BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

GPU ACCELERATION OF ACOUSTIC FIELD PROPAGATOR

AKCELERACE AKUSTICKÉHO PROPAGÁTORU NA GPU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MARTIN LUDVÍK**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. JIŘÍ JAROŠ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

164190

Institut:	Department of Computer Systems (DCSY)
Student:	**Ludvík Martin, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Graphics and Interaction
Title:	**GPU Acceleration of Acoustic Field Propagator**
Category:	Parallel and Distributed Computing
Academic year:	2024/25

Assignment:

1. Familiarize yourself with the technique of simulating ultrasound wave propagation using an acoustic propagator.
2. Study high-performance computing techniques on graphics processing units (GPUs).
3. Learn the software development methodology used by the SC@FIT research group.
4. Propose a strategy for efficient implementation of the acoustic propagator on the GPU.
5. Implement the proposed strategy using CUDA, HIP, or OpenMP.
6. Create a set of test cases to verify the correctness and performance of your implementation.
7. Discuss the benefits of the proposed implementation for accelerating ultrasound simulations.

Literature:

- According to supervisor's advice.

Requirements for the semestral defence:
Items 1 to 4 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Jaroš Jiří, doc. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	14.5.2025

Abstract

This thesis presents the GPU acceleration of the Acoustic Field Propagator (AFP), a one-step wave simulation method commonly used in ultrasound modelling. The original MATLAB-based implementation was rewritten in C++ and CUDA to leverage the parallel processing capabilities of modern GPUs. Several AFP variants were implemented, including absorbing grid, wave wrapping cancellation, nonlinear acoustics, and propagator for heterogeneous media. The new implementation significantly improves computational performance while maintaining numerical accuracy. Additionally, support for wave wrapping cancellation was added to the nonlinear model. The results demonstrate substantial speedup reaching 100x and simultaneously reduced memory usage compared to the original code, making the solution suitable for large-scale acoustic simulations.

Abstrakt

Tato diplomová práce se zabývá akcelerací propagátoru akustického pole (AFP) pomocí GPU. AFP je jedno kroková metoda simulace číření vln, často využívaná v modelování ultrazvuku. Původní implementace v prostředí MATLAB byla přepracována do jazyka C++ s využitím technologie CUDA, čímž se podařilo využít paralelního zpracování moderních grafických karet. Bylo implementováno několik variant AFP, pro absorbující media, s využitím wave wraping cancelation, pro nelineární akustiku a heterogenních médií. Nová implementace výrazně zrychluje výpočty při zachování numerické přesnosti. Pro nelineární model byla navíc přidána podpora metody wave wraping canceling. Výsledky ukazují výrazné zkrácení doby výpočtu v některých případech až 100 násobné a úsporu paměti oproti původnímu kódu, což činí řešení vhodným pro rozsáhlé a časově náročné simulace akustického pole.

Keywords

Acoustic Field Propagator, GPU, Ultrasound, Simulation, Numeric methods, Parallel computing, CUDA

Klíčová slova

Akustický propagátor, GPU, Ultrazvuk, Simulace, Numerické metody, Paralelní výpočty, CUDA

Reference

LUDVÍK, Martin. *GPU Acceleration of Acoustic Field Propagator*. Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Jiří Jaroš, Ph.D.

GPU Acceleration of Acoustic Field Propagator

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of doc. Ing. Jiří Jaroš, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis

.....
Martin Ludvík
May 20, 2025

Acknowledgements

I would like to thank my supervisor, Jiří Jaroš, for his assistance and guidance.

Contents

List of Figures

Chapter 1

Introduction

Ultrasound has a wide range of applications, spanning from medical diagnostics and therapeutic treatments to nondestructive material testing. Classic simulations with time-domain propagation of compressional waves through a medium are computationally and time-expensive. Recent advances in high-performance computing (HPC) have dramatically expanded the scale and complexity of simulations that can be performed, especially in areas requiring intensive numerical computation such as wave propagation. Central to this development is the use of general-purpose computing on graphics processing units (GPGPU), which enables massive parallelism well-suited for the large-scale data processing and numerical transformations involved in acoustic field simulations. With the use of GPU computing, simulations that would traditionally take hours or days on conventional CPUs can now be executed significantly faster, enabling real-time feedback, parameter exploration, and high-resolution modelling.

This thesis situates itself at the intersection of HPC and wave propagation modelling by implementing a GPU-accelerated solver for the Acoustic Field Propagator (AFP). The goal is to utilise CUDA-based GPGPU techniques to achieve efficient memory usage and maximise computational throughput. The integration of HPC principles into acoustic simulation not only reduces computation time but also makes high-resolution, three-dimensional, and nonlinear models feasible for practical applications. As simulation fidelity increases in medical imaging, non-destructive testing, and material characterisation, the role of parallel computing and GPU acceleration becomes indispensable for real-world deployment and research-scale exploration.

This thesis begins with an introduction to the simulation of wave propagation and the types of simulations. The thesis continues with a detailed description of the Acoustic Field Propagator, including its various variants for nonlinear, absorbing, and heterogeneous media. Next, we present the concept of transforming the Matlab AFP into a GPU-accelerated application and introduce suitable technologies used. The developed application is presented in the following chapter. The main goal is to give insight into how the final solution was achieved and the most difficult parts.

In order to verify the correctness, accuracy, and effectiveness of the application, a set of benchmark tasks were created. The final chapter presents the results and compares them with the original MATLAB implementation. To put the results in perspective, the comparison with alternative approaches to simulate is included, specifically the comparison with the k-Wave Toolbox, which implements the full wave time domain solver accounting for heterogeneous absorbing media with nonlinear wave propagation [17].

Chapter 2

Simulation of wave propagation

Wave propagation simulation is a fundamental problem in computational physics and engineering, with applications spanning fields such as acoustics, seismology, electromagnetics, and fluid dynamics. At its core, it involves numerically modelling how waves, whether mechanical, electromagnetic, or otherwise, travel through different media over time. These simulations are essential for understanding complex physical phenomena, designing materials and structures, and predicting the behaviour of systems where analytical solutions are intractable [12].

Accurately capturing wave behaviour requires solving partial differential equations, such as the wave equation, under appropriate initial and boundary conditions. The complexity increases when the medium is heterogeneous, anisotropic, or nonlinear, or when high-frequency components and long-time integration are involved. Numerical techniques such as finite difference, finite element, and spectral methods are commonly employed, each with trade-offs in terms of accuracy, stability, and computational cost [1].

Furthermore, the rise of high-performance computing, particularly GPU acceleration, has enabled simulations at unprecedented scales and resolutions. However, this also introduces new challenges related to algorithmic efficiency, numerical precision, and memory management. Addressing these issues is crucial for the reliable and efficient simulation of wave propagation in real-world scenarios.

One of the uses of wave propagation simulations is in the medical industry.

2.1 Simulation

Multiple established methods for simulating wave propagation through a medium exist, and each is tailored to specific physical conditions and computational requirements. A widely used tool in this domain is the k-Wave toolbox for MATLAB, which is specifically designed to simulate the propagation of acoustic waves in both time and space [17]. It supports the solution of various wave equations, including linear and nonlinear acoustic formulations, and incorporates physical phenomena such as absorption and dispersion.

The k-Wave toolbox offers multiple simulation modes. In time-domain simulations, it computes the evolution of pressure and velocity fields incrementally over time until the specified simulation endpoint is reached. In contrast, frequency-domain simulations solve directly for steady-state wave fields. These simulations leverage FFT-based methods to compute spatial gradients with high accuracy, which is particularly well suited for uniform Cartesian grids [17].

2.2 Acoustic field propagator

The acoustic field propagator (AFP) is a one-step wave propagation solver. Compared to the traditional time-based simulation algorithm, AFP computes the resulting state of the acoustic field at any time in one step. This is achieved by using Green's function with the Fourier transform [18].

Green's function

A Green's function is a mathematical tool used to solve complex ordinary and partial differential equations, particularly in physics and engineering problems involving fields and waves [2].

If we have a differential equation like:

$$Lu(x) = f(x) \tag{2.1}$$

where L is a linear differential operator, and $f(x)$ is a source term, then the Green's function $G(x, x')$ satisfies:

$$LG(x, x') = \delta(x - x') \tag{2.2}$$

with $\delta(x-y)$ the Dirac delta function. We can express the solution to the original equation as:

$$u(x) = \int G(x, x')f(x')dx' \tag{2.3}$$

Strictly speaking, Green's functions are not true functions but distributions, meaning they are defined through their action when integrated against other functions.

In essence, the Green's function represents the response of a system to a point source (like the delta function input). Once this response is known, we can determine the system's response to any arbitrary input using convolution or superposition.

Derivation of propagator with Green's function

The linear wave equation for a homogeneous medium is subject to a continuous wave source defined as the term $S(x, t)$

$$S(\mathbf{x}, t) = A(\mathbf{x})e^{i(\omega_0 t+\phi(\mathbf{x}))} \tag{2.4}$$

where ω_0 is the source frequency, $A(x)$ is a spatially varying amplitude source, and $\phi(x)$ phase. With this, we can use the Green's function convolution to define the propagator in the spatial frequency domain:

$$\begin{aligned} p(\mathbf{x}, t) &= \int_{t'} \int_{\mathbf{x}'} G(\mathbf{x} - \mathbf{x}', t - t')\, S(\mathbf{x}', t')\, d\mathbf{x}'\, dt' \\ &\rightsquigarrow \frac{c_0^2}{(2\pi)^3} \int_{\mathbf{k}} \int_{\mathbf{x}'} I(\mathbf{k}, t) A(\mathbf{x}) e^{i\phi(\mathbf{x})} e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}')}\, d\mathbf{x}'\, d\mathbf{k} \end{aligned} \tag{2.5}$$

where the $\mathbf{k}$ represents wavevector [18].

Numerical solution

Next, in the equation 2.5 , the integrals over x' and $\mathbf{k}$ can be recognised as the forward and inverse Fourier transforms [18], and thus an exact expression for complex acoustic pressure at time t that can be written succinctly as:

$$p(x,t) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{A(x)e^{i\phi(x)}\right\} I(k,t)\right\} \tag{2.6}$$

where $A(x)$ is a spatially varying amplitude source and $\phi(x)$ phase. The $\mathcal{F}^{-1}$ and $\mathcal{F}$ denotes inverse and forward Fourier transform. In practice, the Fast Fourier Transform is used, which brings some challenges. Finally, $I(k,t)$ represents the core function of the acoustic field propagator [18] [19].

$$\begin{aligned} I(\mathbf{k},t) &= \left(\frac{iw_0 e^{-\alpha c_0 t}}{c_0\tilde{k}}\right)\frac{c_0\tilde{k}(e^{(i\omega_0+\alpha c_0)t} - \cos(c_0\tilde{k}t)) - (i\omega_0 + \alpha c_0)\sin(c_0\tilde{k}t)}{(c_0k)^2 + 2i\alpha c_0\omega_0 - \omega_0^2} \\ &+ \left(\frac{e^{-\alpha c_0 t}\sin(c_0\tilde{k}t)}{c_0\tilde{k}}\right) \end{aligned} \tag{2.7}$$

$$\tilde{k} = \sqrt{k^2 + \alpha^2} \tag{2.8}$$

This is the core equation of the acoustic field propagator without applying an initial ramp. The ramp is necessary because using the discrete Fourier transform can introduce Gibbs oscillations in the computed pressure field. [19]. A half-cosine ramp was chosen for the original MATLAB implementation, which smoothly varies from 0 to 1 over a period of the input signal [18]. The resulting numerical equation is much more complex, so the function notation will not be written down here.

The implementation process involves a detailed discretisation of the source distribution, utilising a uniform Cartesian grid to effectively partition the domain. This approach ensures a systematic representation of the data, facilitating accurate computational analysis and modelling within the defined space.

The original realisation of the propagator is done using MATLAB. The code utilises matrix operations that are parallelised on the CPU side. In the following subsection, each of the three versions to be converted and accelerated with CUDA will be explained, along with a description of their original MATLAB implementations.

2.2.1 Absorbing version with grid expansion

This is the first version of the acoustic field propagator targeted for conversion and acceleration using the GPU. As mentioned above, the propagator relies on the FFT to transition between the spatial and frequency domains. The main challenge with this approach is that the FFT assumes the input signal is periodic, which causes waves exiting one side of the computational domain to re-enter from the opposite side. This issue can be mitigated by appropriately zero-padding the domain to increase its size, so the wrapping does not reach the region of interest.

The size of the expanded grid is calculated based on the signal sound speed and the time at which the pressure field will be computed. To make FFT operation faster, the grid is expanded to small prime factors. Most FFT algorithms, particularly the widely used Cooley-Tukey algorithm, are optimised for these sizes.

The simulation time can be calculated to ensure that the user-defined grid reaches a steady state. For a 3D domain, this calculation is given by:

$$t = d\frac{\Delta x}{c_0}\sqrt{(N_x^2 + N_y^2 + N_z^2)} \tag{2.9}$$

where d is the scale factor, Δx is the isotropic spacing between the grid points, and Nx, Ny, and Nz represent the number of grid points in each Cartesian direction [18].

Due to the Cartesian grid, staircase errors can appear when the source geometry does not align well with the grid. However, spectral methods mitigate this issue because their bandlimited interpolant is analytically known [21].

Aforementioned wavevectors are specified based on a grid spacing of Δx. In the MATLAB code, these wavenumbers are calculated using the function *getWaveNumbers* and stored in a matrix matching the grid size for subsequent use.

The MATLAB implementation is contained in the file named after the corresponding function, *acousticFieldPropagatorAbsorbing.m*. This file includes all the essential equations needed to compute the steady-state pressure field resulting from a continuous wave input.

The core propagator function, described by Equation 2.7, along with its initial ramp version, is implemented within the function *getPropagator*, which returns the propagator values at each grid point in the form of a matrix. The function is essentially a direct translation of the mathematical equation into MATLAB code. Each component of the equation is implemented using element-wise matrix operations, which allows the function to efficiently apply the computation across the entire simulation grid. This approach leverages MATLAB's strength in handling array-based operations, ensuring both readability and performance while maintaining a close correspondence between the mathematical formulation and the code itself.

The same can be said about the process of calculating the resulting pressure field with the equation 2.6. The MATLAB code looks like this:

```
pressure = ifftn( propagator .* fftn( amp_in_ex .* exp(1i * phase_in_ex)));
```

If the function is called with two or more output arguments, the amplitude and phase are extracted from the complex pressure field.

```
amp_out = abs(pressure);
phase_out = modPhase(angle(pressure) - w0*t);
```

The *abs* function for complex numbers returns the complex magnitude of the value. The phase is obtained by subtracting the source term $w_0 t$, as defined in equation 2.4, from the phase angle. The resulting value is then wrapped to lie within the range $-\pi$ to $+\pi$.

2.2.2 Wave warping cancellation version

Wave wrapping cancellation addresses the previously mentioned issue of wave wrapping introduced by using the FFT. Earlier implementations of AFP tackled this by expanding the computational grid to ensure that any wrapping occurred beyond the user-defined region. This version solves this issue with shifted wavenumbers and the use of a phase ramp [10]. In practice, the propagator can be implemented by first applying a phase ramp of $e^{\pm i\Delta kx/4x}$ before performing the FFT. After that, the field is multiplied by the wavenumber-shifted propagator, followed by an inverse FFT, and then the phase ramp is removed. This process introduces a phase shift of $\pm\pi/2$ to the wrapped portion of the acoustic field, while the unwrapped part remains unaffected. By combining these modified

fields, the wrapped components can be effectively cancelled out. The number of required wave wrapping cancellations for each grid dimension can be calculated using this equation.

$$O_x > \lceil \frac{C_x c_0 t_{min}}{N_x \Delta_x} \rceil + 1 \tag{2.10}$$

To perform wave wrapping cancellation in a 3D grid, we need to combine every wavenumber shift in each dimension. So for a cube grid, the total number of separate fields that must be computed is O_x^3.

The MATLAB implementation also utilises the same *getPropagator* function. The first distinction lies in the *getWaveNumbers* function, where we shift the numbers in each simulated dimension based on the current shifted field being calculated.

Wrapping for each shifted wavenumber is handled iteratively by computing the corresponding shifted fields and summing them. Finally, the resulting acoustic field is averaged by dividing it by the total number of applied shifts.

2.2.3 Nonlinear acoustics version

Nonlinear acoustics is the study of how sound waves behave when their amplitudes are large enough that the linear approximations of wave propagation break down. Unlike linear acoustics, where sound waves are small, smooth, and predictable, nonlinear acoustics deals with intense wave phenomena, where the wave alters the medium it travels through, and the waveform itself changes shape over time and distance.

To address this, a nonlinear term is added to the wave equation capturing the effects of nonlinear acoustic propagation. As a result, the wave no longer oscillates solely at the source frequency but generates harmonics at integer multiples of it. To model this behaviour, the wave equation is reformulated as a system of coupled Helmholtz equations, where each equation corresponds to a different harmonic.

$$(k_n^2 + \nabla^2)\tilde{p}_n = -S + \mathcal{N}_n(\tilde{p}_1, \tilde{p}_2, \ldots), \tag{2.11}$$

The coupling between these equations arises from nonlinear interactions and must be explicitly determined. The resulting coupled Helmholtz equations look like this [9]:

$$\begin{aligned}
\left(k_1^2 + \nabla^2\right)\tilde{p}_1 &= -S + \eta(1\omega)^2(&& \tilde{p}_1^*\tilde{p}_2 + \tilde{p}_2^*\tilde{p}_3 + \tilde{p}_3^*\tilde{p}_4 + \tilde{p}_4^*\tilde{p}_5 + \ldots), \\
\left(k_2^2 + \nabla^2\right)\tilde{p}_2 &= \eta(2\omega)^2(&& \tfrac{1}{2}\tilde{p}_1\tilde{p}_1 + \tilde{p}_1^*\tilde{p}_3 + \tilde{p}_2^*\tilde{p}_4 + \tilde{p}_3^*\tilde{p}_5 + \tilde{p}_4^*\tilde{p}_6 + \ldots), \\
\left(k_3^2 + \nabla^2\right)\tilde{p}_3 &= \eta(3\omega)^2(&& \tilde{p}_1\tilde{p}_2 + \tilde{p}_1^*\tilde{p}_4 + \tilde{p}_2^*\tilde{p}_5 + \tilde{p}_3^*\tilde{p}_6 + \tilde{p}_4^*\tilde{p}_7 + \ldots), \\
\left(k_4^2 + \nabla^2\right)\tilde{p}_4 &= \eta(4\omega)^2(&& \tfrac{1}{2}\tilde{p}_2\tilde{p}_2 + \tilde{p}_1\tilde{p}_3 + \tilde{p}_1^*\tilde{p}_5 + \tilde{p}_2^*\tilde{p}_6 + \tilde{p}_3^*\tilde{p}_7 + \tilde{p}_4^*\tilde{p}_8 + \ldots), \\
\left(k_5^2 + \nabla^2\right)\tilde{p}_5 &= \eta(5\omega)^2(&& \tilde{p}_2\tilde{p}_3 + \tilde{p}_1\tilde{p}_4 + \tilde{p}_1^*\tilde{p}_6 + \tilde{p}_2^*\tilde{p}_7 + \tilde{p}_3^*\tilde{p}_8 + \tilde{p}_4^*\tilde{p}_9 + \ldots), \\
&\vdots
\end{aligned} \tag{2.12}$$

The MATLAB implementation employs an iterative sequential solver for these equations. In the first iteration, the acoustics fields are sequentially computed up to N harmonics, including the sum terms visualised in equation 2.12 with blue. The second run adds as many difference terms as possible, now visualised in magenta. The subsequent iterations add to the accuracy of each term and to the final field [4].

The number of harmonic fields computed is determined by the function parameter, but for better results, one more harmonic is included in the calculations. The entire process is memory-intensive, as it requires storing an expanded grid to handle wave wrapping. Moreover, the number of such large matrices increases with the number of harmonics being computed.

2.2.4 Heterogeneous sound speed and absorption version

With heterogeneous versions of AFP, we can model phenomena like refraction, scattering, focusing, or mode conversion that only occur due to inhomogeneity. Wave equation in Lippmann-Schwinger form with contrast source V that provides the heterogeneous part [8] is shown below.

$$\left(\frac{1}{c_0^2}\frac{\partial^2 p}{\partial t^2} - \nabla^2\right) p \quad = S + V(c_0, \rho_0, \tau)p \tag{2.13}$$

For this acoustic field propagator, we only account for sound speed contrast c_0 and absorbing contrast τ. With the use of Green's function and the Convergent Born series introduced in [11], we can solve this equation with fixed-point iteration.

Wave wrapping of the acoustic field is once again addressed by zero-padding the user-defined grid. However, it is also necessary to prevent wrapping in the introduced contrast source. To achieve this, an absorbing boundary is added beyond the user-defined grid to eliminate any waves that wrap around it.

The MATLAB function does not internally determine the number of iterations required for the Convergent Born series to converge. Instead, parameters controlling the number of iterations to be performed are provided as input. This implementation is sometimes referred to as CBS (from the use of Convergent Born Series) to keep the command and function names more concise.

2.2.5 MATLAB GPU acceleration

MATLAB provides tools to easily accelerate the code on GPUs. One of these tools is the operations based on gpuArray. GpuArray allows for storing and operating on arrays directly on the GPU instead of the CPU. This enables massive parallelism and hardware acceleration, taking advantage of thousands of GPU cores. The process of converting MATLAB operations and arrays into a GPU array involves several simple steps. First, we need to transfer or allocate the data in the array on the GPU. To achieve this, we must use the gpuArray function [14]:

```
A = gpuArray(rand(1000));
```

Once the data is on the GPU, MATLAB automatically runs supported operations that are coded to be performed with the data, using GPU-accelerated libraries, such as:

- cuBLAS for linear algebra
- cuFFT for fast Fourier transforms
- Thrust/CUDA for element-wise and reduction operations.

```
B = fft(A);         Runs cuFFT on GPU
C = A .* B;  Element-wise GPU kernel
```

Computed data stays on the GPU VRAM. To transfer them back to CPU host memory, we can use the gather() function.

```
A = gather(C);
```

This is what I have done with some original implementations to evaluate performance differences with my CUDA rewrite.

Chapter 3

Program design and technical approach

The main goal of this thesis is the acceleration of an existing MATLAB code of three versions of AFP. To use the full power of the GPU, the kernels for the graphics card will be written in CUDA for Nvidia graphics. This will ensure that we can use every available feature of Nvidia GPUS. Alternatives like OpenCL or OpenACC are viable, but can compare in some cases to dedicated languages like CUDA.

3.1 Application design

The program is designed to be run from the command line or terminal while data is loaded using the hierarchical data format HDF5. For ease of use, the resulting implementation will allow the use of a MATLAB MEX C++ function. This will be achieved by a CMake switch that chooses between compiling for a standalone application and a mex64 file for MATLAB.

To make the process of converting code from MATLAB to CUDA easier, I have decided to make a CPU C++ version of each version of AFP first. This way, debugging and finding errors in the conversion process will be easier. The second advantage is the chance of comparing the performance between CPU and GPU implementations in C++ and CUDA, respectively. Since the primary goal of the implementation is to accelerate the AFP on the GPU, the CMake switch **GPU_BUILD** can be used to compile only the GPU version of the program. This removes the dependency on the Intel MKL FFT library, which is used for CPU FFT operations.

It is important to be able to choose the precision of the calculation. This helps reduce the memory requirement of the GPU device. For this reason, the app will be able to launch with both single and double precision. From a design point of view, this will be achieved with the use of C++ template classes. Kernel classes will be templated with precision and simulation arguments. Each version of AFP will have its own class inherited from the base Kernel class. This will help reduce the repetition of code and support the addition of new models in the future.

The result will be an executable program with switches as arguments, which the user can use to choose the parameters of the simulation and which version of AFP will be run.

The available arguments are outlined below:

- -i inputFile.hdf5: input file, the structure will be described below

- -o outputFile.hdf5: path and name of output file
- -d 3: dimensionality of input file and simulation
- -m noexpansion: selection of AFP version - absorbing, noexpansion, nonlinear, cbs
- -v quick: select AFP model version: quick, balanced, lowmemory, default is balanced
- -n 60: sets the number of iterations for the nonlinear and CBS model, default is 1
- -f 5: sets the number of harmonics for the nonlinear model, default is 5
- -p double: precision of variables and math functions for the whole simulation, two options are available - single/double
- -c: switch between GPU and CPU implementation, if this argument is present, the CPU version will be run
- -e: Run nonlinear model with wave warping cancellation using expansion of the acoustic field grid, if not present, runs with shift cancellation
- -h: display help page

Parameters i, o, d, p, and m are mandatory and must be present when running the application from the command line or terminal.

3.2 CMake

CMake is an open-source, cross-platform build system generator designed to streamline the compilation process across different operating systems and compilers. By using a platform-independent configuration file (CMakeLists.txt), it generates native build files, such as Makefiles or Visual Studio project files, ensuring that the build process remains consistent and portable across various environments.

Purpose and Usage of CMake:

- Cross-platform Build Configuration: CMake allows developers to write one set of build instructions that can be used on Linux, Windows, macOS, and more.
- Compiler and Toolchain Abstraction: It abstracts the details of the compiler and build tools, making it easier to write portable build scripts.
- Dependency Management: CMake can detect libraries, find packages, and manage dependencies through commands like *find_package()* and *target_link_libraries()*.
- Project Organisation: It helps structure large projects by supporting out-of-source builds, modularisation, and hierarchical subprojects.
- Build File Generation: CMake generates build system files tailored to the platform: GNU Makefiles (make), Ninja build files, Visual Studio solution files, Xcode project files, etc.

For this project, CMake is used to find packages of required libraries, switch between targets of the resulting application, and choose a library linking style - static and shared.

3.3 CUDA

As mentioned before, the GPU code will be written in CUDA. This section describes some essential components of the toolbox.

Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface for NVIDIA graphics processing units (GPUS) [20]. When developing a C++ application, we can use the custom language C++ CUDA that can be easily compiled with the use of the nvcc compiler to Parallel Thread Execution (PTX). PTX is an intermediate assembly code that is then converted to executable binary code.

In CUDA terminology, system memory (RAM) is referred to as host memory, while the GPU's memory space is known as device memory.

CUDA is comprised of several libraries that are focused on accelerating a plethora of operations. One of these libraries, called cuFFT, focuses on accelerating the Fast Fourier transform (FFT). As said before, the AFP uses the Fourier transform for its simulations, so the use of this specialised and already accelerated library is necessary. The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets. The cuFFT interface in C++ provides a straightforward function call, using a handle that encapsulates information about the input data and the desired operation. Before executing the computation of the FFT, we must initialise the library and relay information about the input and output. After that, we only need to call simple functions with the provided data.

3.3.1 CUDA Graph

Modern GPUs are incredibly fast, often executing individual operations, such as kernel launches or memory transfers, in just microseconds. However, the overhead associated with dispatching each of these operations to the GPU, also on the microsecond scale, has become increasingly impactful in many scenarios. Real-world applications typically involve a high volume of GPU operations, often organised into many iterations. When these operations are individually dispatched and executed quickly, the cumulative overhead can significantly hinder overall performance.

CUDA Graph, as the name suggests, is a tool that helps with the execution of GPU operations by using a graph structure to optimise their execution times. Thanks to the graph nature, this approach can provide easy parallelism for CUDA operations. Another advantage is the reduction of needed CPU interruptions.

There are two approaches to the process of constructing this graph. The first option is to capture submitted CUDA operations in specific streams, which are not gonna be executed, but only recorded as graph nodes. This approach is good for applications that are already written in CUDA and that we want to execute using a graph.

The second option is creating the graph nodes directly for specific CUDA operations. The advantages of this approach are the manual customisation of nodes and their dependencies.

This manual approach is used in two versions of the AFP when there are iterations involved.

3.4 HDF5

One way of loading and saving data is with files using the hierarchical data format HDF5. HDF5, or Hierarchical Data Format version 5, is an open-source file format and software library developed by the HDF Group. It is designed to store and manage large and complex datasets. HDF5 is particularly popular in fields like scientific computing, engineering, and high-performance data applications due to its efficiency and flexibility [7].

HDF5 organises data in a hierarchical structure that is similar to a file system. This structure includes groups, which function like directories, and datasets, which act like files holding multidimensional data arrays. This design allows for the logical and scalable organisation of complex data.

One of the strengths of HDF5 is its ability to store large amounts of data efficiently. It supports features such as data compression, chunking (breaking data into manageable blocks), and partial input/output, allowing applications to read or write only the needed portions. These capabilities make HDF5 appropriate for high-performance computing and big data applications.

The format is platform-independent and self-describing, meaning that HDF5 files can be transferred between systems and used by different applications without compatibility problems. The library supports a wide range of data types, from basic numeric and string types to complex, user-defined data structures. Additionally, metadata can be stored directly with the data using attributes, which are small pieces of descriptive information attached to datasets or groups.

This functionality is implemented with the use of modified solutions from the previous project in the k-Wave toolbox, acousticFieldPropagator-OMP.

Through the classes *Hdf5Input* and *Hdf5Output*, we can handle input and output files with data for the propagator. The solution handles opening files and the management of file datasets. For the purposes of this project, the solution must be modified to support the template nature of the program solution. The function for loading the parameters file dataset uses templates for precision and simulation dimensions. Templates allow parameter instances to be passed directly to the function.

3.5 File format

The definition of the hierarchical file format must be standardised so that the creation of files can be implemented in other solutions that use this accelerated AFP program. The table 3.1 shows the structure of the input file for AFP nonlinear versions. The absorbing version with and without expansion and heterogeneous requires different input file structures. This version does not need the two parameters for nonlinear AFP bonA and rho0. In the case of the heterogeneous AFP, sound speed and absorption datasets are expected to be Cartesian matrices with respective values for each input grid point.

Dataset name	Description	Units
alpha	Absorption coefficient	[dB/cm]
amp_in	Cartesian matrix with pressure amplitudes of the input medium	[Pa]
bonA	Acoustic nonlinearity parameter (Beyer's parameter)	
speed	Speed of sound in the input medium	[m/s]
dx	Grid point spacing (same for each axis of the matrix)	[mm]
phase_in	Cartesian matrix containing the pressure phase of source	[rad]
rho0	Medium density (homogenous)	$[kg/m^3]$
sz_ex	Size of the input domain with recalculated expansion	
t	Time for which the pressure field is calculated	[s]
f0	Source frequency (scalar value)	[rad/s]

Table 3.1: Structure of input HDF5 file

Attributes of the root group are needed, and some are optional.

Attribute name	Description	Expected value
description	Description of input data	N/A
file_type	Type of data	afp_input
major_version	Major file format version	1
minor_version	Minor file format version	1

Table 3.2: Group attributes of input files

Finally, the output structure of the HDF5 file is described below in the 3.3 table. The output can be separated into pressure amplitude and phase or represented in complex form. The complex dataset is named **pressure_out**, and the real part represents the cosine excitation of the pressure field, and the imaginary part represents the sinus excitation.

Dataset name	Description	Units
amp_out	Cartesian matrix with pressure amplitude	[Pa]
phase_out	Cartesian matrix with pressure phase	[rad]
presure_out	Complex matrix with pressure for cosine and sine excitation	[Pa]

Table 3.3: Structure of output HDF5 file

To be sure that the user input file is compatible with the AFP version, the *file_type* in Table 3.1 is different for each. The major and minor file formats are also different from previous HDF5 file formats in the k-Wave toolbox.

3.6 MATLAB C++ MEX function

A MATLAB C++ MEX function is a way to integrate C++ code into MATLAB, allowing the execution of high-performance C++ routines directly from the MATLAB environment. MEX stands for MATLAB Executable. A MEX function is a C, C++, or Fortran program compiled into a binary that MATLAB can call just like any other built-in function. The main benefit of using MEX functions is the significantly increased speed of computationally

intensive operations compared to MATLAB implementation. Access to system-level or low-level features not directly available in MATLAB.

A C++ MEX function is defined by creating a class named MexFunction, which inherits from matlab::mex::Function. This class overrides the function call operator operator(), enabling instances of MexFunction to be invoked like regular functions.

When the MEX function is called from MATLAB, it creates an instance of this function object, allowing it to retain state over multiple calls to the same MEX file [13]. The standard structure of a C++ MEX function involves subclassing matlab::mex::Function in a class explicitly named MexFunction, with operator() overridden to define the function's behaviour.

Algorithm 1: Design of MEX C++ Format

```
1: class MexFunction: public matlab::mex::Function
2: {
3:     public:
4:     void operator()
5:         (matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs)
6:     {
7:         Function implementation
8:     }
9: };
```

As seen in Algorithm 1, the way data is transferred between MATLAB and C++ Mex class is through object *matlab::mex::ArgumentList*. ArgumentList is a wrapper enabling iteration over the underlying collections holding the input and output data, with the information on the data types.

Chapter 4

Implementation

This chapter outlines several important and interesting aspects of the program implementation. The technology mentioned in the previous chapter was used to implement the resulting application. The conversion from MATLAB code to a GPU-accelerated implementation in CUDA is outlined in the first part of this chapter, followed by a block out of the structure of C++ classes and their methods. Finally, the description of the steps that were taken to accelerate the already converted code to CUDA.

The implementation requires some libraries and a compiler. The project is designed with a C++ 17 version, but can be compiled as a C++ 20 project without issues. The Intel OpenAPI C++ compiler is preferred on Windows because it provides better support for the OpenMP API than Microsoft MSVC. For Linux, the same Intel compiler can be used. It was also tested to work with GCC 12.3.

For compilation, these libraries are required:

- CUDA Toolkit: Developed on version 12.6, also tested with 12.8
- HDF5: Developed with 1.13.2 version
- oneAPI: Math Kernel Library (oneMKL) is required, C++ Compiler is recommended for Windows systems. Developed with 2025.0 version, tested with 2022a
- CMake: For building, it is recommended to use the Ninja build system. Minimum required version 3.26.

4.1 Conversation from MATLAB

The first step before converting any of the three AFP versions to C++ and subsequently to CUDA was to profile the original MATLAB code. This was done with the MATLAB built-in profiler. The profiler provides a line-by-line percentage of time spent from the total program runtime. This gives an easy overview of the parts of the script that take the most time to process.

Line Number	Code	Time(s)	% Time
450	propagator = getPropagator(k, w0, c0, ...	14.955	79.5%
490	pressure = ifft(propagator .* fft(...	3.054	16.2%
481	kx = getwavenumbers(numDim(amp_in) ...	0.427	2.3%
501	pressure = pressure(1:size(amp_in,1) ...	0.276	1.5%
All other lines		0.09	0.5%
Totals		18.803	100%

Table 4.1: Result of profiled original Absorbing AFP, with top 4 time-consuming lines

The first profiled MATLAB AFP function is for an absorbing medium with grid expansion to prevent wave wrapping. The table shows that the function *getPropgator* is the most time-consuming operation.

Line Number	Code	Time(s)	% Time
487	propagator = getPropagator(k, w0, c0, ...	231.061	94.8%
503	pressure = ifft(propagator .* fft(p_in ...	5.563	2.3%
491	inv_ramp = exp(1i * dkx * shift_x * x) ...	2.780	1.1%
490	ramp = exp(-1i * dkx * shift_x * x) .* ...	2.772	1.1%
487	k = getWavenumbers(numDim(p_in), sz, ...	1.215	0.5%
All other lines		0.399	0.2%
Totals		243.790	100%

Table 4.2: Result of profiled original No Expansion AFP, with top 5 time-consuming lines

Once again, profiling the code for the wave wrapping cancellation version reveals that the majority of the computation time is spent on the core function of the acoustic field propagator, as defined in equation 2.7, specifically the line 487. This indicates that the main performance bottleneck lies in the AFP core function itself, which is a candidate for acceleration, rather than in operations like the FFT, which are already optimised through existing libraries. With this information, the main focus of the acceleration is on reducing the time spent on this part of the propagator.

Line Number	Code	Time(s)	% Time
491	propagator = getPropagator(k, w0, c0, t ...	27.439	58.3%
513	pressure_ex(:,:,:,harmonic_loop) = ifftn(...	6.428	13.7%
457	source_ex = source_ex + 0.5 * ...	6.008	12.8%
464	source_ex = source_ex + conj(...	5.830	12.4%
477	source_ex = source_ex .* AFP_scaling .* ...	0.295	0.6%
All other lines		1.077	2.3%
Totals		47.077	100%

Table 4.3: Result of profiled original Non-linear version of AFP, with top 5 time-consuming lines

The acoustic field propagator with nonlinear acoustics is similar to the first version in that, most of the time, it is required for propagator calculation. But it is only approximately 53% of the total time. The remaining time is spent in harmonic sum and difference from

the equation 2.12 and multiplication of the propagator with previous values and forward and inverse FFT.

Line Number	Code	Time(s)	% Time
495	p_out = ifftn(propagator .* fftn(...	170.859	66.6%
498	p_latest = p_latest - (1i * V ./ epsilon) ...	44.071	17.2%
503	total_source = scaling_AFP .* V .* ...	22.377	8.7%
478	propagator = getPropagator(k, w0, c0, ...	15.177	5.9%
487	original_source = amp_in_ex .* exp(...	0.639	0.2%
All other lines		3.531	1.4%
Totals		256.654	100%

Table 4.4: Result of profiled original Heterogeneous version of AFP, with top 5 time-consuming lines

Finally, the last version with the support for heterogeneous sound speed and absorption of the input medium, is the result of the MATLAB profile in 4.4. Compared to the previous two versions, this is computed using a high number of iterations. However, the core propagator function is computed only once, and in each iteration, propagator values are only multiplied by the current acoustic field values. The profiler showed that 66% of the time is spent with forward and inverse Fast Fourier Transform. Another 25% is dedicated to iteration operations, which take care of sums to the total result. The time spent on precomputing the propagator only takes 6%, while running 120 iterations. This means that only the operations in each iteration can bring significant time reduction. Of course, the CUDA library cuFFT accelerates the FFT operation compared to the original MATLAB CPU version, but cannot be further optimised.

4.1.1 Kernel class design

The core class of the application is named Kernel, and its purpose is to run the simulation. The hierarchy of inherited classes from Kernel looks like this:

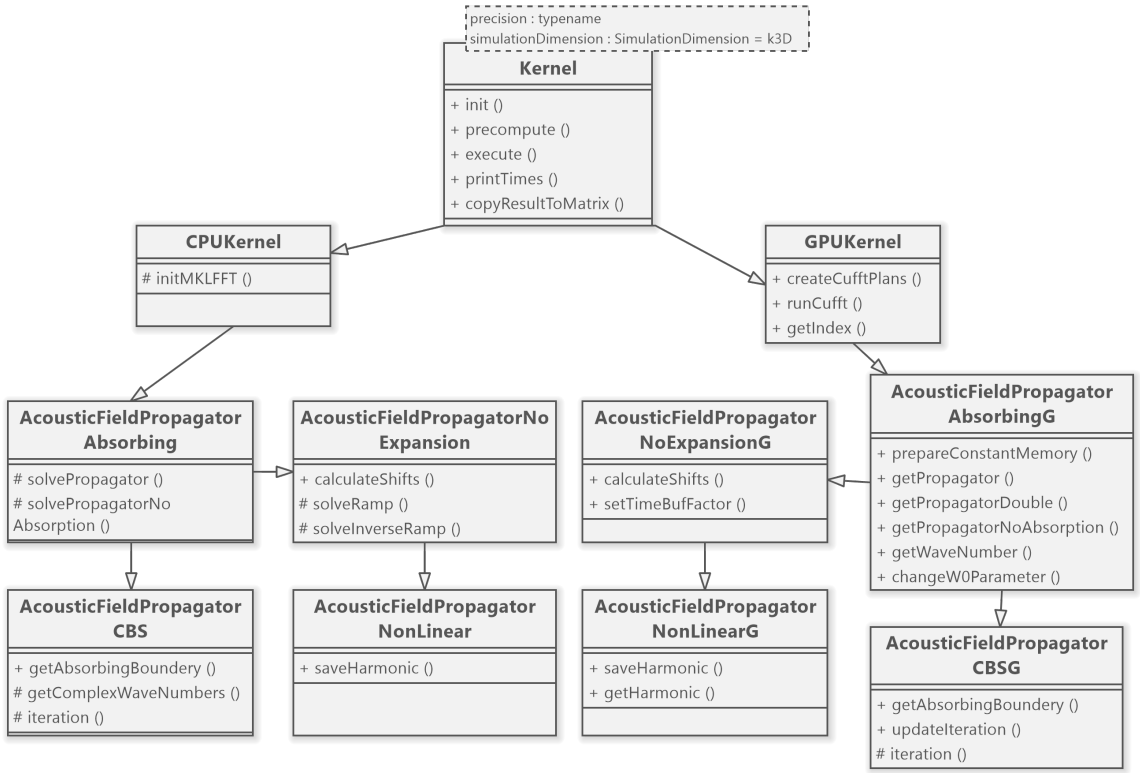

Figure 4.1: Diagram showing the class hierarchy of Kernel and its derived classes.

All classes inherited from Kernel use template arguments for precision and simulation dimension. This enables easy implementation of a switch between different precisions throughout the whole process of the propagator. As mentioned before, the code from MATLAB was first converted to C++ CPU version; therefore, the two classes that directly inherited from the Kernel class are CPUKernel and GPUKernel versions of that class. These are abstract classes that are not meant to be instantiated directly, but rather serve as base classes for specific AFP implementations.

The constructor of each Kernel class expects a parameter containing an instance of the singleton class Parameters, which contains simulation parameters.

In Figure 4.1, important methods in respective classes are mentioned. In the following sections, some of these will be described in detail.

All Kernel methods are virtual, except for the copyResultMatrix() method. They are implemented differently for each version to facilitate the easy creation of variant solutions and allow for future expansion. The mentioned function *copyResultMatrix()* is universal and used for every AFP version. It takes care of final modification and preparation before the data is written into the output file or array. One of the tasks this function performs is trimming the grid back to its original size for AFP versions that use expansion of the grid to prevent wrapping of the waves. Because all data inside the program is in a one-dimensional array, the indexing is accomplished through nested for loops.

Algorithm 2: Idexing of back trimming to original grid size

```
for x=0 to x < outSizeX do
    for y=0 to y < outSizeY do
        for z=0 to z < outSizeZ do
            outArray[x * outSizeY * outSizeZ + y * outSizeZ + z] =
            inputArray[y * inputSizeZ + x * inputSizeZ * inputSizeY + z];
        end
    end
end
```

The previously mentioned virtual methods are designed to provide a unified framework for executing all versions of the AFP. The virtual method *init()* is responsible for initialising version-specific variables and also sets up the appropriate FFT library, Intel oneMKL for CPUKernel and cuFFT for GPUKernel. Additionally, this function takes an instance of the TerminalPrint class, which is used to display progress updates in the console.

The next virtual method, *precompute()*, is used to prepare all necessary data required for executing the core AFP calculations. Measured times of parts of execution should be printed to the terminal using method *printTimes()*, overridden in each AFP kernel class.

Finally, the *execute()* method handles the actual execution of the AFP kernels for the specific implementation.

4.2 Supporting classes

The need for a separate class for each part of the propagator stems from the use of precision as a template argument. Another class used in AFP are Parameters, Matrix, TypedTuple and TerminalPrint. TypedTuple is a modified utility from the earlier-mentioned project acousticFieldPropagator-OMP. The main change is compatibility with the template argument for the simulation dimension. The second change is an added method for returning the diagonal of coordinate values.

Parameters

The Parameters class is designed to store all parameters during the program's runtime and parse application arguments from the terminal/command line. Template arguments are the same as for the Kernel class and every other class.

To support constant memory usage for CUDA kernels, the GPUParameters structure contains all necessary variables for calculating the propagator. Variables are templated with the precision argument. The structure's variables cannot utilise a dynamic allocator for CUDA constant GPU memory. The Propagator uses *cuda::std::complex* library for operations on complex numbers that have a dynamic allocator. So complex variables are represented as an array of two values (either floats or doubles). To use the constant memory structure as is, I had to create a custom class inherited from the CUDA complex one. This is necessary to allow the two float/double values to be easily cast to the *cuda::std::complex* representation and used in computations.

Matrix

The Matrix class serves the purpose of being a data wrapper and allocation tool for acoustic field grid points. The class can contain the host and device memory and copy between them through GPUKernel methods. The *alloc()* method is used for host memory allocation, and it allows the allocation of both pinned and pageable memory. Pinned memory is useful for CUDA kernels because it improves the speed at which the data from system memory can be transferred to device memory.

TerminalPrint

The class provides terminal/command prompt print functions for easy and formatted progress reporting. Methods like *printParameter()* format the input variable value and print it to the selected *std::ostream*. This function is templated, so it allows specialisation of specific variable types and the automated creation of others. One example showing the specialisation for this function is for the variable Size, which contains the size of the acoustic field grid for each simulation dimension, as seen in Figure 4.2 Domain size parameter.

```
======================================================================
 Acoustic Field Propagator - Absorbing No expansion
----------------------------------------------------------------------
Model version:                                              Balanced
Computational precision:                                      Single
Simulation dimension                                               3
Domain size:                                    64 x 64 x 64 points
======================================================================
 Progress:
----------------------------------------------------------------------
Reading input file...
Simulation time:                                          8.33121e-06
Preprocesing...
Executing...
Saving result to output file...
======================================================================
 Execution time:
----------------------------------------------------------------------
Load time:                                                  4.0000 ms
Execution time:                                            10.0000 ms
  -FFT Time:                                                1.2572 ms
  -Solve propag Time:                                       1.7511 ms
  -IFFT Time:                                               0.7590 ms
  -Write to Matrix Time:                                    0.1997 ms
  -Ramp Time:                                               1.4783 ms
  -Inverse ramp Time:                                       0.6106 ms
precompute:                                                 2.2727 ms
Write time:                                                10.0000 ms
----------------------------------------------------------------------
Total time:                                               144.0000 ms
======================================================================
```

Figure 4.2: Example of program terminal output

Program entry point

The program has different entry points that are decided based on the CMake argument **-DBUILD_MATLAB**. If this argument is not used, the resulting build files are generated for a stand-alone application. In this configuration, the first step is parsing application arguments to determine which template arguments will be used for the classes. The function *runAFP()* is located in the main file and utilises two template arguments, which allow the variable type for precision and the dimensionality of the simulation to be defined only once. After this, the data from the input files is loaded to create the Parameters object with simulation parameters.

The second option is the aforementioned MATLAB C++ Mex function. The entry here must be implemented in a precise manner to comply with the strict definition of

MEX function. It is called a function because the resulting binary file after compilation is called with a function named after the file. But in C++, the entry point is overridden () operator of the matlab::mex::Function class. The child class name must be MexFunction, so the compatible compiler knows which class has the implementation. Data between the program and MATLAB is passed through ArgumentList, an iterator of the data array. This means that the parameters must be defined and passed in a specific order. If this is followed, the exchange of parameters from MATLAB to C++ would be easy to implement. However, the program cannot count on the parameters being in the correct order, so the parameters are being checked for the specific data type.

Algorithm 3: MEX function parameters order and their types

```
1:  MexFunctionName(string Model,
2:                  uint64 Size,
3:                   string Presicison,
4:                   string Version,
5:      Array<float/double> Amplitude,
6:      Array<float/double> Phase,
7:            float/double Time,
8:            float/double Speed,
9:            float/double Frequency,
10:           float/double Spacing,
11:           float/double Alpha,
12:                 uint64 ExtandedSize,
13:                  int32 Gpuid,
14:           float/double BonA,
15:           float/double Rho0,
16:                 uint64 NumberOfIter,
17:                 uint64 NumberOfHarmonics);
```

The first five parameters are checked and loaded first, and then, similar to the stand-alone application, a helper function is used to define templates in one place. The remaining parameters are then checked and loaded to the Parameters object or, in the case of input amplitude and phase, to Matrix objects.

4.3 CPU C++ kernel implementation

The purpose of the CPU implementation in C++ has been discussed in chapter 3. I have started the rewrite process with the core function of the propagator 2.7.

The conversion process began with changing the matrix operation to loop notation. For acceleration of these loops, I used the OpenMP standard to parallelise this task. The original definition of the propagator function in MATLAB code is almost one-to-one with the mathematical notation. That means many calculations are repeated through the 40 lines of code. With this in mind, the separation of parts of the code that are repeated can be divided into two categories. The first category is for the parts that are the same for all acoustic field grid points. Then they are precomputed once and saved to the Parameters object.

The second category is repeating parts that are different for each grid point. These are computed in a loop for each point only once. Further optimisation wasn't done as the goal was to use a GPU to accelerate the propagator.

The function *solvePropagator()* mentioned in Figure 4.1 is used for the calculation of the propagator values. The function can be utilised in all inherited classes for all versions implemented in this thesis, as the core calculation of propagator values is the same for all of them.

The Intel oneMKL library handles FFT operations on the CPU side. In the initialisation stage of the library functions, I defined the inverse scaling parameter to be 1/numberOfPoints, so it is the same as the MATLAB ifft function.

No Expansion version

Class *AcousticFieldPropagtorNoExpansion* has a method calculateShifts that takes care of calculating the needed phase shift in each simulation dimension as described in equation 2.10. Additionally, if the time in the input file is set to zero, this method calculates the propagation time in steady state. The unique method for CPU No expansion Kernel is *createArrayOfCoordinates*. This function precomputes real coordinates for usage in the ramp function. These coordinates originate in the middle of the domain and represent the real distance of each acoustic grid point from the centre of the coordinate origin. All coordinates are saved to an array whose indices correspond to the correct input grid point. Two similar functions *solveRamp* and *solveInverseRamp* are used as forward and inverse phase ramps for correcting results like was described in the previous section 2.2.2.

Nonlinear version

The distinct feature of the NonLinear Kernel class is the function called *saveHarmonics()*. The sole purpose of this function is to save the simulation result to n files. One file contains the resulting acoustic pressure field, and the remaining files contain the resulting grid for each harmonic frequency of the numberOfHarmonics parameter. Although the implementations of the virtual methods differ, the CPU versions closely resemble the functionality and structure of the original MATLAB implementation. Therefore, they are not described in detail here.

Heterogeneous version

The heterogeneous version has several methods that are only used in the CPU version because most of the operations are processed with CUDA Kernels in the GPU version. Input amplitude and phase are again expanded with zero values, but the matrix for speed and absorption must be expanded differently.

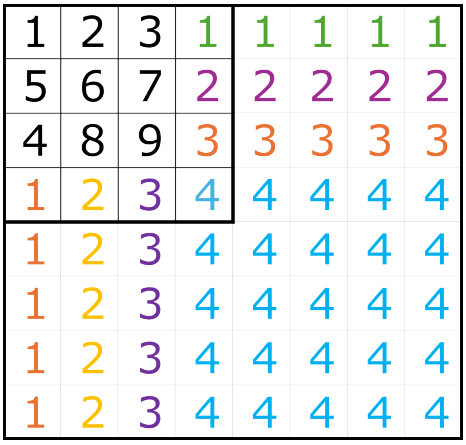

Figure 4.3: 2D domain expansion values

Figure 4.3 for the 2D domain shows how the values from the sides of the square grid are copied to the expanded grid. Each value on the edge of the user-defined grid is copied to the expanded grid in the direction of the expansion, different from the edge axes. The corners are then copied in both expanded dimensions.

In a 3D domain, the process is similar; however, it involves copying from the side of the grid in one direction, for edges in two directions, and from one corner extending into all three directions.

The expanded grid needs an absorbing boundary that prevents artefacts from the contrast source wrapping in the iterations. The necessary computation of the boundary is done in the method *getAbsorbingBoundery*. The boundary is created based on the Half Band frequency windows function with 0.35 shifted grid coordinates. The choice of this window function ensures a smooth transition of wave amplitude to zero, minimising reflections and preserving numerical stability.

Next, the complex wave numbers k^2 are computed with the *getComplexWaveNumbers* method. They are computed on the expanded grid, including the sound speed heterogeneities, absorption heterogeneities, and the absorbing boundary region. As was mentioned in section 2.2, the usage of Born series with damped Green's function requires the calculation to be done in iterations. The method *iteration* computes one iteration with the data from the function parameters.

4.4 CUDA kernels implementation

All CUDA kernels are compiled separately from CPUKernel and its child classes. It is separated in the CMake file into different modules. GPUKernels and their inherited classes are all compiled through the CUDA compiler. Similarly to the CPUKernel class, the function *createCufftPlans()* prepares the FFT library cuFFT. The initialisation of cuFFT is done by preparing a handle with a specific function. Based on the dimensionality of the simulation, the corresponding function is called, i.e. *cufftPlan2d* for 2D domain and *cufftPlan3d* for 3D. To simplify and shorten the code, the execution of forward and inverse FFT is wrapped inside the *runCufft()* method.

Because the input grid, regardless of dimensionality, is a 1D array of values and the kernels are executed with 2D/3D grids, there is a need for a device function that returns the index. Exactly for this purpose, the device's inline function *getIndex()* exists.

Algorithm 4: Function for getting index to 1D array from CUDA thread and block Id

```
return simulationDimension == SimulationDimension::k2D
        ? getYIndex() + getXIndex() * SizeY
        : getZIndex() + SizeZ * getYIndex() + getXIndex() * SizeZ * SizeY;
```

The *getXIndex()* in Algorithm 4 is an inline function that returns the X coordinate of a grid point and looks like this $threadIdx.x + blockIdx.x * blockDim.x$.

Device memory management and transfer functions between host memory are placed inside the GPUKernel class. This is done to separate the CUDA functions from the C++ CPU side. There are all static public methods. Therefore, they can be called from anywhere.

In the original MATLAB implementation, the nonlinearity parameter is defined as a matrix with only two different values for the expanded and original domains. This is a big waste of memory. The function *isInsideROI* checks if the GPU thread is inside the region of interest for the acoustic field grid, aka inside the non-expanded region. It is used to reduce memory usage for the nonlinear parameter calculation by not using wasteful amounts of memory on two distinct values.

Another set of functions inside the GPUKernel class is used to create a CUDA Graph. The definition of nodes for a CUDA graph is relatively long and can be confusing, so these functions simplify and shorten their notation. Provided methods are for creating kernel execution, memory transfer, memory initialisation (memset), and host code nodes, and for adding these nodes to the existing CUDA graph.

AcousticFieldPropagatorAbsorbing

This class implements the core of the propagator named *getPropagator* and its supporting functions. It is not placed in the GPUKernel class to enable support for different implementations of AFP if needed. However, the versions of AFP that are implemented in this thesis all use the same core propagation function, so all GPU kernel classes are derived from this class.

We need the wave number for the grid points to solve the propagator function. In the original MATLAB code, they are defined and saved in a matrix array before solving the propagator function. To save memory, one of the main changes was to move the definition of wave numbers to the inside of the propagator kernel. This means that before the main propagator function is executed, the *getWaveNumber()* device function is called inside the solvePropagator kernel before calling the getPropagator device function.

Algorithm 5: Wave number calculation for 3D simulation

```
xPart = (indexX + halfSizeX) % sizeX - (halfSizeX * scaleX)
yPart = (indexY + halfSizeY) % sizeY - (halfSizeY * scaleY)
zPart = (indexZ + halfSizeZ) % sizeZ - (halfSizeZ * scaleZ)
waveNumber = sqrt(xPart* xPart + yPart * yPart + zPart * zPart)
```

The scaleX variable from Algorithm 5 represents wavenumber spacing, and is defined as $\Delta kx = 2\pi/(Nx\Delta x)$ for a grid x axis with Nx grid points spaced by Δx .

As mentioned in the previous section, parts of the propagator function can be precomputed. The precomputed values will be used for every point in the propagator, and so they need to be present in easy and quick-access memory. For these reasons, the values are resident in the constant memory structure called GPUParameters. Constant memory has its own cache and memory access pattern. For most NVIDIA graphics card devices, the constant memory cache is eight kB per SM. The structure holds precomputed values and other parameters for the simulation time, sound speed, frequency, domain size, grid point spacing, etc., so the total size of the GPUParameters structure is 390 bytes for single aka float precision simulation. So, it easily fits into cache, and that way, the access speed for its values is better.

The precompute section that is constant is calculated in the function *prepareConstantMemory()* and then assigned to the structure. As mentioned in chapter 3, complex values are represented as a two-element array. Inside the kernel, these values are statically cast to the myComplex class. It is an inherited class from cuda::std::complex for easy conversion between these two representations of complex values.

Still, after the rewrite, the code for the propagator function is extensive and requires many registers per CUDA thread to be used. The core propagator function is separated into three parts in the original MATLAB code. I followed this and moved each section to separate kernels. Reduced the maximum number of registers used from 126 to 100, but the execution time did not decrease. Time does not change because some parts are calculated three times in each kernel. Additionally, the GPU used for testing supports the same number of wraps for both numbers of registers used.

The next step was to find a way to use shared memory. The previously mentioned three-part kernel execution would benefit from the use of shared memory. The parts computed multiple times can be calculated in the first kernel and placed in shared memory. This takes care of the repeated computation and helps further reduce the number of registers used. This solution improved the speed of the calculation by reducing register usage and eliminating repeated calculations.

In the original implementation, the propagator is computed differently for media without absorption, so in CUDA, this logic is placed in a separate device function. In testing, the value of the propagator without absorption does not fit in float values, as there are variables that are needed in the power of six. For this reason, if the input medium has the absorption alpha variable set to zero, the propagator core function is computed in double precision and then cast back to float if the program was run in single precision.

All previously mentioned propagator functions use the modified equation with a cosine start-up ramp to reduce Gibbs' oscillations [18]. It results in more accurate results, but adds time for the use of two other integrals. In MATLAB code, the startup ramp is switchable with the parameter UseRamp. This functionality is implemented to parrot the capabilities to separate device function *getPropagatorNoRamp*.

Another exemption for calculating values for the propagator is when the wave number k is nearly equal to the absorption coefficient *alpha*. In MATLAB, this is implemented as logical indexing of a matrix with calculated values for each of the three propagator parts. In my implementation, there is only one if statement that determines if the wave number is equal to the alpha absorption coefficient. Further optimisation of this part of the propagator is possible because all three previously mentioned components of the core propagation function remain constant across every point in the acoustic grid for this specific propagator exemption. This means that this value is computed before GPU kernel execution on the CPU and saved to CUDA constant memory as one complex value.

AcousticFieldPropagatorNoExpansion

Most parts of the class are similar to the previous Absorbing version. The execution is done in cycles for every shift based on the number of wrappings that must be cancelled. In practice, each wave shifted model is applied by first adding a phase ramp before the FFT. After that, the values are multiplied by the shifted propagator, calculated the inverse FFT, and then the phase ramp is removed. This process shifts the phase of the wrapped part of the field by $\pm\pi/2$, while the unwrapped part stays the same. When all shifted parts are added together, the wrapped shifted parts are cancelled, and the unwrapped part stays. The mentioned forward and inverse ramp functions must be implemented as CUDA kernels to accelerate the whole propagator calculation.

The previous absorbing AFP version does not need a buffer array and only requires memory for one matrix with input complex pressure values. The iterative nature of the wave wrapping cancellation technique requires three grid-size data arrays. One for input pressure grid state, second with current wrapping cancellation iteration pressure field and finally the total sum of field values. For this reason, the CUDA implementation can offer three versions, based on the memory requirement for device memory, but host memory is not reduced.

First, the so-called quick version has all three arrays located in device memory and does not have to make any data transfers between each wrapping cancellation iteration. It is the fastest solution, but requires memory for three complex variables on the GPU per acoustic field point.

The balanced version has two out of three arrays allocated in device memory. Thanks to this, at the iteration's start, the input data is copied to the array with the current iteration's acoustic field. Because each iteration starts with the input data, the calculations are needed only in the next iteration.

A low-memory version is included for cases where the simulated domain can almost fit in GPU memory with the balanced version. Device memory only has one array to which the acoustic field is calculated each iteration. At the end of each iteration, the data are copied to the host and added to the total field array. For this reason, the runtime is significantly larger than the previous versions and should be used only in extreme cases.

AcousticFieldPropagatorNonLinear

Nonlinear propagation generates harmonics at multiples of the source frequency. The number of harmonics that we are working with is specified with the program argument *-f* or *–numOfHarmonic*. Compared to the other versions, the output of this propagator is fundamental and harmonic acoustic pressure fields. The HDF5 file structure stays the same, and the fundamental and each harmonic field is saved to its own file.

We need Helmholtz equations to allow the simulation of nonlinear acoustics with AFP. This principle was mentioned and described in the section 2.2.3. To implement this efficiently, we would need all harmonic fields in device memory, but this approach has one problem. It can significantly increase the use of device memory. Like the previous AFP version, I implemented a low memory option.

A minimum of three grid-sized arrays must reside in device memory simultaneously. In the process of adding sum and difference harmonic terms to the pressure fields, one array is needed for the result field and two for the source fields.

Implementing this requires having all of the harmonic data saved in host memory. On the device side, we have two buffer arrays and a third array that is used to calculate the

current harmonic field. Reducing memory transfers is possible only in some cases when one of the two fields is the same between two different terms in figure 2.12. We keep the information about what harmonic field arrays are present in the two buffers on the GPU. This way, we can pass the pointer to the correct array that is needed in the next diff/sum kernel.

Minimising data transfer between the device and host memory necessitates a substantial increase in device memory allocation. So, the quick option does not use any data copy between executions and has all data resident in the device's memory.

The MATLAB implementation of this AFP version deals with wave warping using grid expansion. But thanks to the nonlinear acoustic, the expansion factor must be set higher.

The calculation of the expanded grid was designed to prevent wrapping only when the source is within the user-defined grid size. For the calculation of the harmonics, we must assume that the field exists everywhere. This means that contributions to the harmonic field in the region of interest come from within a distance no greater than the length of the main diagonal. So the expansion must cover at least that distance, i.e. the expansion factor must be bigger than two.

The need for a bigger expanded domain means an even larger memory requirement, and another increase is for the already mentioned harmonic fields that must be included in the simulation. All of this means that, to correctly run the AFP with steady state in mind, on a cubic input domain size of 128 grid points per dimension, we would need to expand it to 1500 points. Plus, this expansion must be done for every harmonic field array, so altogether, with a single precision with five harmonic fields, we would need a GPU with 216 GB of VRAM or 81 GB of low memory version.

With this in mind, I incorporated the wave wrapping cancellation technique used in the previous AFP version. Although extending the propagator in this way was not the primary objective of the thesis, the high memory demands of the grid expansion method motivated me to pursue further optimisation of this version. The only change needed was to change the calculation of the number of shifts, aka wave wrapping steps to perform, to account for the aforementioned harmonic calculation assumption.

This calculation of each fundamental and harmonic field is repeated for a selected number of iterations with the parameter *-i*, to improve the accuracy of the difference and sum terms. With the need for iteration, the large number of kernels executed in the case of the wrapping cancellation version and memory transfers for the low memory version, there is space to use CUDA Graph.

The graph is created using the previously mentioned supporting function in the GPUKernel class. With these, every node is added manually, with the exception of the cuFFT libraries' execution of FFT. This is done through the stream capture of the library call function, and this creates a graph that is added to the main graph as a node. All nodes needed to run one iteration of the harmonics fields are added to the graph. After that, the graph instance is created and then executed sequentially for the number of selected iterations.

AcousticFieldPropagatorCBS

The final AFP is designed to accommodate heterogeneous medium parameters, specifically variations in sound speed and absorption.

The override preprocess method performs operations similar to those of the CPU counterpart. It expands input matrices for medium heterogeneous parameters and prepares constant memory and its constants.

Due to the non-negligible computation time required for calculating complex wavenumbers in larger domains, this process has been offloaded to the GPU. The kernel for this function, similar to the propagator, is launched as a 3D / 2D grid to use the thread indexes to calculate the wave numbers of the grid points. Once this step is completed, the maximum value of the complex wavenumbers must be determined. Since the data resides in device memory, the max-retrieval function has been implemented on the GPU.

The function employs parallel reduction using a greater-than comparison operation. The implementation requires an atomic operation at the end of the kernel to reduce to global memory. Unfortunately, CUDA doesn't support atomicMax for floating point numbers [5]. But this can be implemented with a reinterpret function for float to int and back, and call atomicMax for the int value.

Algorithm 6: Atomic Max for float values

```
1: result = !signbit(value)
2: ? ___int_as_float(atomicMax((int*)addr, ___float_as_int(value)))
3: : ___uint_as_float(atomicMin((unsigned int*)addr, ___float_as_uint(value)));
```

The above implementation takes care of the problem with the negative zero value by using the signbit function [15]. The error can occur when the -0.0 value is compared with -1.0. The function would return -1.0 as the bigger value. The challenge lies in implementing equivalent functionality for double-precision values. Again, CUDA does not have a function to reinterpret double to unsigned long long [6]. Fortunately, this function is only called with non-negative values, so handling negative inputs is unnecessary. So only the atomicMax function portion of the implementation is used, which does not need the reinterpretation to unsigned long long.

This propagator is built based on the need for an absorbing boundary in the expanded region. The wave wrapping cancellation method cannot be implemented without changes to the iteration process. As this was not the goal of this thesis, we used only the grid expansion solution to the wave wrapping problem.

Like the previous propagator versions, I implemented multiple versions based on the required device memory. Each iteration requires multiple grid-sized arrays containing the following data: input pressure field, buffer with contrast pressure source, complex wavenumbers, intermediate results of the previous iteration of the convergent Born series, and precomputed propagator values.

Similarly to the other quick versions of AFP, this approach reduces the memory transfer between the device and the host to a minimum. All five arrays are resident in device memory, and only at the start and end, there are data transfers.

On the other hand, the minimum memory needed to calculate an iteration is two grid-sized complex arrays. The calculation of one iteration is done in five stages. Between each part, one of the arrays is transferred from/to the device memory or both. This is slow and connot be efficiently parallelised. We have to transfer grid size arrays seven times per iteration, so obviously, the execution time is approximately six times slower.

Finally, the balanced version uses one more array than the low-memory version. With this, we only need to separate the iteration calculation into two parts. Total memory transfers per iteration are also decreased to three.

When testing the CUDA implementation, in some cases with a high frequency or time, the calculation of the core propagator function returns infinite values. In the process of calculating the propagator's values, some large exponents are present, and single-precision floating-point precision cannot handle these values. This problem is also present in the propagator version, where the absorbing coefficients are zero. In these cases, the propagator is calculated with double precision, regardless of the selected simulation precision. However, the resulting values are cast back to float when running at a single precision.

4.5 CUDA cmtah library problem

The code and program were developed and written on the Windows operating system. The continuous testing through development was then done on the Windows machine. However, the use of CMake makes it easy to compile for different operating systems. After testing my implementation on a Linux-based machine with a similarly powerful GPU, I saw drastically different times when running in single precision.

To investigate what was happening, I have employed the use of Nsight Compute. Nsight Compute is a performance analysis tool from NVIDIA designed specifically for CUDA kernel profiling on NVIDIA GPUs. It provides detailed insights into how individual CUDA kernels perform on the GPU, helping optimise and debug the code at a low level.

When profiling on a Windows system built using the Intel C++ compiler alongside nvcc (with MSVC handling host code compilation), the majority of arithmetic logic unit (ALU) instructions were dedicated to single-precision floating-point operations.

In contrast, running the same code on a Linux system, where the host code was compiled with GCC, revealed a different behaviour. A significant portion of the computational time was spent on 64-bit (double-precision) floating-point ALU instructions.

To further investigate, I have compiled with the nvcc argument *lineinfo* to include line numbers. The compiler embeds source line number metadata into the generated binary. This information maps the compiled GPU instructions back to the lines of the original source code. After that, I found multiple functions that have different implementations based on the compiler used. For example, the function *___constexpr_scalbn* is used in the divide operation for complex numbers.

```
875  #if defined(_CCCL_COMPILER_MSVC) || defined(_CCCL_COMPILER_NVRTC) || defined(_CCCL_CUDA_COMPILER_CLANG)
876  template <class _Tp>
877  inline _LIBCUDACXX_INLINE_VISIBILITY _Tp __constexpr_scalbn(_Tp __x, int __i)
878  {
879    return static_cast<_Tp>(::scalbn(static_cast<double>(__x), __i));
880  }
881
882  template <>
883  inline _LIBCUDACXX_INLINE_VISIBILITY float __constexpr_scalbn<float>(float __x, int __i)
884  {
885    return ::scalbnf(__x, __i);
886  }

901  #else
902  template <class _Tp>
903  inline _LIBCUDACXX_HIDE_FROM_ABI _LIBCUDACXX_INLINE_VISIBILITY _CCCL_CONSTEXPR_CXX14_COMPLEX _Tp
904  __constexpr_scalbn(_Tp __x, int __exp)
905  {
906  #  if defined(_LIBCUDACXX_IS_CONSTANT_EVALUATED) && !defined(_LIBCUDACXX_HAS_NO_CONSTEXPR_COMPLEX_OPERATIONS)
907    if (_LIBCUDACXX_IS_CONSTANT_EVALUATED())
908  >   {…
956    }
957  #  endif // defined(_LIBCUDACXX_IS_CONSTANT_EVALUATED)
958    return __builtin_scalbn(__x, __exp);
959  }
960  #endif // !_CCCL_COMPILER_MSVC
```

Figure 4.4: scalbn function from libcudacxx library inside CUDA Toolkit

For MSVC, NVRTC, and CLANG compilers, this template function is explicitly specialised for float, double, and long double. Each one uses a different scalbn function from the C math library, tailored to handle specific floating-point precisions. All other compilers use a different implementation that can be seen at the bottom of Figure 4.4. All values are processed using the *___builtin_scalbn* function, which accepts and returns values in double precision. This means that no matter what the template argument is, all values are compiled to a double ALU instruction. This is the same for another function *___constexpr_logb*.

To solve this, I have added my template function, which is explicitly specialised for float, to the GPUKernel class header file for both mentioned functions. This implementation is inside a compiler macro to include these specialised functions only if the code is compiled with the GCC compiler. When profiled again, the usage of double-precision instructions disappears. The runtime of the simulation ran as expected.

Chapter 5

Testing

This chapter discusses the testing procedures for the implementation and the evaluation of the final results of this thesis component. The measurement where it is not mentioned elsewhere, the AFP program was run on this machine:

- CPU: Intel Core i7-13700k
- RAM: 64GB DDR5 5600Mhz (4x16GB in two channels,timing 36-38-38-80 cycles), 89.6 GB/s memory throughput
- GPU: NVIDIA GeForce RTX 3080 Ti 12 GiB VRAM, 912.4 GB/s memory throughput
- SSD: NVMe (Conencted with PCIe 4x 16.0 GT/s) read/write 3,500/2,800 MB/s
- OS: Windows 11 Pro, version 23H2

During the conversion of all versions of the acoustic field propagator, rigorous tests were performed to ensure code accuracy and functional integrity.

5.1 Testing setup

To make the testing process easier and allow quick error checking for new program versions, I have created three scripts. The scripts were written in Python using some libraries. The first script, named *GenerateTestMatrix*, takes care of generating arbitrary test input HDF5 files. It can be called with parameters from the terminal or imported into another script and used with a function. The first library utilised was h5py, which enables the creation and loading of HDF5 file formats. It leverages the widely used NumPy library to represent data within the Python environment [3]. When executed from the terminal, it generates files containing cubic input grids ranging in size from 32 to 1024 grid points per dimension, in increments of 32. The script parameters are :

- -d / –dimensions: Select dimension of the simulation acoustic grid, valid for the values 2 and 3
- -v / –version: Select version AFP the data will be generated, valid for the values 0,1 and 2

If the function is used, we need these parameters:

- **version**: Select version AFP, the data will be generated, valid for the values 0,1 and 2
- **dim**: Select dimension of the simulation acoustic grid, valid for the values 2 and 3
- **speed**: sound speed
- **spacing**: spacing of grid points, uniform for all dimensions
- **time**: time at which we want the AFP to compute acoustic pressure. The script calculates the steady state time for input parameters if set to zero.
- **frequency**: wave source frequency
- **alpha**: absorption coefficient
- **gridExpFactor**: expansion factor for expanding wave warping solution
- **xSize, ySize, zSize**: each parameter defines grid axis size
- **rho0**: medium density for nonlinear version
- **bonA**: acoustic nonlinearity parameter (Beyer's parameter)

A more important script is *RunTest*, which handles executing tests of my implementation and comparing both accuracy and performance against the original MATLAB version. It utilises the **matlab.engine** library to execute the original MATLAB implementation and retrieve the resulting data. It is easy to run any function with two function calls. First, we must start the MATLAB engine with *start_matlab())*, then we change the current folder to the path handed over from the script argument. The remaining step is to invoke the corresponding AFP function.

My GPU AFP program is called with the *subprocess.call* a Python function. The result is then loaded again with the use of the h5py library. To verify correctness, the mean relative error is computed using NumPy functions. To visualise the differences, the matplotlib.pyplot library was used to plot both the results and the raw discrepancies between my implementation and the MATLAB version.

Script arguments are similar to those of the GPU Afp program.:

- **-d / –dimensions** : dimensionality of input file and simulation
- **-m /–Model**: selection of AFP version - absorbing, noexpansion, nonlinear, cbs
- **-v / –modelVersion**: select AFP model version: quick, balanced, lowmemory, default is balanced
- **-n / -numberOfIteration**: sets the number of iterations for the nonlinear and CBS model, default is 60
- **-f / –matlabFolder** : path to the folder with MATLAB function file
- **-p / –precision**: precision of variables and math functions for the whole simulation, two options are available - single/double
- **–CPU**: switch between GPU and CPU implementation, if this argument is present, the CPU version will be run

- **-e / –expansion** : Run nonlinear model with wave warping cancellation using expansion of the acoustic field grid, if not present, runs with shift cancellation
- **-i / –GPUID**: Select gpu id on witch the GpuAfp will run
- **-g / –GPUAfpFolder**: Folder with the executable file of my AFP implementation
- **–outputPressure**: switch to output matrix of the complex pressure field, where the real part corresponds to the pressure field for a cosine excitation and the imaginary part to sine excitation

Finally, the script *RunTests* measures the GPU AFP performance of one of the specific versions. A distinctive feature of this script is the assessment of memory requirements using equations that will be described later. The script generates an input file with the *GenerateTestMatrix* function, running all model versions, aka quick balanced and low memory. It parses the total runtime from the terminal output of the GpuAfp program and appends it to a CSV file. So the output of this script is a CSV file with measured times for one AFP version. Parameters of this script include:

- **-g / –GPUAfpFolder**: Folder with the executable file of my AFP implementation
- **-m /–Model**: select AFP model version: quick, balanced, low memory, default is balanced
- **-p / –precision**: precision of variables and math functions for the whole simulation, two options are available - single/double
- **-i / –GPUID**: Select gpu id on witch the GpuAfp will run
- **-n / –numberOfIteration** : sets number of iteration for nonlinear and CBS model

5.2 Error evaluation

Error evaluation is performed in the aforementioned script *RunTest*, and combines multiple approaches.

To evaluate the accuracy of the GPU-based implementation against the original MATLAB-based solution, several error metrics are computed for both the amplitude and phase components of the pressure field. These include the mean relative error, which quantifies the average pointwise discrepancy relative to the reference, and the root mean squared error (RMSE), which provides a measure of overall deviation in absolute terms. The maximum absolute error is also reported to capture the worst-case local discrepancy.

To further assess the fidelity of the solution in the frequency domain, n-dimensional Fast Fourier Transforms (FFT) are applied to both amplitude and phase fields. The resulting magnitude spectra are normalised and compared to reveal any discrepancies in the frequency content between the implementations. The difference in these spectra is visualised by plotting central 2D slices of the amplitude and phase frequency components, offering insight into how the spectral characteristics of the computed pressure fields differ. These combined spatial and spectral analyzes provide a comprehensive validation of the accuracy and numerical consistency of the implementation.

An example of the resulting report for running the no-expansion version with an input grid size of 256^3 in single precision can look like this:

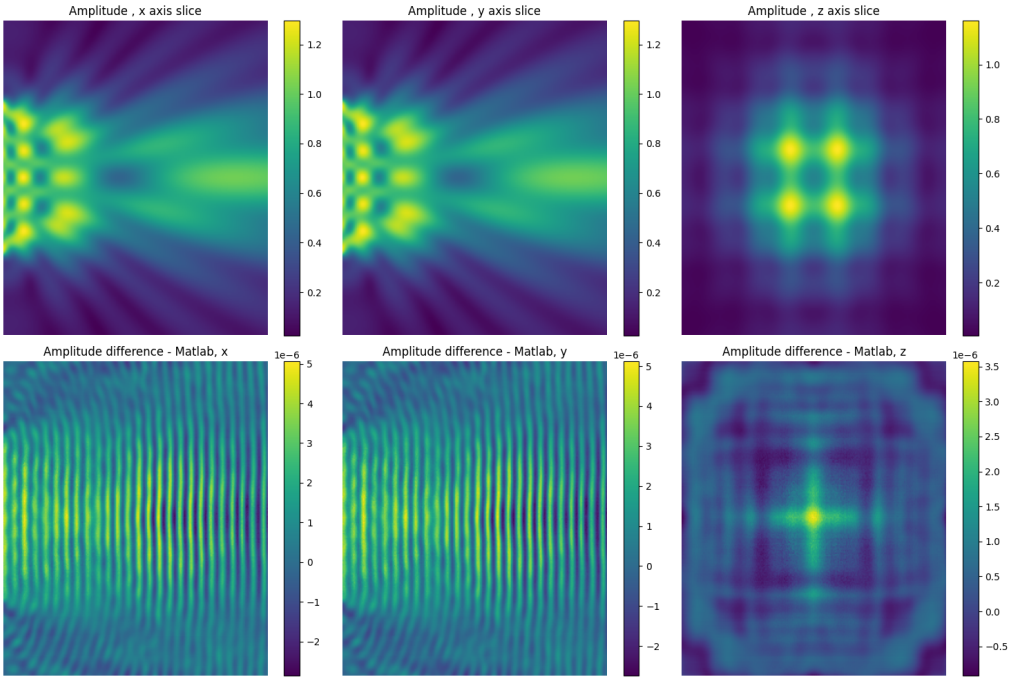

Figure 5.1: Comparison of CUDA and MATLAB absorbing AFP Top: result amplitude slice in each axis Bottom: raw difference from MATLAB result

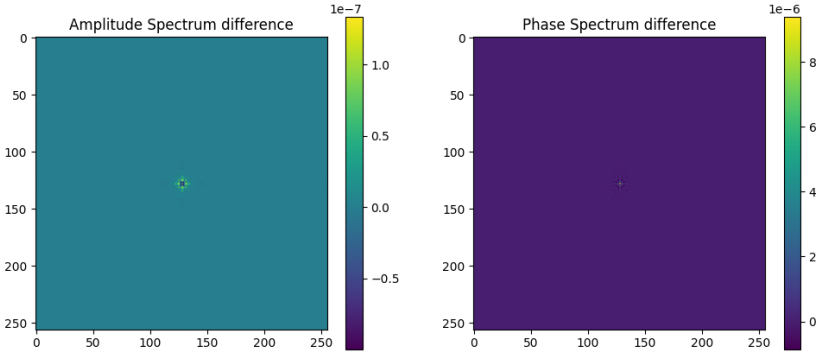

Figure 5.2: Frequency domain difference for amplitude on the left and phase on the right

```
Amplitude mean relative error: 4.49e-06
Phase mean relative error: 3.55e-05
Amplitude Root Mean Squared Error : 9.18e-07
Phase Root Mean Squared Error : 1.15e-05
Amplitude max error: 5.25e-06
Phase max error: 3.00e-03
```

Figure 5.3: Calculated errors

This example test is performed in comparison to the MATLAB implementation of the same AFP, so it only serves as an evaluation of implementation correctness. The slight inaccuracy is caused by the original CPU implementation running in double precision.

The following sets of figures present comparisons with the k-Wave simulation. In this case, the grid size is reduced to 128^3 to shorten the k-Wave simulation time.

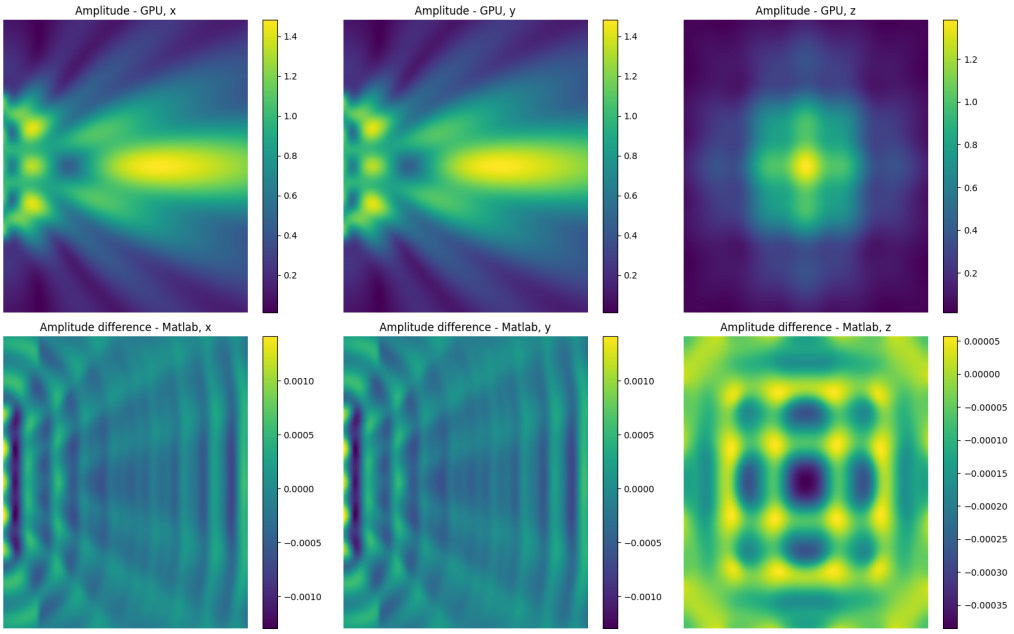

Figure 5.4: Comparison of CUDA AFP and k-Wave Top: result amplitude slice for each axis Bottom: raw difference from k-Wave result

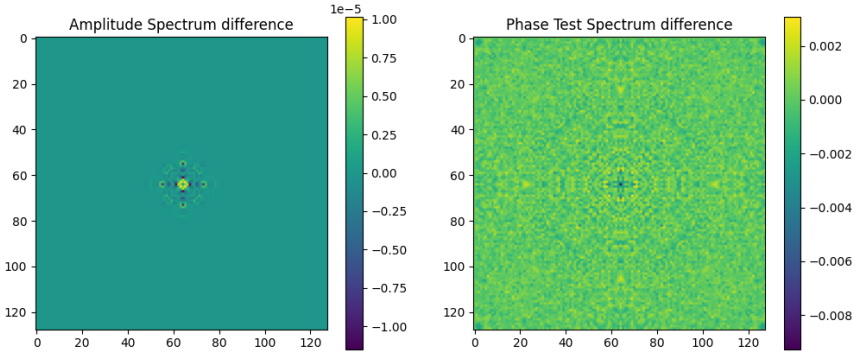

Figure 5.5: Frequency domain difference for amplitude on the left and phase on the right

```
Amplitude mean relative error: 8.17e-04
Phase mean relative error: 8.93e-02
Amplitude Root Mean Squared Error : 2.94e-04
Phase Root Mean Squared Error : 3.27e-01
Amplitude max error: 2.15e-03
Phase max error: 6.27e+00
```

Figure 5.6: Calculated errors

When compared to other types of simulations, such as the time-domain approach used by k-Wave, we can observe that the mean relative error hovers around $8.17e^{-4}$ for amplitude and a little bit bigger for phase error.

To show the difference between single and double precision options of the noexpansion AFP, the same input file and parameters were run and compared:

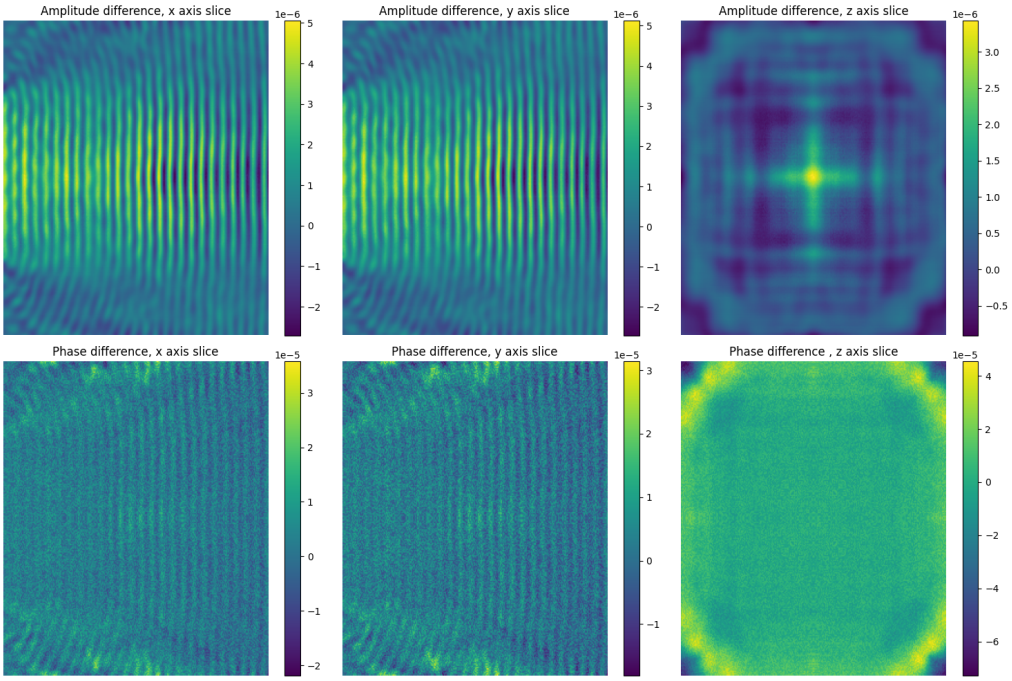

Figure 5.7: Comparison of CUDA and MATLAB nonlinear AFP Top: raw difference in amplitude between precisions Bottom: raw difference in phase between precisions

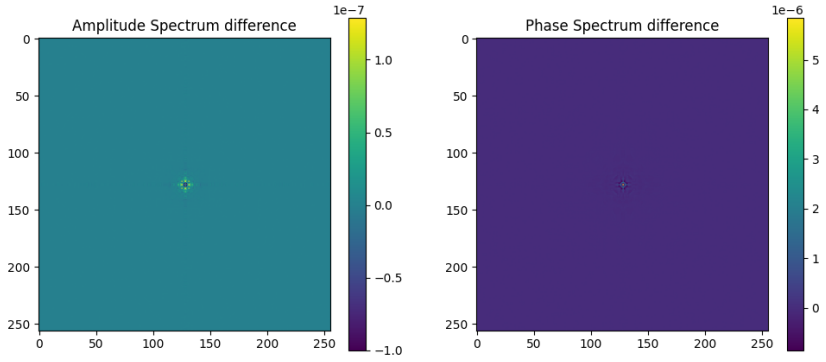

Figure 5.8: Frequency domain difference for amplitude on the left and phase on the right

```
Amplitude mean relative error: 4.46e-06
Phase mean relative error: 2.87e-05
Amplitude Root Mean Squared Error : 9.02e-07
Phase Root Mean Squared Error : 1.07e-05
Amplitude max error: 5.12e-06
Phase max error: 2.86e-03
```

Figure 5.9: Calculated errors

We observe that, in this example, both options yield very similar computational errors. However, as the source frequency increases, the phase error increases noticeably. To address this, the implementation issues a warning in the terminal when the input frequency exceeds a certain threshold.

The original MATLAB code for AFP in nonlinear acoustics was limited to 2D domains. For this comparison, I made minor modifications to enable 3D simulations. Due to the slow performance of the original implementation, I opted for a smaller grid size of 64^3 points. Again, the CUDA-accelerated version was run in single precision with ten iterations for five harmonic fields.

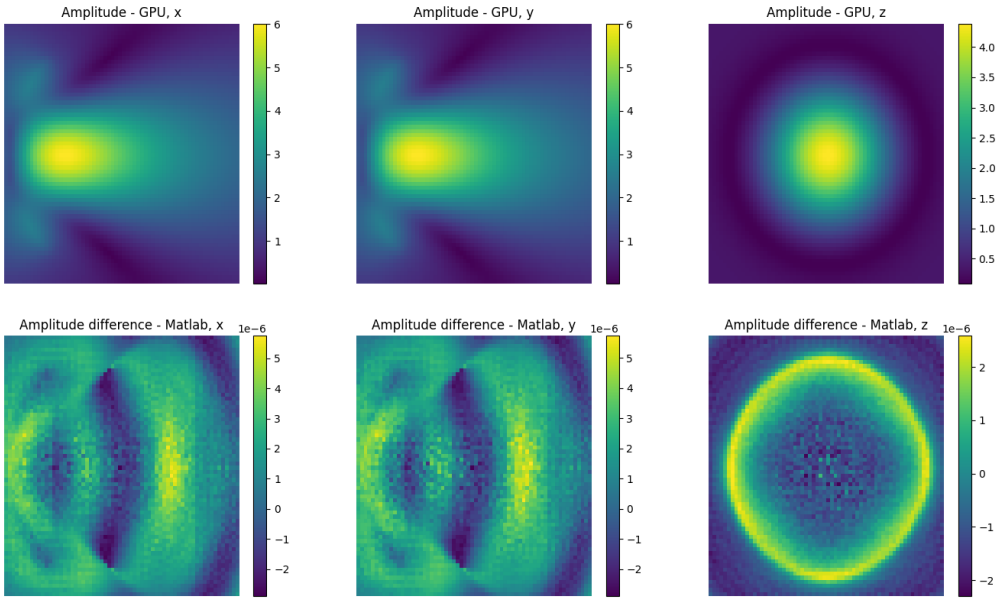

Figure 5.10: Comparison of CUDA and MATLAB nonlinear AFP Top: result amplitude slice for each axis
Bottom: raw difference in amplitude to MATLAB version

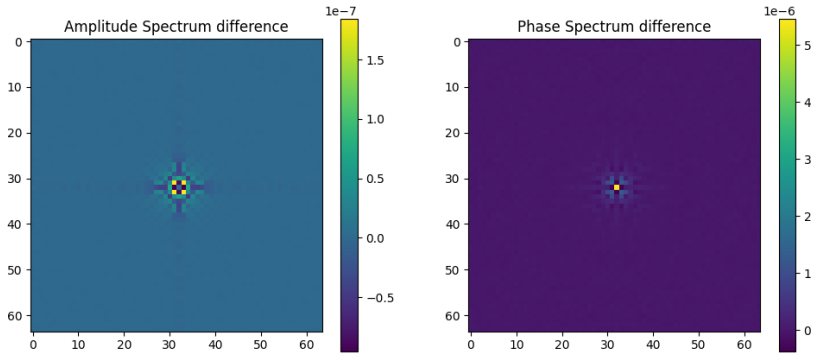

Figure 5.11: Frequency domain difference for amplitude on the left and phase on the right

```
Amplitude mean relative error: 2.46e-06
Phase mean relative error: -8.92e-06
Amplitude Root Mean Squared Error : 1.75e-06
Phase Root Mean Squared Error : 5.95e-06
Amplitude max error: 6.27e-06
Phase max error: 3.71e-04
```

Figure 5.12: Calculated errors

AFP with nonlinear acoustics is similarly accurate. The only difference is with the optimisation option with wave wrapping cancellation. The higher harmonics have a mean

relative error of $3.76e^{-3}$, while the fundamental is around $5.8997e^{-6}$. It is caused by the grid size in the higher harmonics field arrays, where the transmitter frequency is higher.

Finally, the comparison for the heterogeneous version is performed using the same smaller 128^3 grid.

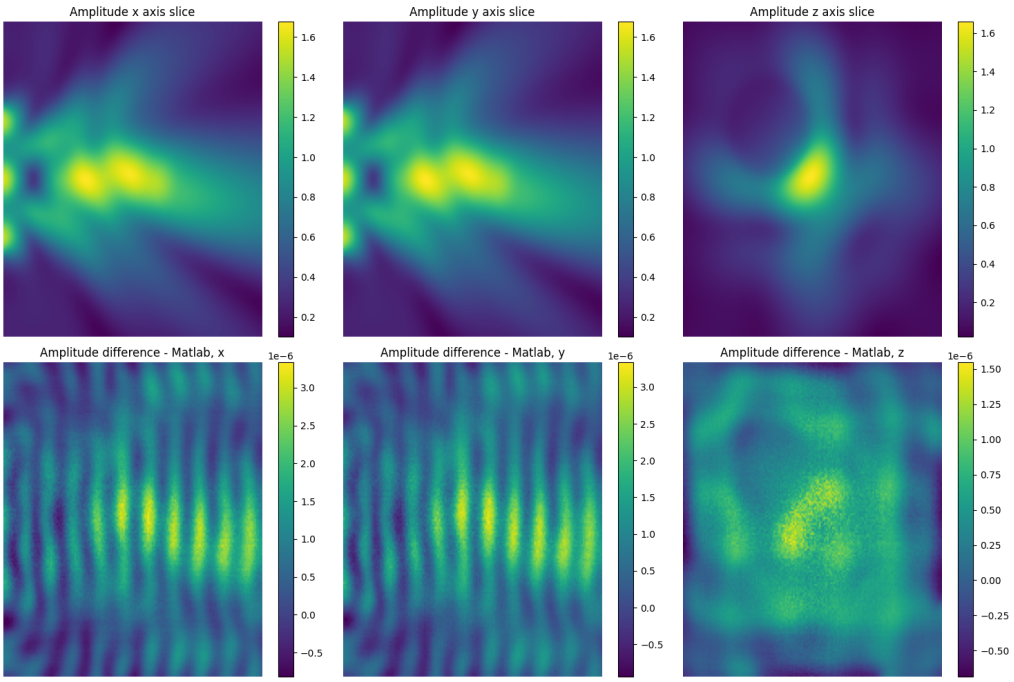

Figure 5.13: Comparison of CUDA and MATLAB heterogeneous AFP Top: result amplitude slice in each axis Bottom: raw difference from MATLAB result

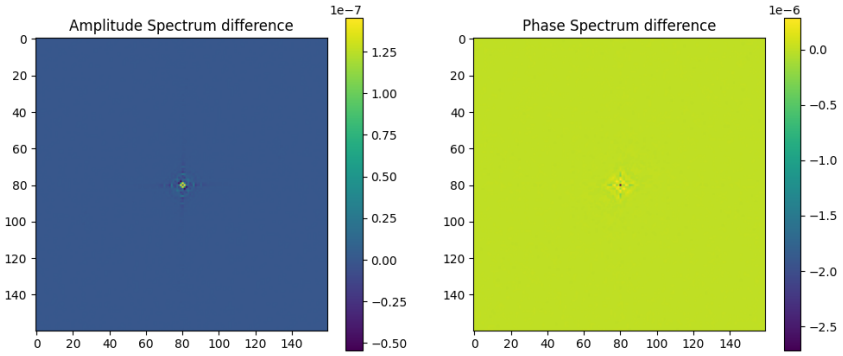

Figure 5.14: Frequency domain difference for amplitude on the left and phase on the right

```
Amplitude mean relative error: 2.19e-06
Phase mean relative error: 1.65e+02
Amplitude Root Mean Squared Error : 6.32e-07
Phase Root Mean Squared Error : 4.83e-06
Amplitude max error: 3.46e-06
Phase max error: 1.44e-03
```

Figure 5.15: Calculated errors

Once again, the results indicate that the implementation is correct, showing strong agreement with the data from the original MATLAB implementation.

5.3 Time comparison

Execution time comparison is performed partly through the *RunTests.py* script and partly by manually running the MATLAB version. Each timing measurement was performed at least twice and the results were averaged if the variation between runs is within 5%. Additional runs were performed if the difference exceeded this threshold.

The first measured AFP version is the absorbing version using an expanding grid for the wave wrapping problem. It is compared against both the original MATLAB CPU-based solution and the CUDA-accelerated k-Wave simulation.

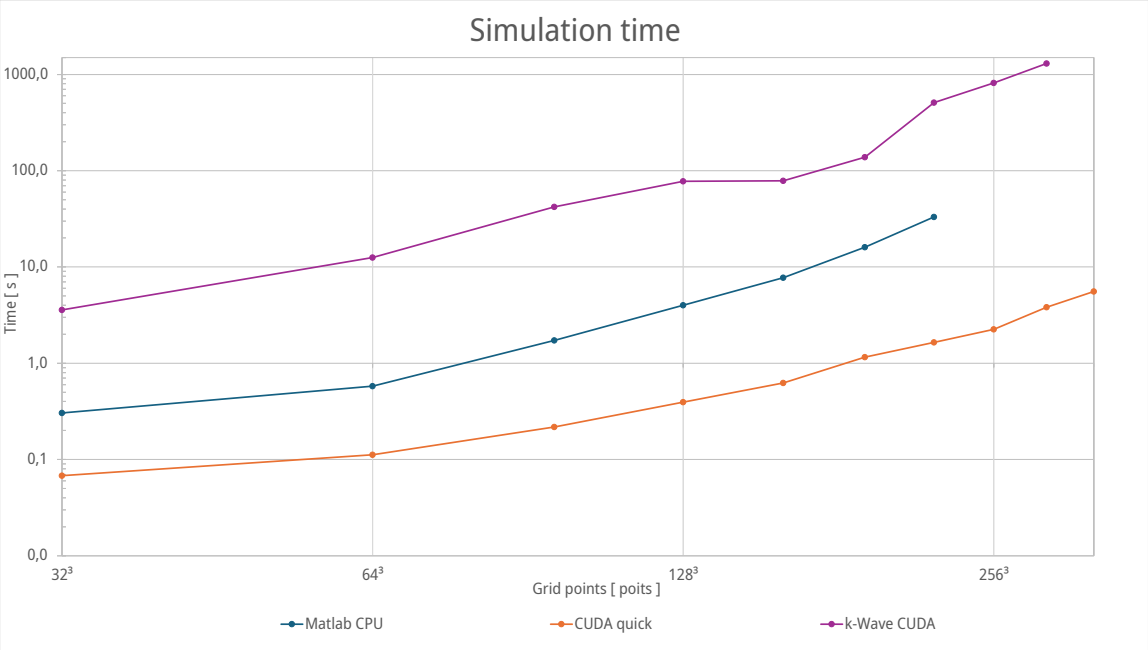

Figure 5.16: Time comparison of absorbing expanding propagator

From the figure 5.16, it is evident that marginal acceleration was achieved when compared to the MATLAB CPU simulation. The simulation time was reduced by between 5x and 20x, depending on the size of the input grid.

Second, the non-expansion AFP is measured and compared against both the original MATLAB implementation and the version utilising gpuArray for GPU execution.

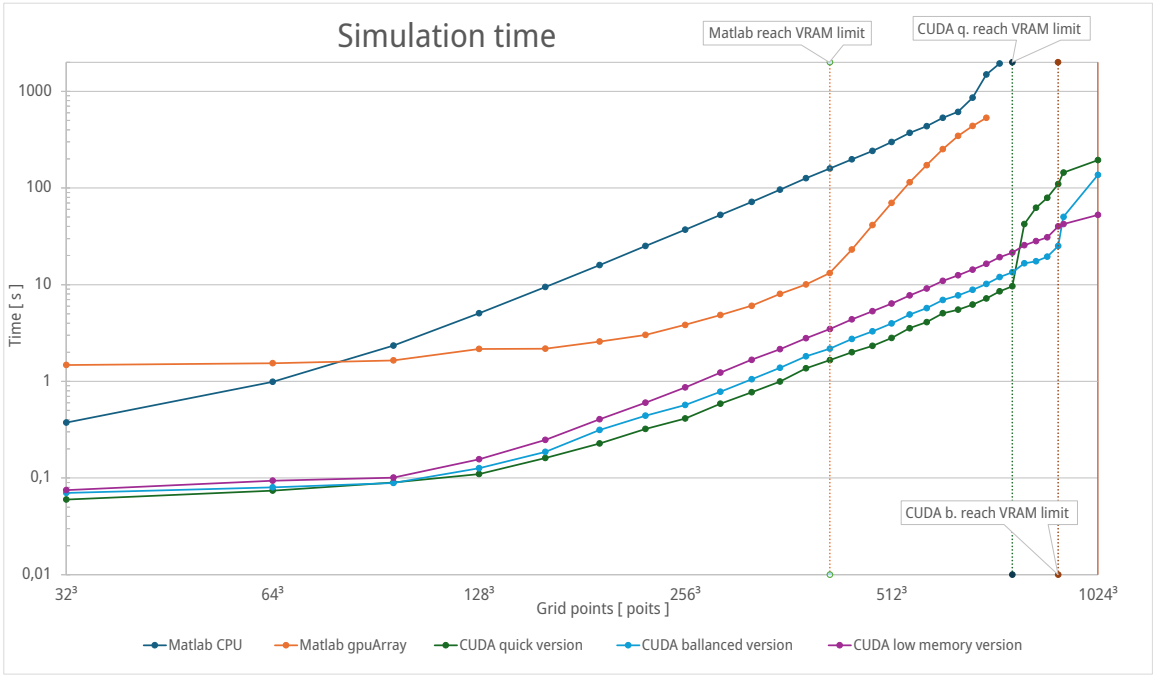

Figure 5.17: Time comparison of no expansion propagator

In figure 5.17, we can see sudden elevated values for time measurement because the test was done on Windows, where the programs can use shared GPU memory. GPU can use this system RAM as backup if it runs out of dedicated memory. It is slower and only used when needed. That is why these spikes are visible every time the grid is too big to fit into GPU VRAM.

Compared to the CPU MATLAB version, the runtime of all CUDA AFP versions is significantly quicker. The quick version usually achieves a speedup ranging from 70× to 90×. For the other two versions, the balanced implementation achieves a speedup of 50× to 70×, while the low-memory version reaches approximately 40× to 45×.

When comparing MATLAB's gpuArray acceleration with CUDA AFP, the speedup is notably less significant, but as we will see, it is more impressive regarding memory efficiency. Nevertheless, until the VRAM limit is reached, the minimum observed speedup remains in the range of approximately 7.5× to 8×.

Next, we have the acoustic field propagator with nonlinear acoustics. Here, the comparison is with the original version and both solutions to the wave wrapping problem. The values shown in Figure 5.18 were obtained from measurements using my GPU AFP implementation and the original MATLAB version, configured with five harmonics and ten iterations.

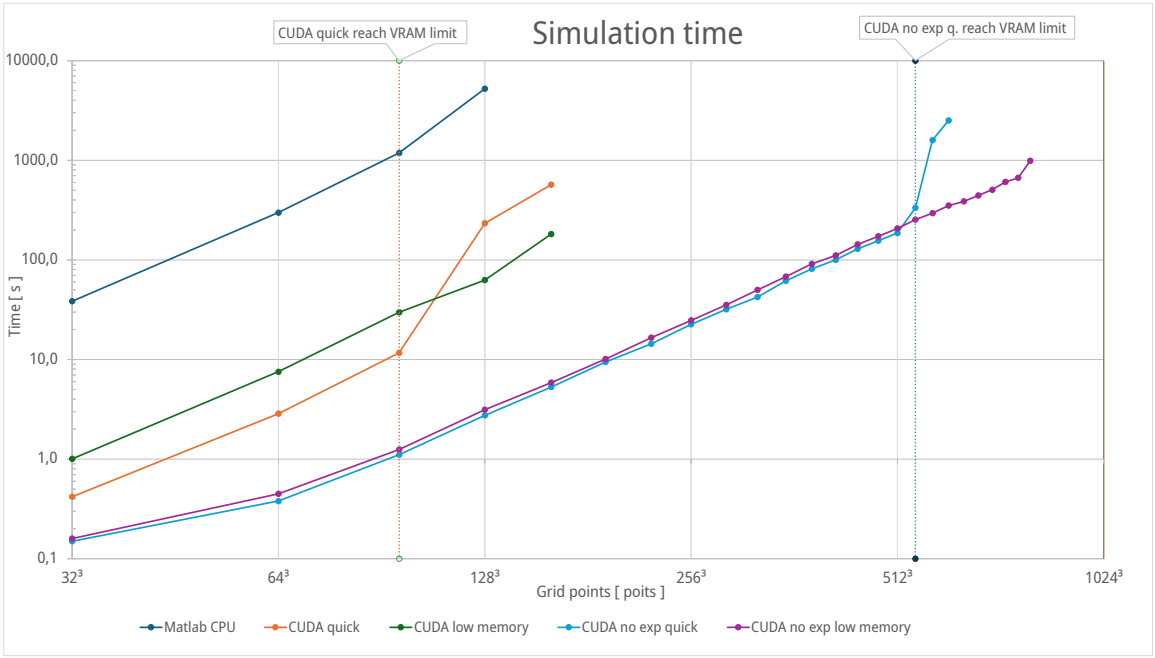

Figure 5.18: Time comparison of nonlinear acoustic propagator

The previous AFP model was quicker because it used no expansion wave warping, whereas this original implementation uses expansion with a scale factor of two. For this reason, on my machine, the biggest simulation that could be run without crashing was a cube with 128^3 grid points. The expanded domain was 648 x 648 x 648 points wide.

When comparing to the CUDA quick version, we can see measured times that are 100x smaller. Of course, this is only until we have enough VRAM memory. The low-memory AFP version is also significantly faster, achieving a speedup of approximately 40× for a cubic grid with 96 points per edge. But the versions with no expansion are on a whole other level.

Even when compared to the CUDA-based expansion counterpart, the non-expansion versions are 10 to 30 times faster. At the same time, we can simulate more than twice the size of cube grids, which means simulating 8 times more input grid points.

The final time comparison for AFP with heterogeneous media includes the original CPU MATLAB version and the k-wave propagator simulation time. k-Wave runtimes are recorded using the CUDA-accelerated version, executed on the same machine with the p-Final parameter configuration [16]. For all versions of the AFP, the simulation parameters were set to 120 iterations and run until reaching a steady state.

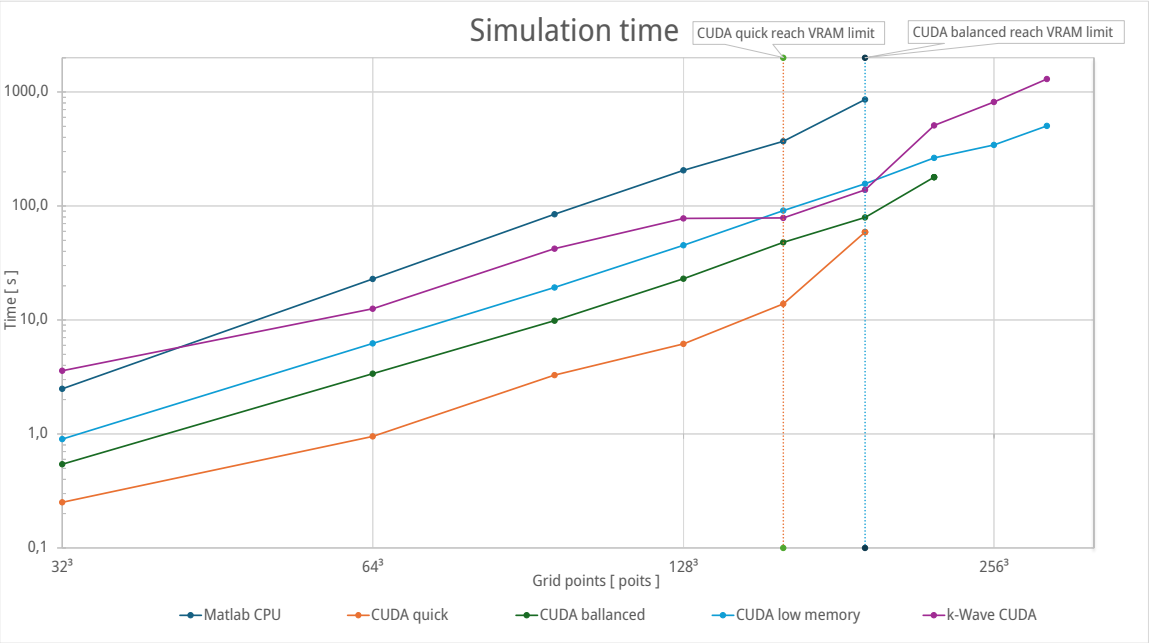

Figure 5.19: Time comparison of heterogeneous propagator

Early profiling of the original MATLAB code indicated that the majority of computation time is spent performing Fast Fourier Transforms within the iterations. Consequently, accelerating only the core propagator function is unlikely to result in substantial overall time savings. This is evident in Figure 5.19, and when compared to the other two figures showing simulation times, it is clear that the run times are much closer to the original execution times.

The quick CUDA implementation of AFP is approximately 20× to 25× faster than the MATLAB CPU-based version. The speedup ranges between 4x and 5x when running the low memory option. Even the CUDA-accelerated k-Wave simulation is always quicker than the original MATLAB version.

Another interesting comparison is the time difference between the two precision options that all versions of AFP can run at. The following graph shows the difference in execution times between single and double precision on a GPU with significantly lower performance in double precision.

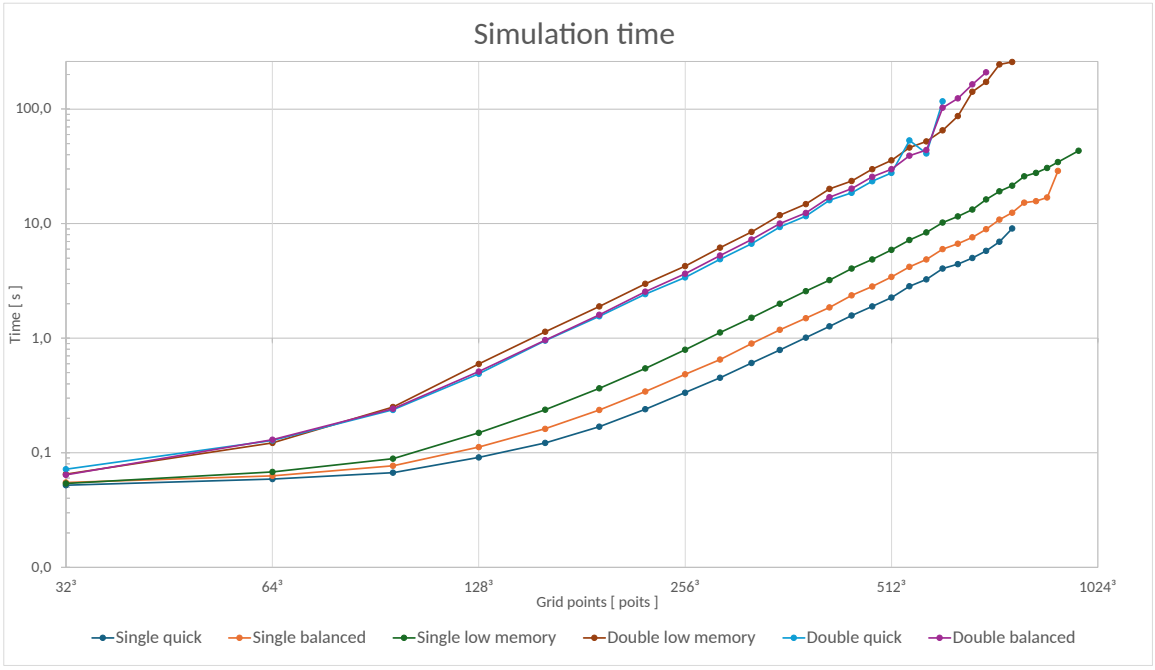

Figure 5.20: Time comparison of no expansion propagator for single and double precision

The measured difference in time ranges from 2 times slower in a small domain to more than 12 times slower with a grid above the size of 460 points. Another notable observation from the graph of measured times is that, for the different memory usage versions, the single precision option shows a greater variation in performance. This may be attributed to the high throughput of double precision computations, with the propagator kernel operating at 90 % compute utilisation and only 2 % memory throughput.

The final observation from Figure 5.20 is the jumps in double precision large domains. This behaviour is due to the internal workings of the cuFFT library. For large or non-standard domain sizes, the library may allocate extra memory for internal work buffers. In this example, that additional allocation causes the total memory usage to exceed the GPU's VRAM, prompting Windows to fall back on system memory.

5.4 Memory comparison

The time comparison does not complete the picture of the improvements achieved by the GPU acceleration of the acoustic field propagator. This section will show and discuss improvements in memory management and memory requirements for each AFP version and input grid sizes. Equations for calculating expected memory requirements based on the input acoustic field grid size will be included. The measurements were performed using the Nsight Systems analysis tool. This tool allows system-wide profiling, captures interactions between CPUs and GPUs, and provides an overview of application behaviour. It can visualise the timeline of the profiling application, displaying events and execution across threads, processes, and GPU kernels on a unified timeline, making bottlenecks easier to identify. It provides insight into how CPU-side code launches GPU kernels and how long each component takes. And importantly, it provides memory activity tracking that monitors data transfers between host and device memory.

With that clarified, we can now examine the results of the measurements. In Figure 5.21, there are measurements of non-expanding CUDA AFP with the MATLAB GPU-accelerated version. Additionally, dotted lines show the input file's calculated and predicted required memory.

The equations below define the device memory requirements for quick, balanced, and low memory versions in bytes.

$$M_d = N * P * 3 \qquad M_d = N * P * 2 \qquad M_d = N * P \tag{5.1}$$

Here, N denotes the number of grid points, and P represents the size of each point in bytes, typically 8 bytes for single precision and 16 bytes for double precision.

For host memory, the equations look like this:

$$M_h = N * P * 2 \qquad M_h = N * P * 2 \qquad M_h = N * P * 3 \tag{5.2}$$

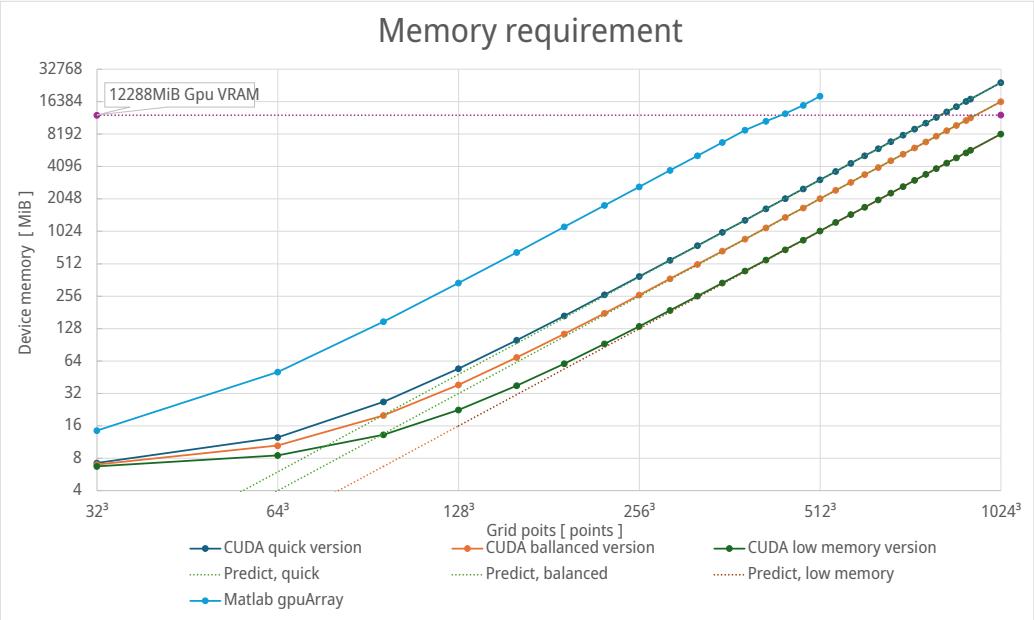

Figure 5.21: Device memory usage comparison of non-expanding grid propagator with calculated predicted values

Compared to the MATLAB gpuArray GPU-accelerated version, the quick CUDA implementation uses approximately one-sixth of the device memory. The difference becomes even more pronounced with the low-memory configuration, with usage approaching just 1/20th.

Next, the nonlinear acoustics version will be compared between the expansion and non-expansion approaches. Memory expectation usage is again described in the equation below. The first column has a quick version for the device, and below for the host memory. The second column again defines the low memory option for device and host memory.

$$\begin{aligned} M_d &= N_e * P * (3 + H) & M_d &= N_e * P \\ M_h &= N_e * P * 3 & M_h &= N_e * P * (2 + H) \end{aligned} \tag{5.3}$$

As before, P denotes the point size in bytes, N_e represents the number of points in the expanded grid, and H is the number of harmonics specified as an argument to the GPU

AFP program. The prediction calculation for the non-expanding option remains the same, except that the parameter N_e is replaced by N, which denotes the number of grid points in the input acoustic field grid. Interestingly, the total memory requirement is slightly higher for the low-memory option. This is expected, as the optimisation targets explicitly a reduction in device memory usage. Fortunately, the increase is minimal, equivalent to the size of a single grid-sized array.

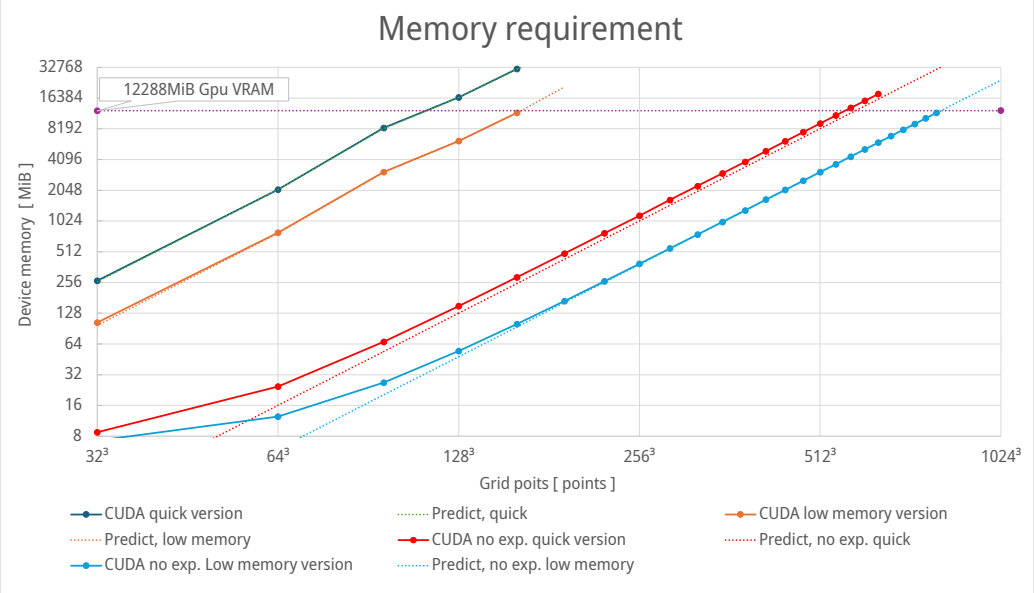

Figure 5.22: Device memory usage comparison of nonlinear acoustic propagator with calculated predicted values for memory usage

The values presented in Figure 5.22 were obtained using the same configuration as the time comparisons: concretely, five harmonics and ten iterations. The graph shows the difference in device memory usage between quick and low memory options. The exact improvement can be extracted from the equations, and it is dependent on the number of harmonics that we are simulating.

Finally, the propagator version with heterogeneous sound speed and absorption in the medium was profiled with NSight Systems. Equations for all versions of this AFP are mentioned below, from the quick to the low memory option and device memory on top, with host memory on the bottom.

$$\begin{array}{lll} M_d = N_e * P * 5 & M_d = N_e * P * 3 & M_d = N_e * P * 2 \\ M_h = N_e * P & M_h = N_e * P * 3 & M_h = N_e * P * 4 \end{array} \tag{5.4}$$

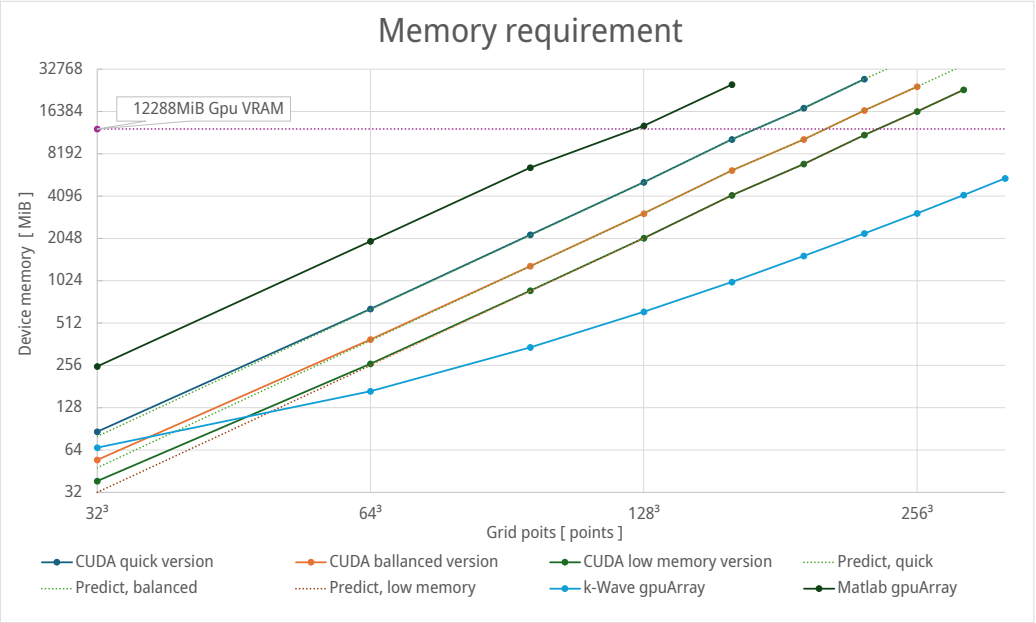

Figure 5.23: Device memory usage comparison of heterogenous propagator with calculated predicted values for memory usage

The graph clearly shows an improvement in my CUDA implementation memory usage from using MATLAB GPU acceleration with gpuArray. On the other hand, the k-Wave CUDA version uses even less memory. This is thanks to the grid expansion that is needed to deal with the wave wrapping problem in the AFP implementation. My implementation uses one-sixth of device memory when comparing MATLAB and CUDA low memory versions.

5.5 Profiling of CUDA kernels

In this section, I will show profiled CUDA kernels with the aforementioned NSight Compute tool. The primary tool for showing this is the roofline model.

The roofline model is a visual and analytical tool used to understand and optimise the performance of compute kernels. It provides a clear framework for evaluating how well a kernel utilises the underlying hardware by plotting achieved performance (in FLOPs per second) against arithmetic intensity (the number of floating-point operations per byte of memory transferred). The model overlays this data with hardware-specific ceilings representing theoretical limits for memory bandwidth and peak computational throughput.

This makes the Roofline model particularly useful for identifying performance bottlenecks. For instance, if a kernel lies below the sloped part of the roofline, it is likely memory-bound, meaning performance is limited by data movement rather than computation. If it lies below the flat portion, it is compute-bound and may benefit from instruction-level optimisations. In the roofline graphs, the lower blue line represents the performance ceiling for double-precision operations, while the upper blue line corresponds to the single-precision computation ceiling.

First, the core propagator function kernel is profiled.

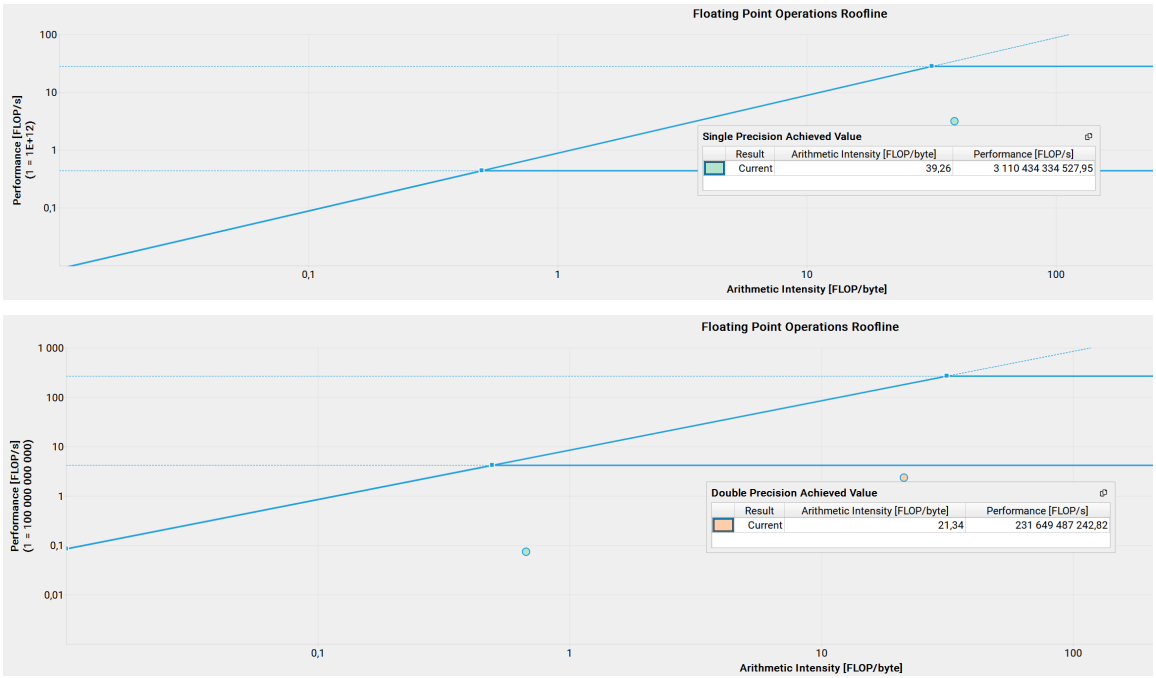

Figure 5.24: Floating Point Operations Roofline for core propagator function kernel
Top: single precision. Bottom: double precision

The figure 5.24 indicates that the kernel is compute-bound. The observed arithmetic intensity is 39.26 FLOPs per byte for the single-precision variant and 21.34 FLOPs per byte for the double-precision variant.

Another measured metric by the NSight Compute is GPU throughput. It measures how effectively the GPU's computational and memory subsystems are being utilised during kernel execution. The single-precision kernel achieved a compute throughput of 58.29% and a memory throughput of 24.41%. In contrast, the double-precision version reached 86.17% compute throughput, while its memory throughput was significantly lower at 2.29%.

The following profiled kernels are from the heterogeneous version of AFP. Specifically, the kernel that takes care of one iteration in the quick version of that propagator. Profiling was performed using a cubic grid with 128 points along each dimension, ensuring full utilisation of the GPU used for the analysis.

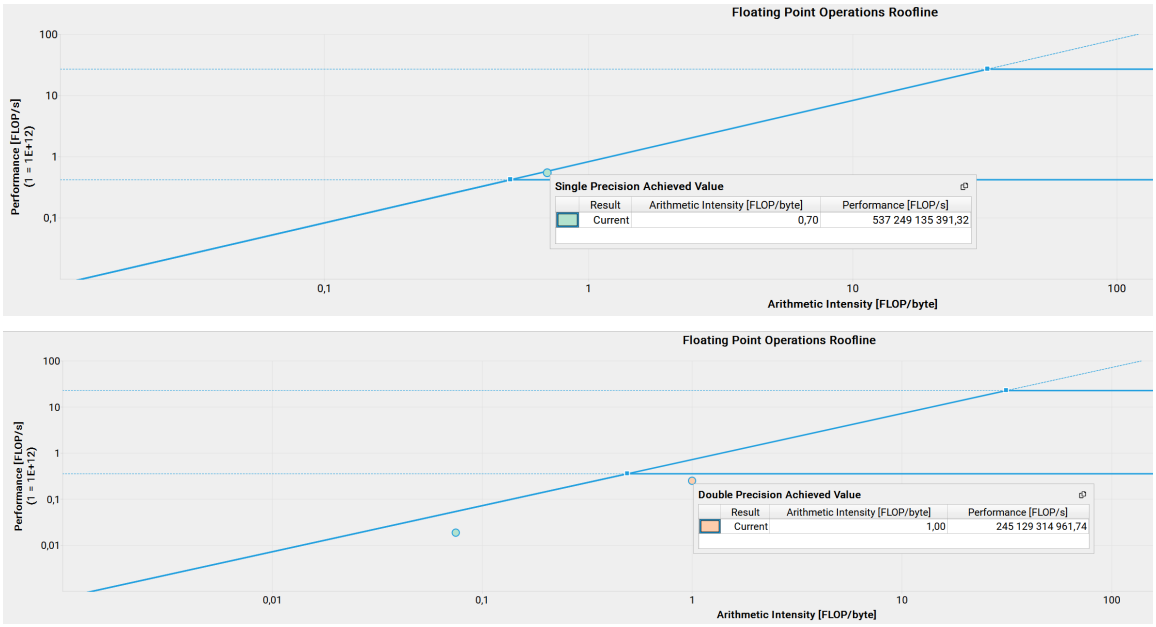

Figure 5.25: Floating Point Operations Roofline for iteration kernel in heterogeneous propagator Top: single precision. Bottom: double precision

The results of this kernel are the complete opposite of the previous one. It is clear that this kernel is not compute-bound, but rather limited by memory bandwidth. Again, the roofline on top is for single precision and the bottom for double. As expected, the arithmetic intensity is lower, specifically, 0.70 FLOPs/byte for single precision and 1.01 FLOPS/byte for double precision. In the single-precision case, the performance aligns closely with the roofline, indicating that the memory throughput limit has been reached. The measured throughput values were 10.63% for compute and 91.16% for memory in single precision, while in double precision, compute throughput was 89.05% and memory throughput reached 32.7%.

The NVIDIA Graph mentioned above can facilitate the parallel execution of operations and minimise the latency between them. This can be verified with the aforementioned tool, NSight Systems, which can profile the whole run of the program. With this tool, we can see the execution of each CUDA API call and the kernel times. To provide an example of this, the version of AFP with heterogeneous acoustics was profiled.

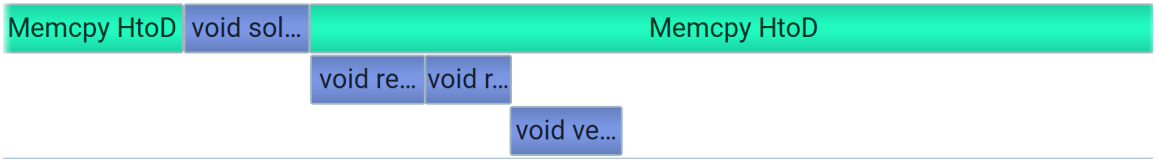

Figure 5.26: Example of concurrent execution of nodes within a CUDA Graph

With the program profile, we can see the parallelisation of some operations. That is, FFT execution and simultaneous copy of precomputed data of the propagator core function.

Chapter 6

Conclusion

This thesis has successfully demonstrated the acceleration of the Acoustic Field Propagator (AFP) using CUDA for GPU architectures, replacing the original MATLAB implementation with a significantly more efficient solution. The work focused on several AFP versions—absorbing with grid expansion, wave wrapping cancellation, nonlinear acoustics, and heterogeneous sound speed and absorption, each presenting unique computational challenges.

During the implementation of the propagators, I have improved my understanding of C++ templates and their usage. This was the most challenging aspect, particularly in implementing templates to function correctly across various versions of compilers and operating systems.

The core propagator function, responsible for most of the computation time in the original implementation, was identified through profiling and then ported to CUDA. Efficient use of GPU resources was achieved by exploiting features such as constant memory, CUDA graphs, and cuFFT for fast Fourier transforms. A modular and extensible C++ framework was designed with template support for varying precision and dimensionality, providing a robust base for future extensions.

The benchmark results showed substantial reductions in computation time and, in some cases, memory usage, particularly for the absorbing and wave-wrapping cancellation variants. The implementation also demonstrated strong agreement with MATLAB's results and alternative methods such as the k-Wave toolbox, validating its numerical correctness.

The goal was to accelerate an existing MATLAB implementation. However, for the nonlinear version, I also added support for wave wrapping cancellation. This brings significant acceleration and allows for simulating bigger domains thanks to the memory usage reduction.

Furthermore, additional tooling, such as a MATLAB MEX interface and HDF5 support, ensures that the solution is compatible with existing workflows in research and development environments. This work not only improves the simulation performance but also lays the foundation for a wider adoption of GPU-accelerated acoustic simulations in real-time and high-resolution applications, especially in medical imaging, non-destructive testing, and validating the creation of wave transmitters.

This thesis implementation could potentially be expanded in the future to utilise multiple GPUs, allowing for the simulation of even larger domains.

The results from this thesis, specifically the acceleration of the wave wrapping cancellation version, will be part of a future journal paper.

Bibliography

[1] Bailly, C. and Bogey, C. An overview of numerical methods for acoustic wave propagation. *European Conference on Computational Fluid Dynamics*, january 2006.

[2] Barton, G. *Elements of Green's Functions and Propagation: Potentials, Diffusion, and Waves*. Clarendon Press, 1989. Oxford science publications. ISBN 9780198519980. Available at: https://books.google.cz/books?id=-iPwVGfDtecC.

[3] Collette, A. *HDF5 for Python* online. 2014. Available at: https://docs.h5py.org/en/stable/. [cit. 2025-05-10].

[4] Cox, B.; Saha, R.; Stanziola, A. and Treeby, B. Rapid computation of steady state acoustic fields in heterogeneous and nonlinear media using the Acoustic Field Propagator. *Acoustical Society of America Journal*, march 2024, vol. 155, no. 1, p. A289–A289.

[5] NVIDIA. *Features and Technical Specifications* online. 2025. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#features-and-technical-specifications-feature-support-per-compute-capability. [cit. 2025-05-10].

[6] NVIDIA. *Type Casting Intrinsics* online. 2025. Available at: https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group__CUDA__MATH__INTRINSIC__CAST.html#type-casting-intrinsics. [cit. 2025-05-10].

[7] Group, H. *Introduction to HDF5* online. 2025. Available at: https://support.hdfgroup.org/documentation/hdf5/latest/_intro_h_d_f5.html. [cit. 2025-05-07].

[8] Joachain, C. J. *Quantum Collision Theory*. North-Holland publishing company Amstrdam, 1975. Oxford science publications. ISBN 978-0-7204-0294-0. Available at: https://www.scribd.com/doc/76844441/Charles-J-Joachain-Quantum-Collision-Theory.

[9] Kaltenbacher, B. Periodic solutions and multiharmonic expansions for the Westervelt equation. *Evolution Equations and Control Theory*, 2021, vol. 10, no. 2, p. 229–247. Available at: https://www.aimsciences.org/article/id/96e95741-1ec4-43dd-a700-3f4d15fb847a.

[10] Osnabrugge, G.; Benedictus, M. and Vellekoop, I. M. Ultra-thin boundary layer for high-accuracy simulations of light propagation. *Opt. Express*. Optica Publishing Group, Jan 2021, vol. 29, no. 2, p. 1649–1658. Available at: https://opg.optica.org/oe/abstract.cfm?URI=oe-29-2-1649.

[11] Osnabrugge, G.; Leedumrongwatthanakun, S. and Vellekoop, I. M. A convergent Born series for solving the inhomogeneous Helmholtz equation in arbitrarily large media. *Journal of Computational Physics*, 2016, vol. 322, p. 113–124. ISSN 0021-9991. Available at: https://www.sciencedirect.com/science/article/pii/S0021999116302595.

[12] Pang, T. *An Introduction to Computational Physics*. Cambridge University Press, 2006. ISBN 9780521825696. Available at: https://doi.org/10.1017/CBO9780511800870.

[13] The MathWorks, I. *C++ MEX Functions* online. 2025. Available at: https://www.mathworks.com/help/matlab/matlab_external/c-mex-functions.html. [cit. 2025-05-16].

[14] The MathWorks, I. *GpuArray Array stored on GPU* online. 2025. Available at: https://www.mathworks.com/help/parallel-computing/gpuarray.html. [cit. 2025-05-16].

[15] timothygiraffe. *How do I use atomicMax on floating-point values in CUDA?* online. 2022. Available at: https://stackoverflow.com/a/72461459. [cit. 2025-05-10].

[16] Treeby, B.; Cox, B. and Jaros, J. *K-Wave A MATLAB toolbox for the time domain simulation of acoustic wave fields* online. 2016. Available at: http://www.k-wave.org/manual/k-wave_user_manual_1.1.pdf. [cit. 2025-05-11].

[17] Treeby, B.; Cox, B. and Jaros, J. *K-Wave A MATLAB toolbox for the time-domain simulation of acoustic wave fields* online. 2022. Available at: http://www.k-wave.org. [cit. 2025-05-16].

[18] Treeby, B. E.; Budisky, J.; Wise, E. S.; Jaros, J. and Cox, B. T. Rapid calculation of acoustic fields from arbitrary continuous-wave sources. *The Journal of the Acoustical Society of America*, january 2018, vol. 143, no. 1, p. 529–537. ISSN 0001-4966. Available at: https://doi.org/10.1121/1.5021245.

[19] Treeby, B. E. and Cox, B. T. A k-space Green's function solution for acoustic initial value problems in homogeneous media with power law absorption. *The Journal of the Acoustical Society of America*, june 2011, vol. 129, no. 6, p. 3652–3660. ISSN 0001-4966. Available at: https://doi.org/10.1121/1.3583537.

[20] Wikipedia contributors. *CUDA — Wikipedia, The Free Encyclopedia*. 2025. Available at: https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1287490412. [Online; accessed 4-May-2025].

[21] Wise, E. S.; Robertson, J. L. B.; Cox, B. T. and Treeby, B. E. Staircase-free acoustic sources for grid-based models of wave propagation. In: *2017 IEEE International Ultrasonics Symposium (IUS)*. 2017, p. 1–4.