications on the levels 4 and 3 are sequentially faster on the GPU, this is because of the fact that the number of nonzeros is considerably larger for those matrix vector multiplications so more computations can be performed in parallel.

8.6. Test 2: Larger Test Problem for MLFMA with 5 Levels and 20,000 Unknowns

Test 2 is a larger problem with maximum level $l_{max} = 5$, minimum level $l_{min} = 2$ and 21,453 unknowns or degrees of freedom. At level $l = 5$ there are 1,839 groups and 98 wave directions, at level $l = 4$ there are 471 groups and 242 wave directions, at level $l = 3$ there are 114 groups and 648 wave directions and at level $l = 2$ there are 28 groups and 1,800 wave directions. As starting vector the 333th unit vector is used. All calculations are performed in single precision.

This problem probably achieves the maximum speed-up faster compared to the problem in test 1, since more calculations are performed in parallel.

8.6.1. Memory Use

The GPU memory contains information about the matrices and solution vectors of each matrix vector multiplication. The matrix sizes are of great importance for the computation of the amount of used memory. In Table 8.11 the number of rows, number of columns and number of non zeros is given for each matrix.

Matrix	**nrow**	**ncol**	**nnz**
Near matrix	21,453	21,453	3,523,269
V_1	180,222	21,453	2,102,394
$\text{Theta}_{l=4}$	283,206	180,222	1,699,236
$\text{Phi}_{l=4}$	445,038	283,206	2,670,228
$\text{Shift}_{l=4}$	113,982	445,038	445,038
$\text{Theta}_{l=3}$	186,516	113,982	1,119,096
$\text{Phi}_{l=3}$	305,208	186,516	1,831,248
$\text{Shift}_{l=3}$	73,872	305,208	305,208
$\text{Theta}_{l=2}$	123,120	73,872	738,720
$\text{Phi}_{l=2}$	205,200	123,120	1,231,200
$\text{Shift}_{l=2}$	50,400	205,200	205,200
$T_{l=5}$	180,222	180,222	7,699,272
$T_{l=4}$	113,982	113,982	5,130,400
$T_{l=3}$	73,872	73,872	3,551,040
$T_{l=2}$	50,400	50,400	766,800
$\text{Shift}_{l=3}$	205,200	50,400	205,200
$\text{Phi}_{l=3}$	123,120	205,200	1,231,200
$\text{Theta}_{l=3}$	73,872	123,120	738,720
$\text{Shift}_{l=4}$	305,208	73,872	305,208
$\text{Phi}_{l=4}$	186,516	305,208	1,831,248
$\text{Theta}_{l=4}$	113,982	186,516	1,119,096
$\text{Shift}_{l=5}$	445,038	113,982	445,038
$\text{Phi}_{l=5}$	283,206	445,038	2,670,228
$\text{Theta}_{l=5}$	180,222	283,206	1,699,236
V_2	21,453	180,222	2,102,394

Table 8.11: Information about the matrices of the MLFMA for test 2.

The information for test 2 uses 738,909,372 bytes = 0.739 GB of memory. The GPU has a total memory of 12 GB, so approximately 6% of the total memory is used.

8.6.2. Baseline Implementation

The results of the *baseline* implementation can be seen in Table 8.12.

Iterations	Time CPU (sec)	Time GPU (sec)	Acceleration
250	51.466	6.890	7.5
500	102.808	9.745	10.5
750	154.265	12.600	12.2
1,000	205.693	15.454	13.3
1,250	257.162	18.347	14.0
1,500	308.534	21.240	14.5
1,750	360.118	24.105	14.9
2,000	411.451	26.969	15.3
5,000	1028.465	61.643	16.7
10,000	2056.930	119.218	17.3
15,000	3085.395	176.793	17.5

Table 8.12: CPU and GPU computation times for the MLFMA baseline implementation for test 2.

The computation time for one iteration on the CPU is approximately $0,206$ seconds and the computation time for one iteration on the GPU is approximately $1.143 \cdot 10^{-2}$ seconds. This means that the maximum speed-up is 18 if enough iterations are performed. Sending the matrices to the GPU takes approximately 3.996 seconds and sending the vector $A\boldsymbol{x}$ back to the CPU takes approximately 0.243 seconds. This time for sending information becomes negligibly small if several iterations are performed.

8.6.3. Storing the Interpolation Operators as Real Matrices

The computation times on the CPU and the GPU for the *onlyintreal* implementation can be seen in Table 8.13.

Iterations	Time CPU (sec)	Time GPU (sec)
250	75.697	131.678
500	151.312	288.888
750	226.650	446.098
1,000	302.100	603.308
1,250	377.656	760.518
1,500	453.052	917.728
1,750	529.413	1074.938
2,000	605.043	1232.148

Table 8.13: CPU and GPU computation times for the MLFMA onlyintreal implementation for test 2.

The result is the same as for test 1: the *onlyintreal* implementation on the GPU is much slower because of the fact that in each iteration data has to be sent to the CPU.

8.6.4. Performing Matrix Vector Products in Real Arithmetic

The computation times on the CPU and the GPU for the *reals* implementation can be seen in Table 8.14.

Iterations	Time CPU (sec)	Time GPU (sec)
250	114.709	11.037
500	228.089	17.686
750	342.485	24.335
1,000	456.732	30.983
1,250	571.349	37.632
1,500	685.594	44.355
1,750	799.860	51.004
2,000	914.125	57.653

Table 8.14: CPU and GPU computation times for the MLFMA reals implementation for test 2.

The results of test 1 are confirmed. The *reals* implementation is again twice as slow as the *baseline* implementation, because of the fact that less computations can be performed in parallel.

8.6.5. Storing the Interpolation Operators as Small Matrices

The computation times on the CPU and the GPU for the *intsmall* implementation can be seen in Table 8.15.

Iterations	Time CPU (sec)	Time GPU (sec)
250	59.984	73.444
500	120.089	146.834
750	180.209	220.224
1,000	239.640	293.614
1,250	299.717	367.004
1,500	359.778	440.394
1,750	420.380	513.784
2,000	479.320	587.174

Table 8.15: CPU and GPU computation times for the MLFMA intsmall implementation for test 2.

The computation times of the *intsmall* implementation on the CPU and the GPU are again slower compared to the *baseline* implementation with matrix vector products for the theta and phi interpolation matrices. This is because of the fact that both the CPU and the GPU prefer to work with larger data sets.

8.6.6. Storing Matrices in Group Within Wave Structure

The computation times on the CPU and the GPU for the *groupinwave* implementation can be seen in Table 8.16.

Iterations	Time CPU (sec)	Time GPU (sec)
250	43.932	7.053
500	87.812	9.947
750	131.701	12.841
1,000	175.543	15.734
1,250	219.424	18.628
1,500	264.415	21.570
1,750	308.430	24.464
2,000	352.491	27.358

Table 8.16: CPU and GPU computation times for the MLFMA groupinwave implementation for test 2.

The computation time of GPU for the *groupinwave* implementation is again the same as for the *baseline* implementation. This means that the *baseline* implementation does not have a better structure compared to the *groupinwave* implementation. On the CPU the *groupinwave* implementation is even faster compared to the *baseline* implementation. It could be that the *groupinwave* implementation has accidentally a better structure for the CPU compared to the *baseline* implementation. This is something to investigate in future research.

8.6.7. Parts of the Baseline Implementation

As for test 1, the *baseline* implementation performs the best on the GPU for test 2.

It is important to know which parts of the *baseline* implementation are most time consuming. Therefore the calculation times of the different parts of the *baseline* implementation on the CPU and the GPU are calculated for one iteration in Table 8.17.

Performance	**Time CPU (sec)**	**Time GPU (sec)**	**Acceleration**
Near matrix	$3.911 \cdot 10^{-3}$	$2.560 \cdot 10^{-4}$	15
Expand waves	$8.385 \cdot 10^{-3}$	$7.504 \cdot 10^{-4}$	11
Interpolation to level 4	$2.731 \cdot 10^{-2}$	$3.591 \cdot 10^{-4}$	76
Interpolation to level 3	$1.751 \cdot 10^{-2}$	$3.210 \cdot 10^{-5}$	545
Interpolation to level 2	$1.151 \cdot 10^{-2}$	$7.111 \cdot 10^{-4}$	16
Transfer at level 5	$2.855 \cdot 10^{-2}$	$2.404 \cdot 10^{-4}$	119
Transfer at level 4	$2.905 \cdot 10^{-2}$	$1.711 \cdot 10^{-5}$	1.698
Transfer at level 3	$1.697 \cdot 10^{-2}$	$1.691 \cdot 10^{-5}$	1.003
Transfer at level 2	$5.624 \cdot 10^{-3}$	$1.707 \cdot 10^{-5}$	329
Anterpolation to level 3	$1.025 \cdot 10^{-2}$	$1.141 \cdot 10^{-3}$	9
Anterpolation to level 4	$1.587 \cdot 10^{-2}$	$3.109 \cdot 10^{-3}$	5
Anterpolation to level 5	$2.387 \cdot 10^{-2}$	$4.829 \cdot 10^{-4}$	49
Incoming waves	$7.273 \cdot 10^{-5}$	$6.140 \cdot 10^{-6}$	12

Table 8.17: CPU and GPU calculation times in different parts of the MLFMA baseline implementation for test 2.

The interpolation matrix vector multiplications on the GPU are again not accelerated more compared to the transfer matrix vector multiplications. The structure is too little to matter for the calculations on the GPU. The transfer matrix vector multiplications on the levels 3 and 4 are sequentially faster on the GPU. It is expected that the transfer at level 5 obtains more speed-up than 119, because of the fact that the matrix is very large. It could be that the bandwidth is full and therefore threads have to wait to start the computations.

8.7. Test 3: Largest Test Problem for MLFMA with 6 levels and 80,000 Unknowns

Test 3 is a problem approximately 4 times larger compared to the problem in test 2. This is the largest test problem to perform on this GPU, larger problems wil exceed the maximum amount of memory on the GPU. Test 3 has a total memory of 3.258 GB: this is approximately 27% of the total amount of memory space on the GPU.

8.7.1. Baseline Implementation

The *baseline* implementation appears to work best on the GPU for test problem 1 and test problem 2. Test 3 is larger, so results can be different because of the memory bandwidth. Computation times on the CPU and the GPU for the *baseline* implementation can be found in Table 8.18.

Iterations	Time CPU (sec)	Time GPU (sec)	Acceleration
200	194.454	18.463	10.5
400	388.907	27.625	14.1
600	583.361	36.787	15.9
800	777.815	45.949	16.9
1,000	972.269	55.111	17.6
2,000	1944.537	100.921	19.3
3,000	2916.806	146.731	19.9
4,000	3889.075	192.541	20.2
5,000	4861.344	238.351	20.4
7,000	6805.881	329.971	20.6
10,000	9722.687	467.401	20.8
15,000	14,584.031	696.451	20.9
20,000	19,445.374	925.501	21.0

Table 8.18: CPU and GPU computation times for the MLFMA baseline implementation for test 3.

For the *baseline* implementation one iteration on the GPU is approximately 21 times faster compared to one iteration on the CPU. This maximum speed-up is obtained after approximately 20,000 iterations. The maximum speed-up is obtained when the time to send information is negligibly small. For larger problems much more iterations have to be performed for the communication time to become negligibly small, because of the fact that more data has to be sent to the GPU.
For larger problems, large accelerations are obtained after less iterations, because of the fact that more calculations can be performed in parallel.

8.7.2. Storing the Interpolation Operators as Real Matrices

The computation times on the CPU and the GPU for the *onlyintreal* implementation can be seen in Table 8.19.

Iterations	Time CPU (sec)	Time GPU (sec)
200	322.008	545.825
400	644.016	1083.537
600	966.024	1621.250
800	1288.032	2158.963
1000	1610.040	2696.675

Table 8.19: CPU and GPU computation times for the MLFMA onlyintreal implementation for test 3.

Data has to be sent to the GPU in each iteration, so for larger problems this bottleneck only becomes larger. The GPU and the CPU computation times are again slower compared to the *baseline* implementation.

8.7.3. Performing Matrix Vector Products in Real Arithmetic

The computation times on the CPU and the GPU for the *reals* implementation can be seen in Table 8.20.

Iterations	Time CPU (sec)	Time GPU (sec)
200	470.637	30.490
400	941.274	54.671
600	1411.912	78.849
800	1882.549	103.028
1,000	2353.186	127.207

Table 8.20: CPU and GPU computation times for the MLFMA reals implementation for test 3.

The computation times for both the CPU and the GPU are again slower for the *reals* implementation compared to the *baseline* implementation.

8.7.4. Storing the Interpolation Operators as Small Matrices

The computation times on the CPU and the GPU for the *intsmall* implementation can be seen in Table 8.21.

Iterations	Time CPU (sec)	Time GPU (sec)
200	236.933	243.470
400	473.866	478.789
600	710.798	714.108
800	947.731	949.427
1000	1184.664	1184.746

Table 8.21: CPU and GPU computation times for the MLFMA intsmall implementation for test 3.

The results are the same as for test 1 and 2: the computation times of the *intsmall* implementation on the CPU and the GPU are slower compared to the *baseline* implementation with matrix vector products for the theta and phi interpolation matrices.

Calculation times of the different parts of the *intsmall* implementation are calculated on the CPU and the GPU. The results can be seen in Table 8.22.

Performance	Time CPU (sec)	Time GPU (sec)	Acceleration
Near matrix	$1.286 \cdot 10^{-2}$	$1.249 \cdot 10^{-5}$	1030
Expand waves	$3.189 \cdot 10^{-2}$	$2.304 \cdot 10^{-5}$	1384
Interpolation to level 5	0.144	0.418	0.3
Interpolation to level 4	$9.287 \cdot 10^{-2}$	0.107	0.9
Interpolation to level 3	$6.258 \cdot 10^{-2}$	$2.658 \cdot 10^{-2}$	2.4
Interpolation to level 2	$4.494 \cdot 10^{-2}$	$6.364 \cdot 10^{-3}$	7
Transfer at level 6	0.112	$1.094 \cdot 10^{-5}$	10,238
Transfer at level 5	0.122	$1.014 \cdot 10^{-5}$	12,032
Transfer at level 4	$7.015 \cdot 10^{-2}$	$9.874 \cdot 10^{-6}$	7,107
Transfer at level 3	$4.913 \cdot 10^{-2}$	$9.896 \cdot 10^{-6}$	4,968
Transfer at level 2	$1.943 \cdot 10^{-2}$	$1.016 \cdot 10^{-5}$	1,912
Anterpolation to level 3	$4.418 \cdot 10^{-2}$	$6.618 \cdot 10^{-3}$	7
Anterpolation to level 4	$6.197 \cdot 10^{-2}$	$4.621 \cdot 10^{-2}$	1.3
Anterpolation to level 5	$9.187 \cdot 10^{-2}$	0.116	0.8
Anterpolation to level 6	0.138	0.450	0.3
Incoming waves	$6.886 \cdot 10^{-5}$	$7.450 \cdot 10^{-8}$	984

Table 8.22: CPU and GPU calculation times in different parts of the MLFMA intsmall implementation for test 3.

In Table 8.22 it can be seen that on coarser levels, the GPU is faster for the interpolation matrix vector multiplications compared to finer levels. This is because of the fact that on coarser levels there are more wave directions and less groups, so the GPU can perform more calculations in parallel and less matrix vector multiplications have to be performed.

8.7.5. Storing Matrices in Group Within Wave Structure

The computation times on the CPU and the GPU for the *groupinwave* implementation can be seen in Table 8.23.

Iterations	Time CPU (sec)	Time GPU (sec)
200	171.712	20.137
400	343.423	30.452
600	515.135	40.767
800	686.847	51.083
1000	858.558	61.398

Table 8.23: CPU and GPU computation times for the MLFMA groupinwave implementation for test 3.

The *groupinwave* GPU implementation has again the same computation time as the *baseline* implementation on the GPU. When a different format is used where the GPU knows that the data is structured, the difference between the *baseline* implementation and the *groupinwave* implementation could be considerably larger. In this case the CRS format is used, where the GPU does not know beforehand if data is structured or unstructured. For the CPU, the *groupinwave* implementation is again faster compared to the *baseline* implementation.

8.7.6. Parts of the Baseline Implementation

The *baseline* implementation performs for the largest problem in test 3 still the fastest.

It is important to know which parts of the *baseline* implementation are most time consuming. Therefore the calculation times of the different parts of the *baseline* implementation can be seen in Table 8.24.

Performance	Time CPU (sec)	Time GPU (sec)	Acceleration
Near matrix	$1.337 \cdot 10^{-2}$	$1.272 \cdot 10^{-3}$	11
Expand waves	$3.211 \cdot 10^{-2}$	$5.064 \cdot 10^{-5}$	634
Interpolation to level 5	0.110	$9.283 \cdot 10^{-5}$	1185
Interpolation to level 4	$7.214 \cdot 10^{-2}$	$2.169 \cdot 10^{-3}$	33
Interpolation to level 3	$4.918 \cdot 10^{-2}$	$1.276 \cdot 10^{-4}$	385
Interpolation to level 2	$3.544 \cdot 10^{-2}$	$3.188 \cdot 10^{-3}$	11
Transfer at level 6	0.111	$6.110 \cdot 10^{-5}$	1817
Transfer at level 5	0.122	$6.694 \cdot 10^{-5}$	1823
Transfer at level 4	$7.031 \cdot 10^{-2}$	$6.659 \cdot 10^{-5}$	1056
Transfer at level 3	$4.779 \cdot 10^{-2}$	$1.242 \cdot 10^{-3}$	38
Transfer at level 2	$1.947 \cdot 10^{-2}$	$2.201 \cdot 10^{-3}$	9
Anterpolation to level 3	$3.161 \cdot 10^{-2}$	$2.530 \cdot 10^{-3}$	12
Anterpolation to level 4	$4.437 \cdot 10^{-2}$	$4.291 \cdot 10^{-3}$	10
Anterpolation to level 5	$6.518 \cdot 10^{-2}$	$5.066 \cdot 10^{-3}$	13
Anterpolation to level 6	$9.573 \cdot 10^{-2}$	$5.655 \cdot 10^{-3}$	17
Incoming waves	$2.775 \cdot 10^{-5}$	$1.738 \cdot 10^{-6}$	16

Table 8.24: CPU and GPU calculation times in different parts of the MLFMA baseline implementation for test 3.

The transfer matrix vector multiplications obtain again much acceleration. The accelerations for the interpolation and the anterpolation to different levels are expected to be monotone. For the interpolation matrix vector multiplications this is not the case. The investigation of the reason for this is something to try in further research. Something notable is the fact that the interpolation matrix vector multiplications obtain a much different speed-up compared to the anterpolation matrix vector multiplications. This shows that the structure of the matrices is of great importance for the degree of acceleration on the GPU.

8.8. CPU and GPU Performances

The CPU and GPU performances are compared for thousand iterations for the three test problems. A plot of the results can be found in Figure 8.7.

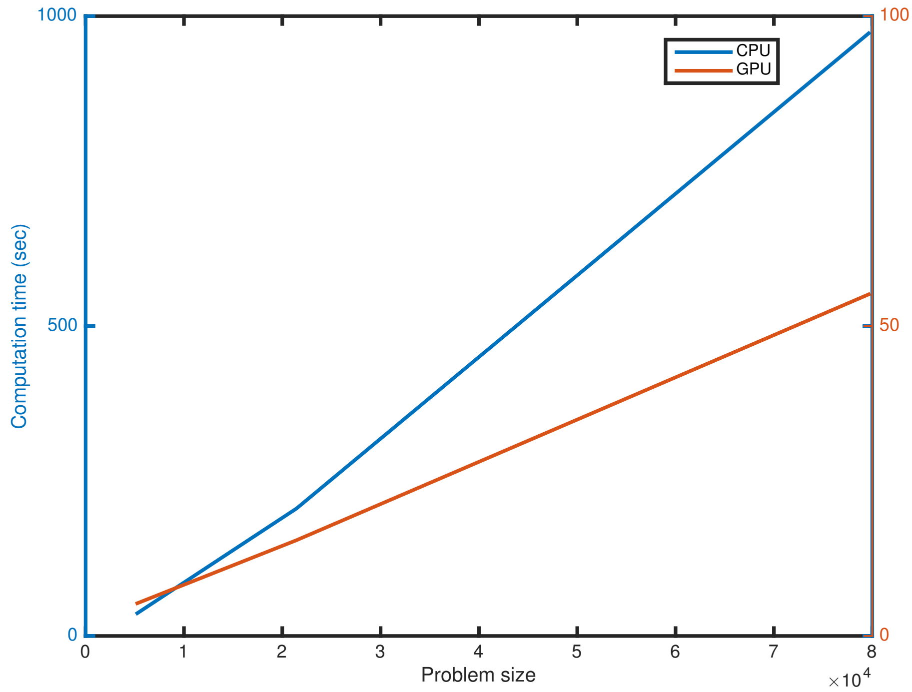

Figure 8.7: CPU and GPU computation times plotted against the problem size for the MLFMA baseline implementation.

It seems like both CPU and GPU computation times show a linear increase, but the CPU computation times are not linear. Calculating the slopes for CPU and GPU computation times in the first part gives 0.010 on the CPU and $6.283 \cdot 10^{-4}$ on the GPU. This gives an average acceleration of 15.9, which corresponds with previous results. The slopes in the second part are 0.013 on the CPU and $6.816 \cdot 10^{-4}$ on the GPU. The average acceleration is 19.1, which corresponds with previous results as well.
The fact that the computation time on the CPU does not linearly increase probably has to do with the memory bandwidth. The memory bandwidth of the CPU becomes full.

8.9. Summary

The three test problems that are performed, give the same results in terms of the performance of different implementations. The best implementation is the *baseline* implementation, where all matrices and vectors are stored with complex numbers and are parallelized over both wave directions and groups. The wave directions are stored within the groups.

The GPU accelerates the test problems 15 to 21 times. Real problems, that are much larger compared to those three test problems, cannot be performed on one single GPU and have to be performed on GPU clusters. This is more difficult because of the fact that the clusters have to communicate with each other.
Something remarkable is the fact that the CPU and the GPU appear to perform the same for structured and unstructured data. This is something that was expected to be different, but could be the case because only small parts of the data is structured. Most likely is that the data appears to be unstructured for the components in both of the test cases (groups within wave directions and wave directions within groups).

In Table 8.25 an overview of the main results is given.

	Memory use			Computation time 100 it			Max speed-up GPU/CPU
	Shako	CPU	GPU	Shako	CPU	GPU	
Test 1	0.0218	0.150	0.150	4.426	3.678	3.107	15.0
Test 2	0.0898	0.739	0.739	21.900	20.586	5.177	17.5
Test 3	0.423	3.258	3.258	89.630	97.227	13.882	21.0

Table 8.25: Memory use, computation time and speed-up for Shako, CPU and GPU.

Shako uses less memory storage compared to the CPU and GPU implementations. The GPU performs on the other hand much faster compared to Shako and accelerates the matrix vector multiplications considerably.

9 Conclusion

The Multi Level Fast Multipole Algorithm (MLFMA) is an acceleration technique for solving discretized integral equations. Graphics Processing Units (GPU's) can potentially accelerate the computations of the MLFMA. Therefore, the research question of this report is: *Which parts of the Multi Level Fast Multipole Algorithm could be calculated faster when those parts are transferred to a GPU, what is the expected speed-up and what are the bottlenecks of using GPU's for those parts?*

There are not many specific publications about the application of the MLFMA on GPU's, but publications about the Method of Moment or Fast Multipole Method suggest different parts of the Multi Level Fast Multipole Algorithm for acceleration on the GPU. Examples are constructing matrices on the GPU, performing matrix vector multiplications on the GPU, using pinned memory or using the 'compute on-the-fly' strategy. The focus in this thesis is on the matrix vector multiplications, because NLR knows from practice that this is the most time consuming part of the MLFMA.

The GPU accelerates three test problems that are performed for the MLFMA 15 to 21 times. The goal of the project, to accelerate the Multi Level Fast Multipole Algorithm with the help of a GPU on a given test problem at least 5 times, is achieved. For larger problems GPU clusters have to be used, which is more difficult, because of the fact that the GPU's have to communicate with each other to share data.

A bottleneck of the GPU is the time needed to send data to the GPU and back to the CPU. Sending a large amount of data makes the GPU algorithm slow. Therefore, it is important to perform several computations in parallel on a GPU, in order to let the time to send information to the GPU and back to the CPU become negligibly small. Another bottleneck of the GPU is the limited memory storage on the GPU. It is important to store as little data as possible. Writing device code directly appears to be difficult with limited knowledge about GPU's. cuBLAS and cuSPARSE routines are standard routines: they divide data as efficiently as possible over the threads and blocks and are therefore easier to use.

In the model problem, where the Conjugate Gradient method is used to solve the Poisson equation, a sparse matrix vector multiplication and different cuBLAS routines are performed on the GPU. The acceleration on the GPU, compared to one single core on the CPU, is approximately 11.5 for both computations in single and in double precision. The speed-up of the GPU compared to the CPU is the same for single and double precision computations, because of the fact that on the GPU the memory bandwidth is fully used and therefore the acceleration of the floating point operations in single precision is negligibly small. This means that only the acceleration of the memory (information is ready for computations sooner) is noticed on the GPU. On the CPU, the memory bandwidth is not fully used and so computations take approximately half of the time in single precision. This means that both CPU and GPU computations are approximately twice as fast for single precision computations compared to double precision computations, so the speed-up of the GPU compared to the CPU is approximately the same.

Three test problems for the MLFMA are used to compare different implementation methods for sparse matrix vector multiplications of the MLFMA. A *baseline* implementation is written in a way which is expected to have the lowest computation time. The other implementation methods differ on one single part of the *baseline* implementation in order to compare the results fairly, for example by storing the interpolation operators as real matrices, storing all matrices as real matrices, storing the interpolation operators as smaller matrices or storing the groups within the wave directions.

When the interpolation operators are stored as real matrices, in each iteration a vector has to be sent back to the CPU to split a complex vector in a real and an imaginary part. This makes the implementation significantly slower, because of the fact that sending information back and forth from the CPU to the GPU is the bottleneck of the GPU. Storing all matrices and vectors as reals appears to be less efficient compared to the *baseline* implementation, because of the fact that the complex cuSPARSE routine operates efficiently compared to four real cuSPARSE routines.

When the interpolation operators are stored as smaller matrices, the GPU can perform fewer computations in parallel and therefore performs slower compared to the *baseline* implementation where the large interpolation operators are stored.

Storing the groups within the wave directions results in the same computation time on the GPU compared to the *baseline* implementation on the GPU, where wave directions are stored within groups. It appears that the ordering of the wave directions within the groups does not perform faster compared to the ordering of groups within the wave directions. Therefore, it is expected that there will be a better ordering.

The *baseline* implementation is memory bound: the largest problem that can be solved on a single GPU has about 100,000 degrees of freedom, but it does give a speed-up of 21 compared to computations on a CPU with a single core.

The GPU prefers to work with as large a data set as possible. When smaller matrices are stored to save memory, the GPU's performance is slower compared to storing large matrices, because various small computations have to be performed.

When utilizing a GPU, three things are very important: performing as many computations in parallel as possible (this makes the GPU perform as efficiently as possible), making use of standard efficient cuBLAS and cuSPARSE routines when limited knowledge about GPU's is available and sending as little data as possible back and forth (this negatively affects the computation time on the GPU).

10

Recommendations for Future Work

In this thesis the focus is on the matrix vector multiplications, because of the fact that this is the most time consuming part of the Multi Level Fast Multipole Algorithm. This was a first step in the acceleration of the MLFMA: there are various additional options to investigate in future research.

Suggestions for future research are stated below.

- Comparison of different compilation options for the matrix vector multiplications of the MLFMA on the GPU.
- Comparison of the Ellpack-Itpack format [37] for sparse matrix vector multiplications of the MLFMA to the CRS format used in this thesis.
- Investigating renumbering techniques.
- Trying to find a balance between memory usage and speed-up for the matrix vector multiplications of the MLFMA.
- Making use of a preconditioner which is efficient on the GPU.
- Making use of GPU clusters to work with real MLFMA problems of 30 millions of unknowns.
- Performing the construction of the matrices and the preconditioner of the MLFMA in parallel on the GPU.
- Making use of the 'compute on-the-fly' strategy [26].

The Kepler K40 has various compilation options that have possibly impact on the computation time. Future research could focus on finding the best compilation options for the MLFMA implementation on a GPU. The options used in this thesis are explained in Appendix C.

If cuSPARSE is used for the matrix vector multiplications of the MLFMA, the matrices have to be stored in a specific cuSPARSE format. In this thesis the matrices are stored in the Compressed Row Storage (CRS) format, which stores the values of the nonzeros, the column indices of the nonzeros and the row offsets. The Ellpack-Itpack format [37] stores two matrices: one with the values of the nonzeros and one with the column indices of the nonzeros. The number of the columns is the number of entries per row. This format uses less memory on the GPU compared to the CRS format, under the assumption that the rows all have approximately the same length.

Storing the groups within the wave directions is just as fast as storing the wave directions within the groups. Therefore, it is expected that there will be a better ordering, which can be found with the help of renumbering techniques.

The result of this thesis is a speed-up of 21 on the GPU compared to the CPU, for a problem size of approximately 86,000 unknowns. This implementation method on the GPU uses 7.7 times more memory compared to the CPU implementation in Shako. Real MLFMA problems have numbers of unknowns of up to 30 million. This means that problems can arise regarding the memory.
In this thesis, an implementation with smaller theta and phi interpolation matrices is written. This saves memory, but unfortunately this implementation of the small matrices is much slower because of the fact that the GPU can perform fewer calculations in parallel. Another option is to store the transfer operators not as matrices, but as function of the path between two groups. This means that device code has to be written directly. The conjugate transpose of the outgoing waves is computed on the CPU in this thesis. In newer versions of the GPU the conjugate transpose can probably be computed directly during the matrix vector multiplication with cuSPARSE: this saves memory on the GPU.

The MLFMA cannot be performed without the use of an efficient preconditioner on the GPU. The current preconditioner in Shako (ILU) is not suitable for a GPU because of recursions.

Real MLFMA problems have numbers of unknowns of up to 30 million, which is a much larger problem compared to the test problems in this thesis. This means that GPU clusters have to be used. This is complicated, because of the fact that the GPU's have to communicate with each other.

The construction of the matrices and the preconditioner of the MLFMA can be performed in parallel on the GPU as well. Each thread can compute one matrix entry.

Sending data to the GPU and back to the CPU is the main bottleneck of the GPU. The 'compute on-the-fly' strategy [26] computes matrices whenever they are needed for computations, so this data does not have to be sent to the GPU. For the matrix vector multiplications on the GPU this is possibly a good idea for the interpolation matrices, because of the fact that they are fairly quick to build. The near matrix and the transfer matrices contain calculations that are too complicated for using the 'compute on-the-fly' strategy: this will take too much time.

There are many different ways of accelerating the MLFMA on GPU's. The suggestions for future research stated above came to mind while writing this thesis.

Bibliography

[1] H. van der Ven and H. Schippers, 'Mathematical definition document for the multi-level fast-multipole method', NLR-TR-2010-529, 2015.

[2] S.A. Hack, 'Spherical harmonics based aggregation in the multilevel fast multipole algorithm', MSc Thesis, NLR-TR-2015-218, also available at http://essay.utwente.nl/67165/.

[3] Ö. Ergül and L. Gürel, 'The Multilevel Fast Multipole Algorithm (MLFMA) for Solving Large-Scale Computational Electromagnetics Problems', 2014.

[4] A. Heldring, 'Full-Wave Analysis of Electrically Large Reflector Antennas', 2002.

[5] J. Jin, C. Lu and W. Chew, 'On the formulation of hybrid finite-element and boundary-integral method for 3D scattering', IEEE transactions on antennas and propagation, 1998.

[6] W. Gibson, 'The Method of Moments in Electromagnetics', 2008.

[7] K. Sertel, 'Multilevel Fast Multipole Method for Modeling Permeable Structures using Conformal Finite Elements', 2003.

[8] J.M. Song and W.C. Chew, 'Multilevel Fast Multipole Algorithm for Solving Combined Field Integral Equation of Electromagnetic Scattering', 1995.

[9] L. Zhanhea, H. Peilina, G. Xub, L. Yinga and J. Jinzua, 'Multi-frequency RCS Reduction Characteristics of Shape Stealth with MLFMA with Improved MMN', Chinese Journal of Aeronautics, vol. 23, no. 3, (2010).

[10] H. van der Ven and H. Schippers, 'High Range Resolution Profiles for a Civilian Aircraft Inlet', NLR-TR-2010-527, 2011.

[11] L. Li, J. He, Z. Liu and L. Carin, 'MLFMA Analysis of Scattering from Multiple Targets In the Presence of a Half Space', 2003.

[12] E. Darve, 'The Fast Multipole Method: Numerical Implementation', 1999.

[13] J. Guan, S. Yan and J.M. Jin, 'An Accurate and Efficient Finite Element-Boundary Integral Method With GPU Acceleration for 3-D Electromagnetic Analysis', IEEE transactions on antennas and propagation, vol. 62, no. 12, (2014).

[14] Ö. Tayfun, J. Malcolm and M. Yuriy, 'Multi-Core CPU and GPU Accelerated FMM-FFT Solver for Antenna Co-Site Interference Analysis on Large Platforms', AP-S, 2014.

[15] Q. Nguyen, V. Dang, O. Kilie and E. El-Araby, 'Parallelizing Fast Multipole Method for Large-Scale Electromagnetic Problems Using GPU Clusters', IEEE antennas and wireless propagation letters, vol. 12, (2013).

[16] Q. Hu, N. Gumerov and R. Duraiswami, 'Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures', University of Maryland.

[17] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros and R. Vudue, 'Optimizing and Tuning the Fast Multipole Method for State-of-the-Art Multicore Architectures', IEEE, 2010.

[18] J. Guan, S. Yan and J. Jin, 'An OpenMP-CUDA Implementation of Multilevel Fast Multipole Algorithm for Electromagnetic Simulation on Multi-GPU Computing Systems', IEEE transactions on antennas and propagation, vol. 61, no. 7, (2013).

[19] R. Coifman, V. Rokhlin and S. Wandzura 'The Fast Multipole Method for the Wave Equation: A pedestrian Prescription', IEEE Antennas and Propagation Magazine, vol. 35, no. 3, (1993).

[20] K. D'Ambrosio and R. Pirich 'Parallel Computation Methods for Enhanced MOM and MLFMM Performance', IEEE, 2009.

[21] M. Cwikla 'Low-Frequency MLFMA on Graphics Processors', IEEE, 2010.

[22] D.M. Hailu 'Hybrid Spectral-Domain Ray Tracing Method for Fast Analysis of Millimeter-Wave and Terahertz-Integrated Antennas', IEEE, 2011.

[23] B. Kolundzija, M. Tasic, D. Olcan, D. Zoric and S. Stevanetic 'Full-Wave Analysis of Electrically Large Structures on Desktop PCs', IEEE, 2011.

[24] E. Yunis, R. Yokota and A. Ahmadia 'Scalable Force Directed Graph Layout Algorithms Using Fast Multipole Methods', IEEE, 2012.

[25] V. Dang, Q. Nguyen, O. Kilic and E. El-Araby 'Single Level Fast Multipole Method on GPU cluster for Electromagnetic Problems', IEEE, 2013.

[26] J. Guan, S. Yan and JM. Jin 'A GPU-Accelerated Integral-Equation Solution for Large-Scale Electromagnetic Problems', IEEE, 2014.

[27] D. Faircloth, T. Killian, M. Horn, M. Shafieipour, I. Jeffrey, J. Aronsson and V. Okhmatovski 'Fast Direct Higher-Order Solution of Complex Large Scale Electromagnetic Scattering Problems via Locally Corrected Nystrom Discretization of CFIE', IEEE, 2014.

[28] N. Tran, T. Phan and O. Kilic 'Analysis of Micro-Doppler Signature Due To Indoor Human Motion Using Multilevel Fast Multipole Algorithm On GPU Cluster', IEEE, 2015.

[29] J. Guan 'OpenMP-CUDA Implementation of the Moment Method and Multilevel Fast Multipole Algorithm on Multi-GPU Computing Systems', Thesis at University of Illinois, 2013.

[30] W. C. Chew, J-M. Jin, E. Michielssen and J. Song 'Fast and Efficient Algorithms in Computational Electromagnetics', Artech House, 2001.

[31] N. Gupta, http://cuda-programming.blogspot.nl/2013/01/thread-and-block-heuristics-in-cuda.html.

[32] M. Lopez-Portugues, J. A. Lopez-Fernandez, J. Menedez-Canel, A. Rodriguez-Campa and J. Ranilla 'Acoustic scattering solver based on single level FMM for multi-GPU systems', Elsevier, 2011.

[33] M. Lopez-Portugues, J. A. Lopez-Fernandez, A. Rodriguez-Campa and J. Ranilla 'A GPGPU solution of the FMM near interactions for acoustic scattering problems', Springer, 2011.

[34] R. Erkamp, http://www.cs.vu.nl/~rob/masters-theses/Ronald-Erkamp.pdf.

[35] 'Tesla K40 GPU Active Accelerator', https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf

[36] 'NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110' https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[37] 'Sparse Matrix Representations & Iterative Solvers', Nathan Bell http://www.bu.edu/pasi/files/2011/01/NathanBell1-10-1000.pdf

A

List of Symbols

Symbol	Explanation
A_l^{l-1}	Agglomeration matrix.
b	Incoming radar wave.
c_m	Group center of degree of freedom r.
$c_{m'}$	Group center of degree of freedom r'.
d_m	Distance between r and c_m.
D	Distance between r' and c_m.
f_n	Basis function.
g_k	Lagrange interpolated function.
G_l	Number of groups on level l.
$h_l^{(1)}$	Spherical Hankel function of the first kind.
I_l^{l-1}	Lagrange interpolation matrix.
j_l	Spherical Bessel function of the first kind.
J	Vector that needs to be determined.
k	Wave number.
$\hat{k}$	Wave direction.
K_l	Number of integration points used in the k-space integral at level l.
$\mathscr{L}'$	Near field interactions between different scatterers.
M	Number of basis functions in one group.
N	Total number of basis functions.
P_l	Legendre polynomial.
r, r'	Degree of freedom.
$r_{mm'}$	Distance between c_m and $c_{m'}$.
S	Surface of a bounded scatterer.
T	Translation matrix.
V	Incoming/outgoing wave information.
w	Weight.
x	Unknown currents on the airplane.
θ, ϕ	Spherical coordinates.

B

Survey Existing Literature MLFMA and GPU's

B.1. Paper 1: An Accurate and Efficient Finite Element-Boundary Integral Method With GPU Acceleration for 3-D Electromagnetic Analysis

Following the reasoning presented by [13], the following notes can be made.

- The accuracy of the FE-BI method is improved using Rao-Wilton-Glisson or Buffa-Christiansen functions as testing functions.
- To accelerate the convergence of the iterative solution in the FE-bi(CFIE) method, the absorbing boundary condition (ABC)-based preconditioner is used.
- To further improve the efficiency of the total computation, the MLFMA is applied to the iterative solution.
- Using multi-GPU computing systems, the assembly of the near field matrices could be calculated on multiple GPU's, the calculations of the inner and outer iterations could be accelerated and the scattered fields could be evaluated.
- The radiation patterns and receiving patterns on each level can be calculated in parallel.
- To obtain a maximum parallel efficiency on all levels, the implementation strategy 'one thread per spectrum sampling' and 'one/several block(s) per group' is adopted, so that groups and their far field patterns partitioned simultaneously.
- An inner iteration scheme will be developed, which can be parallelized efficiently on GPU's.
- BiCGSTAB with the ILU0, the AI and the Jacobi preconditioners are employed to solve $([A])+[M])\{y\} = \{u\}$, where y the unknown.
    1. The ILU0 preconditioner requires roughly the same amount of memory as the input matrix $([A]) + [M])$, but forward and back substitutions make it inefficient for parallel processing on GPU's.
    2. The AI preconditioner is an incomplete approximation of the inverse of the input matrix based on the minimization of the Frobenius norm. In contrast to the ILU-based preconditioners, the AI preconditioner is applied using MVPS, which makes it very suitable for GPU parallelization.
    3. The Jacobi preconditioner is very cheap to generate and apply, but the iterative convergence is slow when the input matrix is ill-conditioned.
- Since the basis functions can be regarded as the current sources from which the scattered fields are generated, a one dimensional grid of threads is allocated on the GPU, and each thread is related to one basis function.

- For the scattered field at a specific observation angle, each GPU thread calculates the corresponding scattered field in parallel, and one CPU thread superposes all the calculated scattered fields in series to avoid write conflicts on the GPU.
- To further accelerate the scattered field evaluation, the OpenMP parallel technique is employed to generate multiple CPU threads, and each CPU thread manages one GPU device to calculate a portion of the scattered fields.
- Two methods for solving $([A]) + [M])\{y\} = \{u\}$:
  1. Direct method solve: $([A]) + [M])$ is first decomposed into a lower and an upper triangular matrix, and solved by the forward and back substitutions.
  2. Iterative method solve: (ILU0, AI or Jacobi) preconditioned BiCGSTAB method with a targeted relative residual error of 10^{-3} is applied for the solve.
- The direct method is expensive in the construction phase and cheap in the solution phase. However, the direct method will require a larger storage and a higher computational cost when the problems become large, and the solution is very difficult to parallelize on GPU's. Therefore only the iterative methods will be considered for the GPU calculation.
- Conclusions of the three preconditioners:
  - AI preconditioned iterative method is efficient in the solution phase.
  - Jacobi is cheapest in the construction phase.
  - ILU0 is not well suited for the GPU parallelization, because it requires forward and backward substitutions.
  - More speedup using AI than Jacobi in the human body and roar object examples, but it also needs more memory.
- Modified ABC-based preconditioner is better compared to the original preconditioner, because it is symmetric and more effective.
- For further speedup the computation, a GPU-accelerated FE-BI algorithm was developed.
- The FE-BI(CFIE) method is accurate, robust and efficient for practical applications.

B.2. Paper 2: Multi-Core CPU and GPU Accelerated FMM-FFT Solver for Antenna Co-Site Interference Analysis on Large Platforms

Following the reasoning presented by [14], the following notes can be made. Important information: This paper used a FMM-FFT solver, which is different from ours.

- The single-level Toeplitz grouping strategy is much easier to parallelize and results in much better scaling compared to the multi-level FMM approach.
- The BiCGstab method can be accelerated on a multi-core CPU. The matrix fill can be accelerated the most: up to 18 times. The solver can be accelerated up to 14 times.
- The BiCGstab method can also be accelerated on a GPU with respect to a single thread CPU. The acceleration of the matrix fill is up to 116,8 times and the solver 7 times.

B.3. Paper 3: Parallelizing Fast Multipole Method for Large-Scale Electromagnetic Problems Using GPU Clusters

Following the reasoning presented by [15], the following notes can be made.

- Uniformly distributing the total number of groups M among the N computing nodes. Each node holds the same workload.

- Rows of Z_{near} are assigned to the computing nodes following the group-based distribution. Each node has approximately an equal number of near interactions, and they are calculated without any communication with the other nodes.
- At a given node, the computation of each element Z_{mn} is performed by one thread enabling all row computations to be allocated to one CUDA block.
  - The Z_{near} matrix is partitioned among the computing nodes, such that each CUDA thread per node is assigned to compute one element Z_{mn}, and threads are grouped into blocks that are responsible for computing the elements of one row of Z_{near} for that node.
  - This results in a total number of CUDA blocks (i.e. rows in the matrix) per node equal to $N_{group}M_{node}$.
- Radiation/receive function calculations: Data is distributed among nodes and the calculations per node are performed such that M_{node} groups are allocated for each node. Each group per node requires K evaluations of the radiation/receiving function. The threads are grouped into blocks such that each block of threads performs N_{group} radiation calculations at a given direction, resulting in a number of blocks per node equal to $M_{node}K$.
- The far field translation is the same as near interactions: each CUDA block is assigned to compute one row of the T_l matrix at a given direction, which results in a total number of thread blocks per node equal to $M_{node}K$.
- For the fast matrix-vector multiplication is the group-based distribution scheme used: the inter-node communication is required only at two steps:
  1. At the beginning of the matrix-vector multiplication to exchange the estimated values for the unknowns among the nodes.
  2. After the aggregation step and before performing the translation step in order to update all nodes with the current aggregation results.
- Before entering the linear system solution, each node calculates its own portion of the radiation/receive functions and the translation matrix.
- During the iterative solution, each node also calculates the estimates values of the unknowns. It is important to note that the distribution of the unknowns among the computing nodes follows the same group-based distribution of the radiate/receive functions.
- In the aggregation step, each unknown is multiplied by its corresponding radiation function and is accumulated within a given group in order to calculate the total field for that group.
- After aggregation, an all-to-all communication is employed by each node to broadcast the aggregated fields to all other nodes.
- For larger problems, it should not be forgotten that the GPU memory has a limit. A solution for this problem is to utilize both CPU and GPU memory spaces.

B.4. Paper 4: Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures

Following the reasoning presented by [16], the following notes can be made.

- Steps for parallelization on GPU's:
  - Determine of the Morton index for each particle.
  - Construction of occupancy histogram and bin sorting.
  - Parallel scan and global particle ranking.
  - Final filtering.
  - Final bin sorting.

- Determining the interacting source boxes in the neighborhood of the receiver boxes.
- Different stages of the FMM have very different efficiency when parallelized on the GPU:
  - Lowest efficiency (due to limited GPU local memory) is for translations.
  - Computation of $A^{near}J$ on GPU is very efficient.
  - Anything having to do with particles is very efficient on the GPU, and translations are relatively efficient on CPU.

B.5. Paper 5: Optimizing and Tuning the Fast Multipole Method for State-of-the-Art multi core Architectures

Following the reasoning presented by [17], the following notes can be made. In this paper a lot of different GPU's are discussed. This is not very relevant for this thesis.

- Taking square roots and divide operations are very slow on GPU's.
- Mistakes
  - Relied on bulk-synchronous parallelism: some working sets did not fit in cache. Data flow and work-queue approaches may mitigate this issue.
  - Optimal algorithmic parameters will vary not only with architecture, but also optimization and scale of parallelism.
  - Manual SIMD transformations and their interaction with data layout was a significant performance win, and should be a priority for new compile and/or programming model efforts.

B.6. Paper 6: An OpenMP-CUDA Implementation of Multilevel Fast Multipole Algorithm for Electromagnetic Simulation on Multi-GPU Computing Systems

Following the reasoning presented by [18], the following notes can be made.

- On finer levels, groups at the same level are partitioned into different processors and each processor gets approximately the same number of groups.
- On coarser levels, far-field patterns (FFPs) are replicated for every processor. When the number of processors increases, this parallelization strategy is not effective around the transition levels where neither the number of groups, nor the number of FFPs is large enough to achieve a good parallel efficiency.
- Examples of 'one thread per observer' are: parent group in the aggregation phase, child group in the disaggregation phase and destination group in translation phase. This is getting a low parallel efficiency when the number of groups decreases at coarser levels.
- Curvilinear Rao-Wilton-Glisson (CRWG) functions are used as the basis functions.
- The intensive parts are:
  - Iterative solution
  - Radiation patterns of the basis functions V_s
  - Receiving patterns of the testing functions V_f
  - Translator T
  - Assembly of near-field system matrix Z_{near}
- For the far field interaction: parallelly computing the radiation patterns and receiving patterns of the groups in the aggregation, translation and disaggregation phases.

- In the aggregation phase, the size of thread block in the algorithm is set as the size of the spectrum at the finest level for the optimal use of hardware resources.
- Global memory strategy: radiation and receiving patterns should be calculated and stored at all levels on a single GPU. This avoids data transfer between host and device and is therefore fast. But the size of the global memory will limit the size of the problems that can be solved.
- Pinned memory strategy: calculates radiation and receiving patterns on multiple GPU's and stores the results in the pinned memory on the host. This could solve larger problems, because the size of the pinned memory is much larger. But data communications between host and device become unavoidable.

C

CPU and GPU compilations

C.1. Kepler K40

A Fortran implementation of an algorithm on the GPU can be compiled with Pgfortran and the use of the following flags:

```
pgfortran -Mcuda=madconst Sort.f90 MLFMA.F90 -o MLFMAGPU -lcusparse -lcublas

-Dgpu,
```

where pgfortran is the call for the compiler, which is the compiler of the hardware manufacturer. -Mcuda=madconst activates the CUDA extension, Sort.f90 is a file which is used by the main file, MLFMA.F90 is the file with the code for the algorithm on the GPU and -o MLFMAGPU specifies the name for the output file. -lcusparse tells the compiler to use the cuSPARSE library and -cublas tells the compiler to use the cuBLAS library. A C-preprocessor is a single implementation where the GPU part is activated using -Dgpu.

C.2. Intel E5-2640

A Fortran implementation of an algorithm on the CPU can be compiled with Ifort and the use of the following flags:

```
ifort Sort.f90 MLFMA.F90 -o MLFMACPU -Dmkl -lmkl_intel_lp64 -lmkl_core

-mkl_sequential -lpthread -lm -ldl,
```

where ifort is the call for the compiler, which is the compiler of the hardware manufacturer. Sort.f90 is a file which is used by the main file, MLFMA.F90 is the file with the code for the algorithm and -o MLFMACPU specifies the name for the output file. A C-preprocessor is a single implementation where the CPU part, including the Intel MKL library, is activated using -Dmkl. The Intel MKL library implements the BLAS and Sparse matrix routines. The rest of the options are advised by intel for a Fortran program that runs sequentially on a CPU.

D

Fortran Codes

D.1. Model Problem in Device Code

```
! Device code
module PoissonGPU
 contains
 attributes(global) subroutine poisson_kernel( it_max, c, unew, &
        pnew, rnew, A, uexact, ufinal, itfinal)
 implicit none
 integer, value :: it_max, c
 real, parameter :: tolerance = 0.000001D+00
 real :: unew(c), pnew(c), rnew(c), A(c,c), uexact(c), ufinal(c), &
        uj, pj, rj, udiffj, total, midj
 real :: Tj, alppj, alpTj, betapj, udifffj, dj, rr, pT, alp, beta, &
        rn, er, error, temp, rnewj, pnewj, unewj
 integer :: i, j, it, itfinal, tx, k

 real, shared :: aalp, bbeta, errorr, pp(c)

 logical converged

 tx = threadIdx%x
 j = blockDim%x * (blockIdx%x -1) + threadIdx%x

 if(j<=c) then
  unewj = unew(j)
  pnewj = pnew(j)
  rnewj = rnew(j)
 end if

 do it = 1, it_max

  if (j<=c) then
   uj = unewj
   pj = pnewj
   rj = rnewj
  end if

  call syncthreads()
```

```
if (tx<=c) then
 pp(tx) = pj
end if

call syncthreads()

Tj = 0.0D+00

if (j<=c) then
 do i = 1,c
  Tj = Tj + A(j,i)*pp(i)
 end do
end if

if (j<=c) then
 dj = rj*rj
end if

call syncthreads()

if (j<=c) then
 pp(j) = dj
end if

call syncthreads()

rr = 0.0D00

if (j==1) then
 do i = 1,c
  rr = rr + pp(i)
 end do
end if

if (j<=c) then
 dj = pj*Tj
end if

call syncthreads()

if (j<=c) then
 pp(j) = dj
end if

call syncthreads()

pT = 0.0D00

if (j==1) then
 do i = 1,c
  pT = pT + pp(i)
 end do
end if

if (j==1) then
 alp = rr/pT
```

```
end if

if (j==1) then
 aalp = alp
end if

call syncthreads()

if (j<=c) then
 alppj = aalp*pj
end if

if (j<=c) then
 unewj = uj + alppj
end if

if (j<=c) then
 alpTj = aalp * Tj
end if

if (j<=c) then
 rnewj = rj - alpTj
end if

if (j<=c) then
 dj = rnewj*rnewj
end if

call syncthreads()

if (j<=c) then
 pp(j) = dj
end if

call syncthreads()

rn = 0.0D00

if (j==1) then
 do i = 1,c
  rn = rn + pp(i)
 end do
end if

if (j==1) then
 beta = rn/rr
end if

if (j==1) then
 bbeta = beta
end if

call syncthreads()

if (j<=c) then
 betapj = bbeta * pj
```

```
  end if

  if (j<=c) then
   ufinal(j) = unewj
   pnewj = rnewj + betapj
  end if

  itfinal = it

  call syncthreads()

 end do

 end subroutine poisson_kernel

end module PoissonGPU

! Host code
program PoissonCPU
 use PoissonGPU
 use cudafor
 implicit none

 integer, parameter :: nx = 10
 integer, parameter :: ny = 10
 integer, parameter :: n = nx * ny
 integer, parameter :: c = (nx-2) * (ny-2)

 real(kind=8) :: start_time, stop_time
 real(kind=8) :: time
 logical converged
 real :: uexact(c), unew(c), ufinal(c), pnew(c), rnew(c), A(c,c)

 integer, parameter :: it_max = 500

 real, device, allocatable, dimension(:,:) :: Adev, unewdev, &
        pnewdev, rnewdev, uexactdev, ufinaldev
 integer, device, allocatable :: itdev
 type(dim3) :: dimGrid, dimBlock
 integer :: dimShMem

! Matrix A and vectors unew, pnew, rnew and uexact are calculated here.

 allocate(Adev(c,c), unewdev(c), pnewdev(c), rnewdev(c), &
   uexactdev(c), ufinaldev(c),itdev)

! Set block size and grid size for GPU
 dimBlock = dim3(256,1,1)
 dimGrid = dim3(1,1,1)
 dimShMem=10000

! Send matrix and vectors to GPU
 Adev = A(1:c,1:c)
 unewdev = unew(1:c)
 pnewdev = pnew(1:c)
 rnewdev = rnew(1:c)
```

```
 uexactdev = uexact(1:c)

! Call the device code
 call poisson_kernel<<<dimGrid,dimBlock,dimShMem>>>(it_max, c, &
        unewdev, pnewdev, rnewdev, Adev, uexactdev, ufinaldev, itdev)

! Send vector back to the CPU
 unew(1:c) = unewdev

 deallocate(Adev, unewdev, pnewdev, rnewdev, uexactdev, ufinaldev, &
        itdev)

end program PoissonCPU
```

D.2. Model Problem cuSPARSE and cuBLAS

```
Program Poisson
use cudafor
use cublas
use cusparse
implicit none
 integer, parameter :: nx = 10
 integer, parameter :: ny = 10
 integer, parameter :: n = nx * ny
 integer, parameter :: c = (nx-2) * (ny-2)

 real(kind=8) :: start_time, stop_time
 logical converged
 integer :: it, NNZ, JA(c+1)
 integer, parameter :: it_max = 500
 real, parameter :: tolerance = 0.000001D+00
 real :: rr, pT, rn, unew(c), pnew(c), rnew(c), uexact(c)
 integer, dimension(:), allocatable :: jndx
 real, dimension(:), allocatable :: val
 integer, device,allocatable,dimension(:) :: JA_d, jndx_d
 real, device :: r_d(c), p_d(c), u_d(c), Ax_d(c), betap_d(c)
 real, device, allocatable, dimension(:) :: val_d
 type(cusparseHandle) :: h
 type(cublasHandle) :: h1
 type(cusparseMatDescr) :: descrA
 integer :: status

! Matrix A is stored in the CRS format and vectors rnew, pnew,
! unew are calculated here.

 allocate(val_d(NNZ), jndx_d(NNZ))

! Send information about matrix and vectors.
 r_d = rnew(1:c)
 p_d = pnew(1:c)
 u_d = unew(1:c)
 JA_d = JA(1:c+1)
 val_d = val(1:NNZ)
 jndx_d = jndx(1:NNZ)
 uexact_d = uexact(1:c)
```

```
 status = cusparseCreate(h)
 istat = cublasCreate(h1)
 status = cusparseCreateMatDescr(descrA)
 status = cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL)
 status = cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE)

 do it = 1, it_max
! Call in each iteration the cuSPARSE routine.
  status = cusparseDcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, c, &
        c, NNZ, 1.0, descrA, val_d, JA_d, jndx_d, p_d, 0.0, Ax_d)

  istat = cublasDdot_v2(h1,c, r_d, 1, r_d, 1,rr)

  istat = cublasDdot_v2(h1,c, p_d, 1, Ax_d, 1, pT)

  alp = rr/pT

  istat = cublasDaxpy_v2(h1, c, alp, p_d, 1, u_d, 1)

  istat = cublasDscal_v2(h1,c, alp, Ax_d, 1)

  istat = cublasDaxpy_v2(h1, c, b, Ax_d, 1, r_d, 1)

  istat = cublasDdot_v2(h1,c,r_d,1,r_d,1, rn)

  beta = rn/rr

  istat = cublasDcopy_v2(h1, c, p_d, 1, betap_d,1)

  istat =  cublasDcopy_v2(h1, c, r_d, 1, p_d,1)

  istat = cublasDaxpy_v2(h1, c, beta, betap_d, 1, p_d, 1)
 end do

 status = cusparseDestroy(h)

! Send vector back to the CPU.
 unew(1:c) = u_d

End Program Poisson
```

D.3. MLFMA Baseline Implementation

```
Program MLFMA
use Sort
use cudafor
use cublas
use cusparse
implicit none

! Matrices are read from files and saved in the CRS format here.

 do i = 1,nearmatrix%ncol
  x(i) = cmplx(0.0,0.0)
 end do
```

```
x(333) = cmplx(1.0,0.0)

alpha = cmplx(1.0,0.0)
beta = cmplx(0.0,0.0)

! Send information about the matrices to the GPU.
nearmatrix_d%val = nearmatrix%val(1:nearmatrix%nnz)
nearmatrix_d%JA = nearmatrix%JA(1:nearmatrix%nrow+1)
nearmatrix_d%jndx = nearmatrix%jndx(1:nearmatrix%nnz)
x_d = x(1:nearmatrix%ncol)

V1_d%val = V1%val(1:V1%nnz)
V1_d%val2 = V1%val2(1:V1%nnz)
V1_d%val3 = V1%val3(1:V1%nnz)
V1_d%JA = V1%JA(1:V1%nrow+1)
V1_d%jndx = V1%jndx(1:V1%nnz)

V2_d%val = V2%val(1:V2%nnz)
V2_d%val2 = V2%val2(1:V2%nnz)
V2_d%val3 = V2%val3(1:V2%nnz)
V2_d%JA = V2%JA(1:V2%nrow+1)
V2_d%jndx = V2%jndx(1:V2%nnz)

do l = lmax-1, lmin, -1
 Itheta1_d(l)%val = Itheta1(l)%val(1:Itheta1(l)%nnz)
 Itheta1_d(l)%JA = Itheta1(l)%JA(1:Itheta1(l)%nrow+1)
 Itheta1_d(l)%jndx = Itheta1(l)%jndx(1:Itheta1(l)%nnz)

 Itheta2_d(l)%val = Itheta2(l)%val(1:Itheta2(l)%nnz)
 Itheta2_d(l)%JA = Itheta2(l)%JA(1:Itheta2(l)%nrow+1)
 Itheta2_d(l)%jndx = Itheta2(l)%jndx(1:Itheta2(l)%nnz)

 Iphi1_d(l)%val = Iphi1(l)%val(1:Iphi1(l)%nnz)
 Iphi1_d(l)%JA = Iphi1(l)%JA(1:Iphi1(l)%nrow+1)
 Iphi1_d(l)%jndx = Iphi1(l)%jndx(1:Iphi1(l)%nnz)

 Iphi2_d(l)%val = Iphi2(l)%val(1:Iphi2(l)%nnz)
 Iphi2_d(l)%JA = Iphi2(l)%JA(1:Iphi2(l)%nrow+1)
 Iphi2_d(l)%jndx = Iphi2(l)%jndx(1:Iphi2(l)%nnz)

 A1_d(l)%val = A1(l)%val(1:A1(l)%nnz)
 A1_d(l)%JA = A1(l)%JA(1:A1(l)%nrow+1)
 A1_d(l)%jndx = A1(l)%jndx(1:A1(l)%nnz)

 A2_d(l)%val = A2(l)%val(1:A2(l)%nnz)
 A2_d(l)%JA = A2(l)%JA(1:A2(l)%nrow+1)
 A2_d(l)%jndx = A2(l)%jndx(1:A2(l)%nnz)
end do

do l = lmax, lmin, -1
 T_d(l)%val = T(l)%val(1:T(l)%nnz)
 T_d(l)%JA = T(l)%JA(1:T(l)%nrow+1)
 T_d(l)%jndx = T(l)%jndx(1:T(l)%nnz)
end do

istat = cublasCreate(h1)
```

```
 status = cusparseCreate(h)
 status = cusparseCreateMatDescr(descrA)
 status = cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL)
 status = cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE)

do ij = 1, it
! near matrix
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              nearmatrix%nrow, nearmatrix%ncol, &
                              nearmatrix%nnz, alpha, descrA, &
                              nearmatrix_d%val, nearmatrix_d%JA, &
                              nearmatrix_d%jndx, x_d, beta, Axnear_d)

! expand waves
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                              V1_d%val, V1_d%JA, V1_d%jndx, x_d, &
                              beta, v_begin_d%vector(:,1))

 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                              V1_d%val2, V1_d%JA, V1_d%jndx, x_d, &
                              beta, v_begin_d%vector(:,2))

 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                              V1_d%val3, V1_d%JA, V1_d%jndx, x_d, &
                              beta, v_begin_d%vector(:,3))

 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,1), 1, &
         v_outgoing_d(lmax)%vector(:,1),1)
 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,2), 1, &
         v_outgoing_d(lmax)%vector(:,2),1)
 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,3), 1, &
         v_outgoing_d(lmax)%vector(:,3),1)

! interpolate waves
 do l = lmax-1, lmin, -1
  status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                               Itheta1(l)%nrow, 3, Itheta1(l)%ncol,
                               Itheta1(l)%nnz, alpha, descrA, &
                               Itheta1_d(l)%val, Itheta1_d(l)%JA, &
                               Itheta1_d(l)%jndx, v_outgoing_d(l+1)%vector, &
                               Itheta1(l)%ncol, beta, tijdelijk_d(l)%uit, &
                               Itheta1(l)%nrow)

  status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                               Iphi1(l)%nrow, 3, Iphi1(l)%ncol, &
                               Iphi1(l)%nnz, alpha, descrA, &
                                 Iphi1_d(l)%val, Iphi1_d(l)%JA, &
                               Iphi1_d(l)%jndx, tijdelijk_d(l)%uit, &
                               Iphi1(l)%ncol, beta, tijdelijk_d(l)%uit2, &
                               Iphi1(l)%nrow)

  status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                               A1(l)%nrow, 3, A1(l)%ncol, &
```

```
                                A1(l)%nnz, alpha, descrA, &
                                A1_d(l)%val, A1_d(l)%JA, &
                                A1_d(l)%jndx, tijdelijk_d(l)%uit2, &
                                A1(l)%ncol, beta, v_outgoing_d(l)%vector, &
                                A1(l)%nrow)
 end do

! transfer waves
 do l = lmax, lmin, -1
  status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                                T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                                descrA, T_d(l)%val, T_d(l)%JA, T_d(l)%jndx, &
                                v_outgoing_d(l)%vector, T(l)%ncol, beta, &
                                v_incoming_d(l)%vector, T(l)%nrow)
 end do

 istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
        %vector(:,1), 1, w_incoming_d(lmin-1)%vector(:,1),1)
 istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
        %vector(:,2), 1, w_incoming_d(lmin-1)%vector(:,2),1)
 istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
        %vector(:,3), 1, w_incoming_d(lmin-1)%vector(:,3),1)

! anterpolate waves
 do l = lmin, lmax-1
  status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                                A2(l)%nrow, 3, A2(l)%ncol, A2(l)%nnz, &
                                alpha, descrA, A2_d(l)%val, A2_d(l)%JA, &
                                A2_d(l)%jndx, w_incoming_d(l-1)%vector, &
                                A2(l)%ncol, beta, tijdelijk1_d(l)%uit, &
                                A2(l)%nrow)

  status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                                Iphi2(l)%nrow, 3, Iphi2(l)%ncol, Iphi2(l)%nnz, &
                                alpha, descrA, Iphi2_d(l)%val, Iphi2_d(l)%JA, &
                                Iphi2_d(l)%jndx, tijdelijk1_d(l)%uit, &
                                Iphi2(l)%ncol, beta, tijdelijk1_d(l)%uit2, &
                                Iphi2(l)%nrow)

  status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                                Itheta2(l)%nrow, 3, Itheta2(l)%ncol, Itheta2(l)%nnz, &
                                alpha, descrA, Itheta2_d(l)%val, Itheta2_d(l)%JA, &
                                Itheta2_d(l)%jndx, tijdelijk1_d(l)%uit2, &
                                Itheta2(l)%ncol, beta, w_incoming_d(l) &
                                %vector, Itheta2(l)%nrow)

  istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
        (l+1)%vector(:,1), 1 , w_incoming_d(l)%vector(:,1), 1)
  istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
        (l+1)%vector(:,2), 1 , w_incoming_d(l)%vector(:,2), 1)
  istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
        (l+1)%vector(:,3), 1 , w_incoming_d(l)%vector(:,3), 1)
 end do

! incoming waves
   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
```

```
                                      V2%nrow, V2%ncol, V2%nnz, alpha, &
                                      descrA, V2_d%val, V2_d%JA, V2_d%jndx, &
                                      w_incoming_d(lmax-1)%vector(:,1), beta, &
                                      incoming_d%vector(:,1))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, &
                         descrA, V2_d%val2, V2_d%JA, V2_d%jndx, &
                         w_incoming_d(lmax-1)%vector(:,2), beta, &
                         incoming_d%vector(:,2))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, &
                         descrA, V2_d%val3, V2_d%JA, V2_d%jndx, &
                         w_incoming_d(lmax-1)%vector(:,3), beta, &
                         incoming_d%vector(:,3))

  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,2), &
        1 , incoming_d%vector(:,1), 1)

  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,3), &
        1 , incoming_d%vector(:,1), 1)

  istat = cublasCcopy_v2(h1, V2%nrow, incoming_d%vector(:,1), &
        1, Axfar_d%vector(:,1),1)

! final vector
  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, Axnear_d, 1, &
        Axfar_d%vector(:,1), 1)
 end do

 status = cusparseDestroy(h)

! Send solution vector back to CPU.
 Axfar%vector(:,1) = Axfar_d%vector(:,1)

End Program MLFMA
```

D.4. MLFMA Storing the Interpolation Operators as Real Matrices

```
Program MLFMA
use Sort
use cudafor
use cublas
use cusparse
implicit none

! Matrices are read from files and saved in the CRS format here.

 do i = 1,nearmatrix%ncol
  x(i) = cmplx(0.0,0.0)
 end do

 x(333) = cmplx(1.0,0.0)

 alpha = cmplx(1.0,0.0)
```

```
beta = cmplx(0.0,0.0)

! Send information about the matrices to the GPU.
nearmatrix_d%val = nearmatrix%val(1:nearmatrix%nnz)
nearmatrix_d%JA = nearmatrix%JA(1:nearmatrix%nrow+1)
nearmatrix_d%jndx = nearmatrix%jndx(1:nearmatrix%nnz)
x_d = x(1:nearmatrix%ncol)

V1_d%val = V1%val(1:V1%nnz)
V1_d%val2 = V1%val2(1:V1%nnz)
V1_d%val3 = V1%val3(1:V1%nnz)
V1_d%JA = V1%JA(1:V1%nrow+1)
V1_d%jndx = V1%jndx(1:V1%nnz)

V2_d%val = V2%val(1:V2%nnz)
V2_d%val2 = V2%val2(1:V2%nnz)
V2_d%val3 = V2%val3(1:V2%nnz)
V2_d%JA = V2%JA(1:V2%nrow+1)
V2_d%jndx = V2%jndx(1:V2%nnz)

do l = lmax-1, lmin, -1
 Itheta1_d(l)%val = Itheta1(l)%val(1:Itheta1(l)%nnz)
 Itheta1_d(l)%JA = Itheta1(l)%JA(1:Itheta1(l)%nrow+1)
 Itheta1_d(l)%jndx = Itheta1(l)%jndx(1:Itheta1(l)%nnz)

 Itheta2_d(l)%val = Itheta2(l)%val(1:Itheta2(l)%nnz)
 Itheta2_d(l)%JA = Itheta2(l)%JA(1:Itheta2(l)%nrow+1)
 Itheta2_d(l)%jndx = Itheta2(l)%jndx(1:Itheta2(l)%nnz)

 Iphi1_d(l)%val = Iphi1(l)%val(1:Iphi1(l)%nnz)
 Iphi1_d(l)%JA = Iphi1(l)%JA(1:Iphi1(l)%nrow+1)
 Iphi1_d(l)%jndx = Iphi1(l)%jndx(1:Iphi1(l)%nnz)

 Iphi2_d(l)%val = Iphi2(l)%val(1:Iphi2(l)%nnz)
 Iphi2_d(l)%JA = Iphi2(l)%JA(1:Iphi2(l)%nrow+1)
 Iphi2_d(l)%jndx = Iphi2(l)%jndx(1:Iphi2(l)%nnz)

 A1_d(l)%val = A1(l)%val(1:A1(l)%nnz)
 A1_d(l)%JA = A1(l)%JA(1:A1(l)%nrow+1)
 A1_d(l)%jndx = A1(l)%jndx(1:A1(l)%nnz)

 A2_d(l)%val = A2(l)%val(1:A2(l)%nnz)
 A2_d(l)%JA = A2(l)%JA(1:A2(l)%nrow+1)
 A2_d(l)%jndx = A2(l)%jndx(1:A2(l)%nnz)
end do

do l = lmax, lmin, -1
 T_d(l)%val = T(l)%val(1:T(l)%nnz)
 T_d(l)%JA = T(l)%JA(1:T(l)%nrow+1)
 T_d(l)%jndx = T(l)%jndx(1:T(l)%nnz)
end do

istat = cublasCreate(h1)
status = cusparseCreate(h)
status = cusparseCreateMatDescr(descrA)
status = cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL)
```

```fortran
status = cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE)

do ij = 1, it
! near matrix
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         nearmatrix%nrow, nearmatrix%ncol, &
                         nearmatrix%nnz, alpha, descrA, nearmatrix_d &
                         %val, nearmatrix_d%JA, nearmatrix_d%jndx, &
                         x_d, beta, Axnear_d)

! expand waves
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector(:,1))

 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val2, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector(:,2))

 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val3, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector(:,3))

 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,1), 1, &
       v_outgoing_d(lmax)%vector(:,1),1)
 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,2), 1, &
       v_outgoing_d(lmax)%vector(:,2),1)
 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,3), 1, &
       v_outgoing_d(lmax)%vector(:,3),1)

! interpolate waves
 do l = lmax-1, lmin, -1
  v_outgoing(l+1)%vector(:,1:3) = v_outgoing_d(l+1)%vector

  do i = 1, 3
   v_beginn(l+1)%inn(:,i) = real(v_outgoing(l+1)%vector(:,i))
   v_beginn(l+1)%uit(:,i) = aimag(v_outgoing(l+1)%vector(:,i))
  end do

  v_beginn_d(l+1)%inn = v_beginn(l+1)%inn(:,1:3)
  v_beginn_d(l+1)%uit = v_beginn(l+1)%uit(:,1:3)

  status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Itheta1(l)%nrow, 3, Itheta1(l)%ncol, &
                          Itheta1(l)%nnz, 1.0, descrA, Itheta1_d(l) &
                          %val, Itheta1_d(l)%JA, Itheta1_d(l)%jndx, &
                          v_beginn_d(l+1)%inn, Itheta1(l)%ncol, 0.0, &
                          v_beginnn_d(l)%inn, Itheta1(l)%nrow)

  status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Itheta1(l)%nrow, 3, Itheta1(l)%ncol, &
                          Itheta1(l)%nnz, 1.0, descrA, Itheta1_d(l) &
                          %val, Itheta1_d(l)%JA, Itheta1_d(l)%jndx, &
```

```
                              v_beginn_d(l+1)%uit, Itheta1(l)%ncol, 0.0, &
                              v_beginnn_d(l)%uit, Itheta1(l)%nrow)

     status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              Iphi1(l)%nrow, 3, Iphi1(l)%ncol, &
                              Iphi1(l)%nnz, 1.0, descrA, Iphi1_d(l) &
                              %val, Iphi1_d(l)%JA, Iphi1_d(l)%jndx, &
                              v_beginnn_d(l)%inn, Iphi1(l)%ncol, 0.0, &
                              v_begi_d(l)%inn, Iphi1(l)%nrow)

     status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              Iphi1(l)%nrow, 3, Iphi1(l)%ncol, &
                              Iphi1(l)%nnz, 1.0, descrA, Iphi1_d(l) &
                              %val, Iphi1_d(l)%JA, Iphi1_d(l)%jndx, &
                              v_beginnn_d(l)%uit, Iphi1(l)%ncol, 0.0, &
                              v_begi_d(l)%uit, Iphi1(l)%nrow)

     v_begi(l)%inn = v_begi_d(l)%inn(:,1:3)
     v_begi(l)%uit = v_begi_d(l)%uit(:,1:3)

     do i = 1, 3
      tijdelijk(l)%uit2(:,i) = cmplx(v_begi(l)%inn(:,i), &
          v_begi(l)%uit(:,i))
     end do

     tijdelijk_d(l)%uit2 = tijdelijk(l)%uit2(:,1:3)

     status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              A1(l)%nrow, 3, A1(l)%ncol, A1(l)%nnz, &
                              alpha, descrA, A1_d(l)%val, A1_d(l)%JA, &
                              A1_d(l)%jndx, tijdelijk_d(l)%uit2, A1(l) &
                              %ncol, beta, v_outgoing_d(l)%vector, &
                              A1(l)%nrow)
    end do

  ! transfer waves
    do l = lmax, lmin, -1
     status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                              descrA, T_d(l)%val, T_d(l)%JA, T_d(l)%jndx, &
                              v_outgoing_d(l)%vector, T(l)%ncol, beta, &
                              v_incoming_d(l)%vector, T(l)%nrow)
    end do

    istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector(:,1), 1, w_incoming_d(lmin-1)%vector(:,1),1)
    istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector(:,2), 1, w_incoming_d(lmin-1)%vector(:,2),1)
    istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector(:,3), 1, w_incoming_d(lmin-1)%vector(:,3),1)

  ! anterpolate waves
    do l = lmin, lmax-1
     status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              A2(l)%nrow, 3, A2(l)%ncol, A2(l)%nnz, &
                              alpha, descrA, A2_d(l)%val, A2_d(l)%JA, &
```

```
                        A2_d(l)%jndx, w_incoming_d(l-1)%vector, &
                        A2(l)%ncol, beta, tijdelijk1_d(l)%uit, &
                        A2(l)%nrow)

tijdelijk1(l)%uit = tijdelijk1_d(l)%uit(:,1:3)

do i = 1, 3
 w_beginn(l)%inn(:,i) = real(tijdelijk1(l)%uit(:,i))
 w_beginn(l)%uit(:,i) = aimag(tijdelijk1(l)%uit(:,i))
end do

w_beginn_d(l)%inn = w_beginn(l)%inn(:,1:3)
w_beginn_d(l)%uit = w_beginn(l)%uit(:,1:3)

status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        Iphi2(l)%nrow, 3, Iphi2(l)%ncol, Iphi2(l) &
                        %nnz, 1.0, descrA, Iphi2_d(l)%val, Iphi2_d &
                        (l)%JA, Iphi2_d(l)%jndx, w_beginn_d(l)%in, &
                        Iphi2(l)%ncol, 0.0, w_beginnn_d(l)%in, &
                        Iphi2(l)%nrow)

status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        Iphi2(l)%nrow, 3, Iphi2(l)%ncol, Iphi2(l) &
                        %nnz, 1.0, descrA, Iphi2_d(l)%val, Iphi2_d &
                        (l)%JA, Iphi2_d(l)%jndx, w_beginn_d(l)%uit, &
                        Iphi2(l)%ncol, 0.0, w_beginnn_d(l)%uit, &
                        Iphi2(l)%nrow)

status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        Itheta2(l)%nrow, 3, Itheta2(l)%ncol,
                        Itheta2(l)%nnz, 1.0, descrA, Itheta2_d(l) &
                        %val, Itheta2_d(l)%JA, Itheta2_d(l)%jndx, &
                        w_beginnn_d(l)%inn, Itheta2(l)%ncol, 0.0, &
                        w_begi_d(l)%inn, Itheta2(l)%nrow)

status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        Itheta2(l)%nrow, 3, Itheta2(l)%ncol, &
                        Itheta2(l)%nnz, 1.0, descrA, Itheta2_d(l) &
                        %val, Itheta2_d(l)%JA, Itheta2_d(l)%jndx, &
                        w_beginnn_d(l)%uit, Itheta2(l)%ncol, 0.0, 7
                        w_begi_d(l)%uit, Itheta2(l)%nrow)

w_begi(l)%inn = w_begi_d(l)%inn(:,1:3)
w_begi(l)%uit = w_begi_d(l)%uit(:,1:3)

do i = 1,3
 w_incoming(l)%vector(:,i) = cmplx(w_begi(l)%inn(:,i), &
     w_begi(l)%uit(:,i))
end do

w_incoming_d(l)%vector = w_incoming(l)%vector(:,1:3)

istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
     (l+1)%vector(:,1), 1 , w_incoming_d(l)%vector(:,1), 1)
istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
     (l+1)%vector(:,2), 1 , w_incoming_d(l)%vector(:,2), 1)
```

```
   istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
         (l+1)%vector(:,3), 1 , w_incoming_d(l)%vector(:,3), 1)
  end do

! incoming waves
  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          V2%nrow, V2%ncol, V2%nnz, alpha, &
                          descrA, V2_d%val, V2_d%JA, V2_d%jndx, &
                          w_incoming_d(lmax-1)%vector(:,1), beta, &
                          incoming_d%vector(:,1))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          V2%nrow, V2%ncol, V2%nnz, alpha, &
                          descrA, V2_d%val2, V2_d%JA, V2_d%jndx, &
                          w_incoming_d(lmax-1)%vector(:,2), beta, &
                          incoming_d%vector(:,2))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          V2%nrow, V2%ncol, V2%nnz, alpha, &
                          descrA, V2_d%val3, V2_d%JA, V2_d%jndx, &
                          w_incoming_d(lmax-1)%vector(:,3), beta, &
                          incoming_d%vector(:,3))

  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,2), &
        1 , incoming_d%vector(:,1), 1)

  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,3), &
        1 , incoming_d%vector(:,1), 1)

  istat = cublasCcopy_v2(h1, V2%nrow, incoming_d%vector(:,1), 1, &
        Axfar_d%vector(:,1),1)

! final vector
  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, Axnear_d, 1, &
        Axfar_d%vector(:,1), 1)
 end do

 status = cusparseDestroy(h)

! Send solution vector back to CPU.
 Axfar%vector(:,1) = Axfar_d%vector(:,1)

End Program MLFMA
```

D.5. MLFMA Performing Matrix Vector Products in Real Arithmetic

```
Program MLFMA
use Sort
use cudafor
use cublas
use cusparse
implicit none

! Matrices are read from files and saved in the CRS format here.

 do i = 1,nearmatrix%ncol
```

```fortran
 x(i) = cmplx(0.0,0.0)
end do

x(333) = cmplx(1.0,0.0)

alpha = cmplx(1.0,0.0)
beta = cmplx(0.0,0.0)

! Send information about the matrices to the GPU.
nearmatrix_d%val = nearmatrix%val(1:nearmatrix%nnz)
nearmatrix_d%JA = nearmatrix%JA(1:nearmatrix%nrow+1)
nearmatrix_d%jndx = nearmatrix%jndx(1:nearmatrix%nnz)
x_d = x(1:nearmatrix%ncol)

V1_d%val = V1%val(1:V1%nnz)
V1_d%val2 = V1%val2(1:V1%nnz)
V1_d%val3 = V1%val3(1:V1%nnz)
V1_d%JA = V1%JA(1:V1%nrow+1)
V1_d%jndx = V1%jndx(1:V1%nnz)

V2_d%val = V2%val(1:V2%nnz)
V2_d%val2 = V2%val2(1:V2%nnz)
V2_d%val3 = V2%val3(1:V2%nnz)
V2_d%JA = V2%JA(1:V2%nrow+1)
V2_d%jndx = V2%jndx(1:V2%nnz)

do l = lmax-1, lmin, -1
 Itheta1_d(l)%val = Itheta1(l)%val(1:Itheta1(l)%nnz)
 Itheta1_d(l)%JA = Itheta1(l)%JA(1:Itheta1(l)%nrow+1)
 Itheta1_d(l)%jndx = Itheta1(l)%jndx(1:Itheta1(l)%nnz)

 Itheta2_d(l)%val = Itheta2(l)%val(1:Itheta2(l)%nnz)
 Itheta2_d(l)%JA = Itheta2(l)%JA(1:Itheta2(l)%nrow+1)
 Itheta2_d(l)%jndx = Itheta2(l)%jndx(1:Itheta2(l)%nnz)

 Iphi1_d(l)%val = Iphi1(l)%val(1:Iphi1(l)%nnz)
 Iphi1_d(l)%JA = Iphi1(l)%JA(1:Iphi1(l)%nrow+1)
 Iphi1_d(l)%jndx = Iphi1(l)%jndx(1:Iphi1(l)%nnz)

 Iphi2_d(l)%val = Iphi2(l)%val(1:Iphi2(l)%nnz)
 Iphi2_d(l)%JA = Iphi2(l)%JA(1:Iphi2(l)%nrow+1)
 Iphi2_d(l)%jndx = Iphi2(l)%jndx(1:Iphi2(l)%nnz)

 A1_d(l)%val = A1(l)%val(1:A1(l)%nnz)
 A1_d(l)%JA = A1(l)%JA(1:A1(l)%nrow+1)
 A1_d(l)%jndx = A1(l)%jndx(1:A1(l)%nnz)

 A2_d(l)%val = A2(l)%val(1:A2(l)%nnz)
 A2_d(l)%JA = A2(l)%JA(1:A2(l)%nrow+1)
 A2_d(l)%jndx = A2(l)%jndx(1:A2(l)%nnz)
end do

do l = lmax, lmin, -1
 T_d(l)%val = T(l)%val(1:T(l)%nnz)
 T_d(l)%JA = T(l)%JA(1:T(l)%nrow+1)
 T_d(l)%jndx = T(l)%jndx(1:T(l)%nnz)
```

```
 end do

 istat = cublasCreate(h1)
 status = cusparseCreate(h)
 status = cusparseCreateMatDescr(descrA)
 status = cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL)
 status = cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE)

 do ij = 1, it
! near matrix
  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         nearmatrix%nrow, nearmatrix%ncol, &
                         nearmatrix%nnz, alpha, descrA, nearmatrix_d &
                         %val, nearmatrix_d%JA, nearmatrix_d%jndx, &
                         x_d, beta, Axnear_d%vector)

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         nearmatrix%nrow, nearmatrix%ncol, &
                         nearmatrix%nnz, alpha, descrA, nearmatrix_d &
                         %val2, nearmatrix_d%JA, nearmatrix_d%jndx, &
                         x_d, beta, Axnear_d%vector2)

! expand waves

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector(:,1))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val2, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector2(:,1))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val3, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector(:,2))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val4, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector2(:,2))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val5, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector(:,3))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                         V1_d%val6, V1_d%JA, V1_d%jndx, x_d, beta, &
                         v_begin_d%vector2(:,3))

  istat = cublasScopy_v2(h1, V1%nrow, v_begin_d%vector(:,1), 1, &
        v_outgoing_d(lmax)%vector(:,1),1)
```

```fortran
  istat = cublasScopy_v2(h1, V1%nrow, v_begin_d%vector(:,2), 1, &
         v_outgoing_d(lmax)%vector(:,2),1)
  istat = cublasScopy_v2(h1, V1%nrow, v_begin_d%vector(:,3), 1, &
        v_outgoing_d(lmax)%vector(:,3),1)
  istat = cublasScopy_v2(h1, V1%nrow, v_begin_d%vector2(:,1), 1, &
         v_outgoing_d(lmax)%vector2(:,1),1)
  istat = cublasScopy_v2(h1, V1%nrow, v_begin_d%vector2(:,2), 1, &
        v_outgoing_d(lmax)%vector2(:,2),1)
  istat = cublasScopy_v2(h1, V1%nrow, v_begin_d%vector2(:,3), 1, &
        v_outgoing_d(lmax)%vector2(:,3),1)

! interpolate waves
  do l = lmax-1, lmin, -1
   status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Itheta1(l)%nrow, 3, Itheta1(l)%ncol, &
                          Itheta1(l)%nnz, alpha, descrA, Itheta1_d &
                          (l)%val, Itheta1_d(l)%JA, Itheta1_d(l)%&
                          jndx, v_outgoing_d(l+1)%vector, Itheta1 &
                          (l)%ncol, beta, tijdelijk_d(l)%uit, &
                          Itheta1(l)%nrow)

   status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Itheta1(l)%nrow, 3, Itheta1(l)%ncol, &
                          Itheta1(l)%nnz, alpha, descrA, Itheta1_d &
                          (l)%val, Itheta1_d(l)%JA, Itheta1_d(l)% &
                          jndx, v_outgoing_d(l+1)%vector2, Itheta1 &
                          (l)%ncol, beta, tijdelijk_d(l)%uit3, &
                          Itheta1(l)%nrow)

   status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Iphi1(l)%nrow, 3, Iphi1(l)%ncol, &
                          Iphi1(l)%nnz, alpha, descrA, Iphi1_d &
                          (l)%val, Iphi1_d(l)%JA, Iphi1_d(l)% &
                          jndx, tijdelijk_d(l)%uit, Iphi1(l)%ncol, &
                          beta, tijdelijk_d(l)%uit2, Iphi1(l)%nrow)

   status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Iphi1(l)%nrow, 3, Iphi1(l)%ncol, &
                          Iphi1(l)%nnz, alpha, descrA, Iphi1_d &
                          (l)%val, Iphi1_d(l)%JA, Iphi1_d(l)% &
                          jndx, tijdelijk_d(l)%uit3, Iphi1(l)%ncol, &
                          beta, tijdelijk_d(l)%uit4, Iphi1(l)%nrow)

   status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          A1(l)%nrow, 3, A1(l)%ncol, &
                          A1(l)%nnz, alpha, descrA, A1_d(l) &
                          %val, A1_d(l)%JA, A1_d(l)%jndx, &
                          tijdelijk_d(l)%uit2, A1(l)%ncol, &
                          beta, v_outgoing_d(l)%vector, &
                          A1(l)%nrow)

   status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          A1(l)%nrow, 3, A1(l)%ncol, &
                          A1(l)%nnz, alpha, descrA, A1_d(l) &
                          %val2, A1_d(l)%JA, A1_d(l)%jndx, &
                          tijdelijk_d(l)%uit4, A1(l)%ncol, &
```

```
                        beta, v_outgoing_d(l)%vectorr, &
                        A1(l)%nrow)

    istat = cublasSaxpy_v2(h1, A1(l)%nrow, -1.0, v_outgoing_d(l) &
         %vectorr(:,1), 1 , v_outgoing_d(l)%vector(:,1), 1)
    istat = cublasSaxpy_v2(h1, A1(l)%nrow, -1.0, v_outgoing_d(l) &
         %vectorr(:,2), 1 , v_outgoing_d(l)%vector(:,2), 1)
    istat = cublasSaxpy_v2(h1, A1(l)%nrow, -1.0, v_outgoing_d(l) &
         %vectorr(:,3), 1 , v_outgoing_d(l)%vector(:,3), 1)

    status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        A1(l)%nrow, 3, A1(l)%ncol, A1(l)%nnz, &
                        alpha, descrA, A1_d(l)%val, A1_d(l)%JA, &
                        A1_d(l)%jndx, tijdelijk_d(l)%uit4, A1 &
                        (l)%ncol, beta, v_outgoing_d(l)%vector2, &
                        A1(l)%nrow)

    status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        A1(l)%nrow, 3, A1(l)%ncol, A1(l)%nnz, &
                        alpha, descrA, A1_d(l)%val2, A1_d(l)%JA, &
                        A1_d(l)%jndx, tijdelijk_d(l)%uit2, A1(l) &
                        %ncol, beta, v_outgoing_d(l)%vectorr2, &
                        A1(l)%nrow)

    istat = cublasSaxpy_v2(h1, A1(l)%nrow, 1.0, v_outgoing_d(l) &
         %vectorr2(:,1), 1 , v_outgoing_d(l)%vector2(:,1), 1)
    istat = cublasSaxpy_v2(h1, A1(l)%nrow, 1.0, v_outgoing_d(l) &
         %vectorr2(:,2), 1 , v_outgoing_d(l)%vector2(:,2), 1)
    istat = cublasSaxpy_v2(h1, A1(l)%nrow, 1.0, v_outgoing_d(l) &
         %vectorr2(:,3), 1 , v_outgoing_d(l)%vector2(:,3), 1)
  end do

! transfer waves
  do l = lmax, lmin, -1
    status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                        descrA, T_d(l)%val, T_d(l)%JA, T_d(l)%jndx, &
                        v_outgoing_d(l)%vector, T(l)%ncol, beta, &
                        v_incoming_d(l)%vector, T(l)%nrow)

    status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                        descrA, T_d(l)%val2, T_d(l)%JA, T_d(l)%jndx, &
                        v_outgoing_d(l)%vector2, T(l)%ncol, beta, &
                        v_incoming_d(l)%vectorr, T(l)%nrow)

    istat = cublasSaxpy_v2(h1, T(l)%nrow, -1.0, v_incoming_d(l) &
         %vectorr(:,1), 1 , v_incoming_d(l)%vector(:,1), 1)
    istat = cublasSaxpy_v2(h1, T(l)%nrow, -1.0, v_incoming_d(l) &
         %vectorr(:,2), 1 , v_incoming_d(l)%vector(:,2), 1)
    istat = cublasSaxpy_v2(h1, T(l)%nrow, -1.0, v_incoming_d(l) &
         %vectorr(:,3), 1 , v_incoming_d(l)%vector(:,3), 1)

    status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                        T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                        descrA, T_d(l)%val, T_d(l)%JA, T_d(l)%jndx, &
```

```
                              v_outgoing_d(l)%vector2, T(l)%ncol, beta, &
                              v_incoming_d(l)%vector2, T(l)%nrow)

     status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                              descrA, T_d(l)%val2, T_d(l)%JA, T_d(l)%jndx, &
                              v_outgoing_d(l)%vector, T(l)%ncol, beta, &
                              v_incoming_d(l)%vectorr2, T(l)%nrow)

     istat = cublasSaxpy_v2(h1, T(l)%nrow, 1.0, v_incoming_d(l) &
          %vectorr2(:,1), 1 , v_incoming_d(l)%vector2(:,1), 1)
     istat = cublasSaxpy_v2(h1, T(l)%nrow, 1.0, v_incoming_d(l) &
          %vectorr2(:,2), 1 , v_incoming_d(l)%vector2(:,2), 1)
     istat = cublasSaxpy_v2(h1, T(l)%nrow, 1.0, v_incoming_d(l) &
          %vectorr2(:,3), 1 , v_incoming_d(l)%vector2(:,3), 1)
    end do

    istat = cublasScopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector(:,1), 1, w_incoming_d(lmin-1)%vector(:,1),1)
    istat = cublasScopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector(:,2), 1, w_incoming_d(lmin-1)%vector(:,2),1)
    istat = cublasScopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector(:,3), 1, w_incoming_d(lmin-1)%vector(:,3),1)
    istat = cublasScopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector2(:,1), 1, w_incoming_d(lmin-1)%vector2(:,1),1)
    istat = cublasScopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector2(:,2), 1, w_incoming_d(lmin-1)%vector2(:,2),1)
    istat = cublasScopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
          %vector2(:,3), 1, w_incoming_d(lmin-1)%vector2(:,3),1)

  ! anterpolate waves
    do l = lmin, lmax-1
     status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              A2(l)%nrow, 3, A2(l)%ncol, A2(l)%nnz, &
                              alpha, descrA, A2_d(l)%val, A2_d(l)%JA, &
                              A2_d(l)%jndx, w_incoming_d(l-1)%vector, &
                              A2(l)%ncol, beta, tijdelijk1_d(l)%uit, &
                              A2(l)%nrow)

     status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              A2(l)%nrow, 3, A2(l)%ncol, A2(l)%nnz, &
                              alpha, descrA, A2_d(l)%val2, A2_d(l)%JA, &
                              A2_d(l)%jndx, w_incoming_d(l-1)%vector2, &
                              A2(l)%ncol, beta, tijdelijk1_d(l)%uitt, &
                              A2(l)%nrow)

     istat = cublasSaxpy_v2(h1, A2(l)%nrow, -1.0, tijdelijk1_d &
          (l)%uitt(:,1), 1 , tijdelijk1_d(l)%uit(:,1), 1)
     istat = cublasSaxpy_v2(h1, A2(l)%nrow, -1.0, tijdelijk1_d &
          (l)%uitt(:,2), 1 , tijdelijk1_d(l)%uit(:,2), 1)
     istat = cublasSaxpy_v2(h1, A2(l)%nrow, -1.0, tijdelijk1_d &
          (l)%uitt(:,3), 1 , tijdelijk1_d(l)%uit(:,3), 1)

     status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
            A2(l)%nrow, 3, A2(l)%ncol, A2(l)%nnz, alpha, descrA, &
            A2_d(l)%val, A2_d(l)%JA, A2_d(l)%jndx, w_incoming_d &
```

```
              (l-1)%vector2, A2(l)%ncol, beta, tijdelijk1_d(l)%uit3, &
              A2(l)%nrow)

      status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
              A2(l)%nrow, 3, A2(l)%ncol, A2(l)%nnz, alpha, descrA, &
              A2_d(l)%val2, A2_d(l)%JA, A2_d(l)%jndx, w_incoming_d &
              (l-1)%vector, A2(l)%ncol, beta, tijdelijk1_d(l)%uitt3, &
              A2(l)%nrow)

      istat = cublasSaxpy_v2(h1, A2(l)%nrow, 1.0, tijdelijk1_d(l) &
           %uitt3(:,1), 1 , tijdelijk1_d(l)%uit3(:,1), 1)
      istat = cublasSaxpy_v2(h1, A2(l)%nrow, 1.0, tijdelijk1_d(l) &
           %uitt3(:,2), 1 , tijdelijk1_d(l)%uit3(:,2), 1)
      istat = cublasSaxpy_v2(h1, A2(l)%nrow, 1.0, tijdelijk1_d(l) &
           %uitt3(:,3), 1 , tijdelijk1_d(l)%uit3(:,3), 1)

      status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              Iphi2(l)%nrow, 3, Iphi2(l)%ncol, &
                              Iphi2(l)%nnz, alpha, descrA, Iphi2_d(l) &
                              %val, Iphi2_d(l)%JA, Iphi2_d(l)%jndx, &
                              tijdelijk1_d(l)%uit, Iphi2(l)%ncol, beta, &
                              tijdelijk1_d(l)%uit2, Iphi2(l)%nrow)

      status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              Iphi2(l)%nrow, 3, Iphi2(l)%ncol, &
                              Iphi2(l)%nnz, alpha, descrA, Iphi2_d(l) &
                              %val, Iphi2_d(l)%JA, Iphi2_d(l)%jndx, &
                              tijdelijk1_d(l)%uit3, Iphi2(l)%ncol, beta, &
                              tijdelijk1_d(l)%uit4, Iphi2(l)%nrow)

      status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              Itheta2(l)%nrow, 3, Itheta2(l)%ncol, &
                              Itheta2(l)%nnz, alpha, descrA, Itheta2_d(l) &
                              %val, Itheta2_d(l)%JA, Itheta2_d(l)%jndx, &
                              tijdelijk1_d(l)%uit2, Itheta2(l)%ncol, beta, &
                              w_incoming_d(l)%vector, Itheta2(l)%nrow)

      status = cusparseScsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              Itheta2(l)%nrow, 3, Itheta2(l)%ncol, &
                              Itheta2(l)%nnz, alpha, descrA, Itheta2_d(l) &
                              %val, Itheta2_d(l)%JA, Itheta2_d(l)%jndx, &
                              tijdelijk1_d(l)%uit4, Itheta2(l)%ncol, beta, &
                              w_incoming_d(l)%vector2, Itheta2(l)%nrow)

      istat = cublasSaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
           (l+1)%vector(:,1), 1 , w_incoming_d(l)%vector(:,1), 1)
      istat = cublasSaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
           (l+1)%vector(:,2), 1 , w_incoming_d(l)%vector(:,2), 1)
      istat = cublasSaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
           (l+1)%vector(:,3), 1 , w_incoming_d(l)%vector(:,3), 1)
      istat = cublasSaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
           (l+1)%vector2(:,1), 1 , w_incoming_d(l)%vector2(:,1), 1)
      istat = cublasSaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
           (l+1)%vector2(:,2), 1 , w_incoming_d(l)%vector2(:,2), 1)
      istat = cublasSaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
           (l+1)%vector2(:,3), 1 , w_incoming_d(l)%vector2(:,3), 1)
```

```
  end do

! incoming waves
  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                         V2_d%val, V2_d%JA, V2_d%jndx, w_incoming_d &
                         (lmax-1)%vector(:,1), beta, incoming_d &
                         %vector(:,1))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                         V2_d%val2, V2_d%JA, V2_d%jndx, w_incoming_d &
                         (lmax-1)%vector2(:,1), beta, incoming_d &
                         %vectorr(:,1))

  istat = cublasSaxpy_v2(h1, V2%nrow, -1.0, incoming_d%vectorr(:,1), &
        1 , incoming_d%vector(:,1), 1)

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                         V2_d%val, V2_d%JA, V2_d%jndx, w_incoming_d &
                         (lmax-1)%vector2(:,1), beta, incoming_ d&
                         %vector2(:,1))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                         V2_d%val2, V2_d%JA, V2_d%jndx, w_incoming_d &
                         (lmax-1)%vector(:,1), beta, incoming_d &
                         %vectorr2(:,1))

  istat = cublasSaxpy_v2(h1, V2%nrow, 1.0, incoming_d%vectorr2(:,1), &
        1 , incoming_d%vector2(:,1), 1)

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                         V2_d%val3, V2_d%JA, V2_d%jndx, w_incoming_d &
                         (lmax-1)%vector(:,2), beta, incoming_d &
                         %vector(:,2))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                         V2_d%val4, V2_d%JA, V2_d%jndx, w_incoming_d &
                         (lmax-1)%vector2(:,2), beta, incoming_d &
                         %vectorr(:,2))

  istat = cublasSaxpy_v2(h1, V2%nrow, -1.0, incoming_d%vectorr(:,2), &
        1 , incoming_d%vector(:,2), 1)

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                         V2_d%val3, V2_d%JA, V2_d%jndx, w_incoming_d &
                         (lmax-1)%vector2(:,2), beta, incoming_d &
                         %vector2(:,2))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
```

```
                              V2_d%val4, V2_d%JA, V2_d%jndx, w_incoming_d &
                              (lmax-1)%vector(:,2), beta, incoming_d &
                              %vectorr2(:,2))

  istat = cublasSaxpy_v2(h1, V2%nrow, 1.0, incoming_d%vectorr2(:,2), &
        1 , incoming_d%vector2(:,2), 1)

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                              V2_d%val5, V2_d%JA, V2_d%jndx, w_incoming_d &
                              (lmax-1)%vector(:,3), beta, incoming_d &
                              %vector(:,3))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                              V2_d%val6, V2_d%JA, V2_d%jndx, w_incoming_d &
                              (lmax-1)%vector2(:,3), beta, incoming_d &
                              %vectorr(:,3))

  istat = cublasSaxpy_v2(h1, V2%nrow, -1.0, incoming_d%vectorr(:,3), &
        1 , incoming_d%vector(:,3), 1)

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                              V2_d%val5, V2_d%JA, V2_d%jndx, w_incoming_d &
                              (lmax-1)%vector2(:,3), beta, incoming_d &
                              %vector2(:,3))

  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                              V2%nrow, V2%ncol, V2%nnz, alpha, descrA, &
                              V2_d%val6, V2_d%JA, V2_d%jndx, w_incoming_d &
                              (lmax-1)%vector(:,3), beta, incoming_d &
                               %vectorr2(:,3))

  istat = cublasSaxpy_v2(h1, V2%nrow, 1.0, incoming_d%vectorr2(:,3), &
        1 , incoming_d%vector2(:,3), 1)

  istat = cublasSaxpy_v2(h1, V1%ncol, alpha, incoming_d%vector(:,2), &
        1 , incoming_d%vector(:,1), 1)
  istat = cublasSaxpy_v2(h1, V1%ncol, alpha, incoming_d%vector2(:,2), &
        1 , incoming_d%vector2(:,1), 1)

  istat = cublasSaxpy_v2(h1, V1%ncol, alpha, incoming_d%vector(:,3), &
        1 , incoming_d%vector(:,1), 1)
  istat = cublasSaxpy_v2(h1, V1%ncol, alpha, incoming_d%vector2(:,3), &
        1 , incoming_d%vector2(:,1), 1)

  istat = cublasScopy_v2(h1, V1%ncol, incoming_d%vector(:,1), 1, &
        Axfar_d%vector(:,1),1)
  istat = cublasScopy_v2(h1, V1%ncol, incoming_d%vector2(:,1), 1, &
        Axfar_d%vector2(:,1),1)

! final vector
  istat = cublasSaxpy_v2(h1, V1%ncol, alpha, Axnear_d%vector, 1, &
        Axfar_d%vector(:,1), 1)
  istat = cublasSaxpy_v2(h1, V1%ncol, alpha, Axnear_d%vector2, 1, &
```

```
        Axfar_d%vector2(:,1), 1)

 end do

 status = cusparseDestroy(h)

! Send solution vector back to CPU.
 Axfar%vector(:,1) = Axfar_d%vector(:,1)
 Axfar%vector2(:,1) = Axfar_d%vector2(:,1)

End Program MLFMA
```

D.6. MLFMA Storing the Interpolation Operators as Small Matrices

```
Program MLFMA
use Sort
use cudafor
use cublas
use cusparse
implicit none

! Matrices are read from files and saved in the CRS format here.

 do i = 1,nearmatrix%ncol
  x(i) = cmplx(0.0,0.0)
 end do

 x(333) = cmplx(1.0,0.0)

 alpha = cmplx(1.0,0.0)
 beta = cmplx(0.0,0.0)

! Send information about the matrices to the GPU.
 nearmatrix_d%val = nearmatrix%val(1:nearmatrix%nnz)
 nearmatrix_d%JA = nearmatrix%JA(1:nearmatrix%nrow+1)
 nearmatrix_d%jndx = nearmatrix%jndx(1:nearmatrix%nnz)
 x_d = x(1:nearmatrix%ncol)

 V1_d%val = V1%val(1:V1%nnz)
 V1_d%val2 = V1%val2(1:V1%nnz)
 V1_d%val3 = V1%val3(1:V1%nnz)
 V1_d%JA = V1%JA(1:V1%nrow+1)
 V1_d%jndx = V1%jndx(1:V1%nnz)

 V2_d%val = V2%val(1:V2%nnz)
 V2_d%val2 = V2%val2(1:V2%nnz)
 V2_d%val3 = V2%val3(1:V2%nnz)
 V2_d%JA = V2%JA(1:V2%nrow+1)
 V2_d%jndx = V2%jndx(1:V2%nnz)

 do l = lmax-1, lmin, -1
  Itheta1_d(l)%val = Itheta1(l)%val(1:Itheta1(l)%nnz)
  Itheta1_d(l)%JA = Itheta1(l)%JA(1:Itheta1(l)%nrow+1)
  Itheta1_d(l)%jndx = Itheta1(l)%jndx(1:Itheta1(l)%nnz)

  Itheta2_d(l)%val = Itheta2(l)%val(1:Itheta2(l)%nnz)
```

```
  Itheta2_d(l)%JA = Itheta2(l)%JA(1:Itheta2(l)%nrow+1)
  Itheta2_d(l)%jndx = Itheta2(l)%jndx(1:Itheta2(l)%nnz)

  Iphi1_d(l)%val = Iphi1(l)%val(1:Iphi1(l)%nnz)
  Iphi1_d(l)%JA = Iphi1(l)%JA(1:Iphi1(l)%nrow+1)
  Iphi1_d(l)%jndx = Iphi1(l)%jndx(1:Iphi1(l)%nnz)

  Iphi2_d(l)%val = Iphi2(l)%val(1:Iphi2(l)%nnz)
  Iphi2_d(l)%JA = Iphi2(l)%JA(1:Iphi2(l)%nrow+1)
  Iphi2_d(l)%jndx = Iphi2(l)%jndx(1:Iphi2(l)%nnz)

  A1_d(l)%val = A1(l)%val(1:A1(l)%nnz)
  A1_d(l)%JA = A1(l)%JA(1:A1(l)%nrow+1)
  A1_d(l)%jndx = A1(l)%jndx(1:A1(l)%nnz)

  A2_d(l)%val = A2(l)%val(1:A2(l)%nnz)
  A2_d(l)%JA = A2(l)%JA(1:A2(l)%nrow+1)
  A2_d(l)%jndx = A2(l)%jndx(1:A2(l)%nnz)
 end do

 do l = lmax, lmin, -1
  T_d(l)%val = T(l)%val(1:T(l)%nnz)
  T_d(l)%JA = T(l)%JA(1:T(l)%nrow+1)
  T_d(l)%jndx = T(l)%jndx(1:T(l)%nnz)
 end do

 istat = cublasCreate(h1)
 status = cusparseCreate(h)
 status = cusparseCreateMatDescr(descrA)
 status = cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL)
 status = cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE)

 do ij = 1,it
! near matrix
  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          nearmatrix%nrow, nearmatrix%ncol, &
                          nearmatrix%nnz, alpha, &
                          descrA, nearmatrix_d%val, nearmatrix_d%JA, &
                          nearmatrix_d%jndx, x_d, beta, Axnear_d)

! expand waves
  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, V1%nrow, &
                          V1%ncol, V1%nnz, alpha, descrA, V1_d%val, &
                          V1_d%JA, V1_d%jndx, x_d, beta, v_begin_d &
                          %vector(:,1))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, V1%nrow, &
                          V1%ncol, V1%nnz, alpha, descrA, V1_d%val2, &
                          V1_d%JA, V1_d%jndx, x_d, beta, v_begin_d &
                          %vector(:,2))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, V1%nrow, &
                          V1%ncol, V1%nnz, alpha, descrA, V1_d%val3, &
                          V1_d%JA, V1_d%jndx, x_d, beta, v_begin_d &
                          %vector(:,3))
```

```
 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,1), 1, &
       v_outgoing_d(lmax)%vector(:,1),1)
 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,2), 1, &
       v_outgoing_d(lmax)%vector(:,2),1)
 istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,3), 1, &
       v_outgoing_d(lmax)%vector(:,3),1)

! interpolate waves
 do l = lmax-1, lmin, -1
  do i = 1, T(l+1)%G
   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Itheta1(l)%nrow2, Itheta1(l)%ncol2, &
                          Itheta1(l)%nnz2, alpha, descrA, &
                          Itheta1_d(l)%val, Itheta1_d(l)%JA, &
                          Itheta1_d(l)%jndx, v_outgoing_d(l+1)% &
                          vector((i-1)*Itheta1(l)%ncol2+1: &
                          i*Itheta1(l)%ncol2,1), beta, tijdelijk_d &
                          (l)%uit((i-1)*Itheta1(l)%nrow2+1: &
                          i*Itheta1(l)%nrow2,1))
  end do

  do i = 1, T(l+1)%G
   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Itheta1(l)%nrow2, Itheta1(l)%ncol2, &
                          Itheta1(l)%nnz2, alpha, descrA, &
                          Itheta1_d(l)%val, Itheta1_d(l)%JA, &
                          Itheta1_d(l)%jndx, v_outgoing_d(l+1)% &
                          vector((i-1)*Itheta1(l)%ncol2+1: &
                          i*Itheta1(l)%ncol2,2), beta, tijdelijk_d &
                          (l)%uit((i-1)*Itheta1(l)%nrow2+1: &
                          i*Itheta1(l)%nrow2,2))
  end do

  do i = 1, T(l+1)%G
   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Itheta1(l)%nrow2, Itheta1(l)%ncol2, &
                          Itheta1(l)%nnz2, alpha, descrA, &
                          Itheta1_d(l)%val, Itheta1_d(l)%JA, &
                          Itheta1_d(l)%jndx, v_outgoing_d(l+1)% &
                          vector((i-1)*Itheta1(l)%ncol2+1: &
                          i*Itheta1(l)%ncol2,3), beta, tijdelijk_d &
                          (l)%uit((i-1)*Itheta1(l)%nrow2+1: &
                          i*Itheta1(l)%nrow2,3))
  end do

  do i = 1, T(l+1)%G
   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          Iphi1(l)%nrow2, Iphi1(l)%ncol2, &
                          Iphi1(l)%nnz2, alpha, descrA, &
                          Iphi1_d(l)%val, Iphi1_d(l)%JA,&
                          Iphi1_d(l)%jndx, tijdelijk_d(l)%uit &
                          ((i-1)*Iphi1(l)%ncol2+1:i*Iphi1(l)% &
                          ncol2,1), beta, tijdelijk_d(l)%uit2 &
                          ((i-1)*Iphi1(l)%nrow2+1:Iphi1(l)%nrow2,1))
  end do
```

```
    do i = 1, T(l+1)%G
     status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                             Iphi1(l)%nrow2, Iphi1(l)%ncol2, &
                             Iphi1(l)%nnz2, alpha, descrA, &
                             Iphi1_d(l)%val, Iphi1_d(l)%JA, &
                             Iphi1_d(l)%jndx, tijdelijk_d(l)%uit &
                             ((i-1)*Iphi1(l)%ncol2+1:i*Iphi1(l)% &
                             ncol2,2), beta, tijdelijk_d(l)%uit2 &
                             ((i-1)*Iphi1(l)%nrow2+1:Iphi1(l)%nrow2,2))
    end do

    do i = 1, T(l+1)%G
     status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                             Iphi1(l)%nrow2, Iphi1(l)%ncol2, &
                             Iphi1(l)%nnz2, alpha, descrA, &
                             Iphi1_d(l)%val, Iphi1_d(l)%JA, &
                             Iphi1_d(l)%jndx, tijdelijk_d(l)%uit &
                             ((i-1)*Iphi1(l)%ncol2+1:i*Iphi1(l)% &
                             ncol2,3), beta, tijdelijk_d(l)%uit2 &
                             ((i-1)*Iphi1(l)%nrow2+1:Iphi1(l)%nrow2,3))
    end do

    status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            A1(l)%nrow, 3, A1(l)%ncol, &
                            A1(l)%nnz, alpha, descrA,
                            A1_d(l)%val, A1_d(l)%JA, &
                            A1_d(l)%jndx, tijdelijk_d(l)%uit2, &
                            A1(l)%ncol, beta, v_outgoing_d &
                            (l)%vector, A1(l)%nrow)
   end do

! transfer waves
   do l = lmax, lmin, -1
    status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                            descrA, T_d(l)%val, T_d(l)%JA, T_d(l) &
                            %jndx, v_outgoing_d(l)%vector, T(l)%ncol, &
                            beta, v_incoming_d(l)%vector, T(l)%nrow)
   end do

   istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
         %vector(:,1), 1, w_incoming_d(lmin-1)%vector(:,1),1)
   istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
         %vector(:,2), 1, w_incoming_d(lmin-1)%vector(:,2),1)
   istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
         %vector(:,3), 1, w_incoming_d(lmin-1)%vector(:,3),1)

! anterpolate waves
   do l = lmin, lmax-1
    status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            A2(l)%nrow, 3, A2(l)%ncol, A2(l)%nnz, &
                            alpha, descrA, A2_d(l)%val, A2_d(l)%JA, &
                            A2_d(l)%jndx, w_incoming_d(l-1)%vector, &
                            A2(l)%ncol, beta, tijdelijk1_d(l)%uit, &
                            A2(l)%nrow)
```

```
do i = 1, T(l+1)
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         Iphi2(l)%nrow2, Iphi2(l)%ncol2, Iphi2(l) &
                         %nnz2, alpha, descrA, Iphi2_d(l)%val, &
                         Iphi2_d(l)%JA, Iphi2_d(l)%jndx, &
                         tijdelijk1_d(l)%uit((i-1)*Iphi2(l)% &
                         ncol2+1:i*Iphi2(l)%ncol2,1), beta, &
                         tijdelijk1_d(l)%uit2((i-1)*Iphi2(l)% &
                         nrow2+1:i*Iphi2(l)%nrow2,1))
 end do

do i = 1, T(l+1)
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         Iphi2(l)%nrow2, Iphi2(l)%ncol2, Iphi2(l) &
                         %nnz2, alpha, descrA, Iphi2_d(l)%val, &
                         Iphi2_d(l)%JA, Iphi2_d(l)%jndx, &
                         tijdelijk1_d(l)%uit((i-1)*Iphi2(l)% &
                         ncol2+1:i*Iphi2(l)%ncol2,2), beta, &
                         tijdelijk1_d(l)%uit2((i-1)*Iphi2(l)% &
                         nrow2+1:i*Iphi2(l)%nrow2,2))
end do

do i = 1, T(l+1)
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         Iphi2(l)%nrow2, Iphi2(l)%ncol2, Iphi2(l) &
                         %nnz2, alpha, descrA, Iphi2_d(l)%val, &
                         Iphi2_d(l)%JA, Iphi2_d(l)%jndx, &
                         tijdelijk1_d(l)%uit((i-1)*Iphi2(l)% &
                         ncol2+1:i*Iphi2(l)%ncol2,3), beta, &
                         tijdelijk1_d(l)%uit2((i-1)*Iphi2(l)% &
                         nrow2+1:i*Iphi2(l)%nrow2,3))
end do

do i = 1, T(l+1)
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         Itheta2(l)%nrow2, Itheta2(l)%ncol2, &
                         Itheta2(l)%nnz2, alpha, descrA, &
                         Itheta2_d(l)%val, Itheta2_d(l)%JA, &
                         Itheta2_d(l)%jndx, tijdelijk1_d(l)% &
                         uit2((i-1)*Itheta2(l)%ncol2+1:i*Itheta2 &
                         (l)%ncol2,1),beta, w_incoming_d &
                         (l)%vector((i-1)*Itheta2(l)%nrow2 &
                         +1:i*Itheta2(l)%nrow2,1))
end do

do i = 1, T(l+1)
 status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                         Itheta2(l)%nrow2, Itheta2(l)%ncol2, &
                         Itheta2(l)%nnz2, alpha, descrA, &
                         Itheta2_d(l)%val, Itheta2_d(l)%JA, &
                         Itheta2_d(l)%jndx, tijdelijk1_d(l)% &
                         uit2((i-1)*Itheta2(l)%ncol2+1:i*Itheta2 &
                         (l)%ncol2,2),beta, w_incoming_d &
                         (l)%vector((i-1)*Itheta2(l)%nrow2 &
                         +1:i*Itheta2(l)%nrow2,2))
end do
```

```
    do i = 1, T(l+1)
     status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                             Itheta2(l)%nrow2, Itheta2(l)%ncol2, &
                             Itheta2(l)%nnz2, alpha, descrA, &
                             Itheta2_d(l)%val, Itheta2_d(l)%JA, &
                             Itheta2_d(l)%jndx, tijdelijk1_d(l)% &
                             uit2((i-1)*Itheta2(l)%ncol2+1:i*Itheta2 &
                             (l)%ncol2,3),beta, w_incoming_d &
                             (l)%vector((i-1)*Itheta2(l)%nrow2 &
                             +1:i*Itheta2(l)%nrow2,3))
    end do

    istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
         (l+1)%vector(:,1), 1 , w_incoming_d(l)%vector(:,1), 1)
    istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
         (l+1)%vector(:,2), 1 , w_incoming_d(l)%vector(:,2), 1)
    istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
         (l+1)%vector(:,3), 1 , w_incoming_d(l)%vector(:,3), 1)
   end do

! incoming waves
   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                           V2%nrow, V2%ncol, V2%nnz, alpha, &
                           descrA, V2_d%val, V2_d%JA, V2_d%jndx, &
                           w_incoming_d(lmax-1)%vector(:,1), beta, &
                           incoming_d%vector(:,1))

   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                           V2%nrow, V2%ncol, V2%nnz, alpha, &
                           descrA, V2_d%val2, V2_d%JA, V2_d%jndx, &
                           w_incoming_d(lmax-1)%vector(:,2), beta, &
                           incoming_d%vector(:,2))

   status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                           V2%nrow, V2%ncol, V2%nnz, alpha, &
                           descrA, V2_d%val3, V2_d%JA, V2_d%jndx, &
                           w_incoming_d(lmax-1)%vector(:,3), beta, &
                           incoming_d%vector(:,3))

   istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,2), &
         1 , incoming_d%vector(:,1), 1)

   istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,3), &
         1 , incoming_d%vector(:,1), 1)

   istat = cublasCcopy_v2(h1, V2%nrow, incoming_d%vector(:,1), &
         1, Axfar_d%vector(:,1),1)

! final vector
   istat = cublasCaxpy_v2(h1, V2%nrow, alpha, Axnear_d, &
         1, Axfar_d%vector(:,1), 1)
  end do

  status = cusparseDestroy(h)
```

```
! Send solution vector back to CPU.
 Axfar%vector(:,1) = Axfar_d%vector(:,1)

End Program MLFMA
```

D.7. MLFMA Storing Matrices in Group Within Wave Structure

```
Program MLFMA
use Sort
use cudafor
use cublas
use cusparse
implicit none

! Matrices are read from files and saved in the CRS format here.

 do i = 1,nearmatrix%ncol
  x(i) = cmplx(0.0,0.0)
 end do

 x(333) = cmplx(1.0,0.0)

 alpha = cmplx(1.0,0.0)
 beta = cmplx(0.0,0.0)

! Send information about the matrices to the GPU.
 nearmatrix_d%val = nearmatrix%val(1:nearmatrix%nnz)
 nearmatrix_d%JA = nearmatrix%JA(1:nearmatrix%nrow+1)
 nearmatrix_d%jndx = nearmatrix%jndx(1:nearmatrix%nnz)
 x_d = x(1:nearmatrix%ncol)

 V1_d%val = V1%val(1:V1%nnz)
 V1_d%val2 = V1%val2(1:V1%nnz)
 V1_d%val3 = V1%val3(1:V1%nnz)
 V1_d%JA = V1%JA(1:V1%nrow+1)
 V1_d%jndx = V1%jndx(1:V1%nnz)

 V2_d%val = V2%val(1:V2%nnz)
 V2_d%val2 = V2%val2(1:V2%nnz)
 V2_d%val3 = V2%val3(1:V2%nnz)
 V2_d%JA = V2%JA(1:V2%nrow+1)
 V2_d%jndx = V2%jndx(1:V2%nnz)

 do l = lmax-1, lmin, -1
  Itheta1_d(l)%val = Itheta1(l)%val(1:Itheta1(l)%nnz)
  Itheta1_d(l)%JA = Itheta1(l)%JA(1:Itheta1(l)%nrow+1)
  Itheta1_d(l)%jndx = Itheta1(l)%jndx(1:Itheta1(l)%nnz)

  Itheta2_d(l)%val = Itheta2(l)%val(1:Itheta2(l)%nnz)
  Itheta2_d(l)%JA = Itheta2(l)%JA(1:Itheta2(l)%nrow+1)
  Itheta2_d(l)%jndx = Itheta2(l)%jndx(1:Itheta2(l)%nnz)

  Iphi1_d(l)%val = Iphi1(l)%val(1:Iphi1(l)%nnz)
  Iphi1_d(l)%JA = Iphi1(l)%JA(1:Iphi1(l)%nrow+1)
  Iphi1_d(l)%jndx = Iphi1(l)%jndx(1:Iphi1(l)%nnz)
```

```
  Iphi2_d(l)%val = Iphi2(l)%val(1:Iphi2(l)%nnz)
  Iphi2_d(l)%JA = Iphi2(l)%JA(1:Iphi2(l)%nrow+1)
  Iphi2_d(l)%jndx = Iphi2(l)%jndx(1:Iphi2(l)%nnz)

  A1_d(l)%val = A1(l)%val(1:A1(l)%nnz)
  A1_d(l)%JA = A1(l)%JA(1:A1(l)%nrow+1)
  A1_d(l)%jndx = A1(l)%jndx(1:A1(l)%nnz)

  A2_d(l)%val = A2(l)%val(1:A2(l)%nnz)
  A2_d(l)%JA = A2(l)%JA(1:A2(l)%nrow+1)
  A2_d(l)%jndx = A2(l)%jndx(1:A2(l)%nnz)
 end do

 do l = lmax, lmin, -1
  T_d(l)%val = T(l)%val(1:T(l)%nnz)
  T_d(l)%JA = T(l)%JA(1:T(l)%nrow+1)
  T_d(l)%jndx = T(l)%jndx(1:T(l)%nnz)
 end do

 istat = cublasCreate(h1)
 status = cusparseCreate(h)
 status = cusparseCreateMatDescr(descrA)
 status = cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL)
 status = cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE)

 do ij = 1, it
! near matrix
  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          nearmatrix%nrow, nearmatrix%ncol, &
                          nearmatrix%nnz, alpha, &
                          descrA, nearmatrix_d%val, nearmatrix_d%JA, &
                          nearmatrix_d%jndx, x_d, beta, Axnear_d)

! expand waves
  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                          V1_d%val, V1_d%JA, V1_d%jndx, x_d, beta, &
                          v_begin_d%vector(:,1))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                          V1_d%val2, V1_d%JA, V1_d%jndx, x_d, beta, &
                          v_begin_d%vector(:,2))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                          V1%nrow, V1%ncol, V1%nnz, alpha, descrA, &
                          V1_d%val3, V1_d%JA, V1_d%jndx, x_d, beta, &
                          v_begin_d%vector(:,3))

  istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,1), 1, &
        v_outgoing_d(lmax)%vector(:,1),1)
  istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,2), 1, &
        v_outgoing_d(lmax)%vector(:,2),1)
  istat = cublasCcopy_v2(h1, V1%nrow, v_begin_d%vector(:,3), 1, &
        v_outgoing_d(lmax)%vector(:,3),1)
```

```fortran
! interpolate waves
  do l = lmax-1, lmin, -1
   status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            Itheta1(l)%nrow, 3, Itheta1(l)%ncol, &
                            Itheta1(l)%nnz, alpha, descrA, &
                            Itheta1_d(l)%val, Itheta1_d(l)%JA, &
                            Itheta1_d(l)%jndx, v_outgoing_d &
                            (l+1)%vector, Itheta1(l)%ncol, beta, &
                            tijdelijk_d(l)%uit, Itheta1(l)%nrow)

   status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            Iphi1(l)%nrow, 3, Iphi1(l)%ncol, &
                            Iphi1(l)%nnz, alpha, descrA, &
                            Iphi1_d(l)%val, Iphi1_d(l)%JA, &
                            Iphi1_d(l)%jndx, tijdelijk_d(l) &
                            %uit, Iphi1(l)%ncol, beta, &
                            tijdelijk_d(l)%uit2, Iphi1(l)%nrow)

   status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            A1(l)%nrow, 3, A1(l)%ncol, A1(l)%nnz, &
                            alpha, descrA, A1_d(l)%val, &
                            A1_d(l)%JA, A1_d(l)%jndx, tijdelijk_d &
                            (l)%uit2, A1(l)%ncol, &
                            beta, v_outgoing_d(l)%vector, A1(l)%nrow)
  end do

! transfer waves
  do l = lmax, lmin, -1
   status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            T(l)%nrow, 3, T(l)%ncol, T(l)%nnz, alpha, &
                            descrA, T_d(l)%val, T_d(l)%JA, T_d(l) &
                            %jndx, v_outgoing_d(l)%vector, T(l)%ncol, &
                            beta, v_incoming_d(l)%vector, T(l)%nrow)
  end do

  istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
        %vector(:,1), 1, w_incoming_d(lmin-1)%vector(:,1),1)
  istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
        %vector(:,2), 1, w_incoming_d(lmin-1)%vector(:,2),1)
  istat = cublasCcopy_v2(h1, T(lmin)%nrow, v_incoming_d(lmin) &
        %vector(:,3), 1, w_incoming_d(lmin-1)%vector(:,3),1)

! anterpolate waves
  do l = lmin, lmax-1
   status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE,  &
                            A2(l)%nrow, 3, A2(l)%ncol, &
                            A2(l)%nnz, alpha, descrA, A2_d(l)%val, &
                            A2_d(l)%JA, A2_d(l)%jndx, w_incoming_d &
                            (l-1)%vector, A2(l)%ncol, &
                            beta, tijdelijk1_d(l)%uit, A2(l)%nrow)

   status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                            Iphi2(l)%nrow, 3, Iphi2(l)%ncol, &
                            Iphi2(l)%nnz, alpha, descrA, &
                            Iphi2_d(l)%val, Iphi2_d(l)%JA, Iphi2_d &
                            (l)%jndx, tijdelijk1_d(l)%uit, &
```

```
                                Iphi2(l)%ncol, beta, tijdelijk1_d(l)%uit2, &
                                Iphi2(l)%nrow)

    status = cusparseCcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                                Itheta2(l)%nrow, 3, Itheta2(l)%ncol, &
                                Itheta2(l)%nnz, alpha, descrA, &
                                Itheta2_d(l)%val, Itheta2_d(l)%JA, &
                                Itheta2_d(l)%jndx, tijdelijk1_d(l)%uit2, &
                                Itheta2(l)%ncol, beta, w_incoming_d &
                                (l)%vector, Itheta2(l)%nrow)

    istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
         (l+1)%vector(:,1), 1 , w_incoming_d(l)%vector(:,1), 1)
    istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
         (l+1)%vector(:,2), 1 , w_incoming_d(l)%vector(:,2), 1)
    istat = cublasCaxpy_v2(h1, Itheta2(l)%nrow, alpha, v_incoming_d &
         (l+1)%vector(:,3), 1 , w_incoming_d(l)%vector(:,3), 1)
  end do

! incoming waves
  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                               V2%nrow, V2%ncol, V2%nnz, alpha, &
                               descrA, V2_d%val, V2_d%JA, V2_d%jndx, &
                               w_incoming_d(lmax-1)%vector(:,1), beta, &
                               incoming_d%vector(:,1))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                               V2%nrow, V2%ncol, V2%nnz, alpha, &
                               descrA, V2_d%val2, V2_d%JA, V2_d%jndx, &
                               w_incoming_d(lmax-1)%vector(:,2), beta, &
                               incoming_d%vector(:,2))

  status = cusparseCcsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
                               V2%nrow, V2%ncol, V2%nnz, alpha, &
                               descrA, V2_d%val3, V2_d%JA, V2_d%jndx, &
                               w_incoming_d(lmax-1)%vector(:,3), beta, &
                               incoming_d%vector(:,3))

  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,2), &
        1 , incoming_d%vector(:,1), 1)

  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, incoming_d%vector(:,3), &
        1 , incoming_d%vector(:,1), 1)

  istat = cublasCcopy_v2(h1, V2%nrow, incoming_d%vector(:,1), &
        1, Axfar_d%vector(:,1),1)

! final vector
  istat = cublasCaxpy_v2(h1, V2%nrow, alpha, Axnear_d, 1, &
        Axfar_d%vector(:,1), 1)
 end do

 status = cusparseDestroy(h)
! Send solution vector back to CPU.
 Axfar%vector(:,1) = Axfar_d%vector(:,1)
End Program MLFMA
```