FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Parallel computations on orthogonal grids on GPU
Student:	Yury Hayeu
Supervisor:	Ing. Tomáš Oberhuber, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

TNL (Template Numerical Library) is a modern library for development of parallel algorithms for GPUs, multicore CPUs and distributed clusters. One of the most important classes of algorithms is stencil computations on orthogonal grids, together with the finite difference and finite volume methods on orthogonal numerical meshes. The aim of this work is to improve current implementation of grids in TNL and implement new parallel algorithms for convolutions.

1. Get familiar with fundamentals of programming for GPUs in CUDA.
2. Get familiar with basic data structures and parallel algorithms of TNL library.
3. Refactor the current implementation of numerical grids in TNL.
4. Implement CUDA kernels for convolutions on the grids.
5. Implement a numerical solver for the heat equation by means of the finite difference method (FDM).
6. Implement unit tests and compare solution of the heat equation obtained by the FDM and convolution with Gaussian kernel.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 10 December 2021 in Prague.

Bachelor's thesis

PARALLEL COMPUTATIONS ON ORTHOGONAL GRIDS ON GPU

Yury Hayeu

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: Ing. Tomáš Oberhuber, Ph.D.
May 10, 2022

Contents

List of Figures

List of Tables

List of code listings

Throughout the writing of this thesis, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Ing. Tomáš Oberhuber, Ph.D., who explained to me the essential concepts and contributed to the completion of this work.

I also gratefully acknowledge the assistance of my brother, who helped me with continuous reviews of the thesis text part.

Finally, I wish to thank the CTU FIT staff for being open and ready to help during the three years of my study.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 10, 2022 .....

Abstract

Different numerical methods are based on structured orthogonal grids due to their simplicity of implementation. The thesis focuses on improving the implementation of structured orthogonal meshes in the Template Numerical Library (TNL). In the TNL the main problem was the tedious usage of the grid entities' traversal operation and the repetitive code base between grids of different dimensions. To resolve these issues I took the advantage of the grid properties, the features of modern C++ and the latest improvements in the TNL. Then the grid implementation was covered with unit tests and the numerical solver of the heat equation using the finite difference method was implemented. In addition to the numerical solution, the heat equation has a pseudo-analytical solution based on the convolution operator. At the end of the thesis, I implemented a convolution operator optimized for GPU and showed the similarity between the numerical and pseudo-analytical solutions.

Keywords Structured orthogonal grid, Finite difference method, Heat equation, convolution, GPU

Abstrakt

Různé numerické metody používají strukturované ortogonální mřížky kvůli jejich jednoduché implementaci. Tato práce se zaměřuje na zdokonalení implementace strukturovaných ortogonálních mřížek v knihovně Template Numerical Library (TNL). Hlavním problémem implementace strukturovaných mřížek v knihovně TNL je obtížné procházení objektů mřížky a opakující se zdrojový kód pro mřížky různých dimenze. K vyřešení těchto problémů jsem využil vlastností mřížky, vlastností moderního jazyka C++ a poslední vylepšení ve knihovně TNL. Následně byla implementace otestována a byl implementován numerický řešič metodou konečných diferencí pro rovnici vedení tepla. Kromě numerického řešení má rovnice vedení tepla i pseudo-analytické řešení založené na operátoru konvoluce. Na závěr práce jsem implementoval operátor konvoluce optimalizovaný pro GPU a ukázal jsem podobnost numerického a pseudo-analytického řešeni.

Klíčová slova Strukturované ortogonální mřížky, Metoda konečných diferencí, Rovnice vedení tepla, konvoluce, GPU

List of abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
PDE	Partial Differential Equation
TNL	Template Numeric Library
SM	Streaming Multiprocessor
SIMT	Single Instruction Multiple Threads
SIMD	Single Instruction Mutliple Data
DRAM	Dynamic Random Access Memory
FDM	Finite Difference Method

Introduction

Grids (or meshes) are essential tools in computer science with a wide variety of applications. All of them attempt to discretize a domain based on specific rules for calculations. One of the notable members is the orthogonal structured grid. Despite its simplicity, its implementation design has a crucial role in computations. Poorly designed structured grids result in slow calculations or difficult problem declaration. Grids are the main ingredient for the implementation of different numerical methods for solving partial differential equations (PDE). And PDEs are ubiquitous in mathematically oriented fields, such as physics and chemistry. They are widely used in fluid dynamics and other theoretical and practical fields.

Most numerical methods require performing thousands of iterations which makes them computationally expensive. It must be noted, however, that some of them perform the same set of operations simultaneously for each entity in the grid, therefore, they are parallelizable. GPUs are excellent for these tasks.

At the beginning of the thesis, the main goals will be outlined.

The theoretical part starts by describing the main components of the graphics processing unit (GPU) hardware architecture. In addition to that, the thesis covers the basics of programming for CUDA in C++ and the profiling process of CUDA kernels. After this, the thesis concentrates on the description of the Template Numerical Library (TNL). CUDA programming model only provides developers with essential primitives needed to create applications for GPUs. Therefore, different open-source libraries were developed to simplify programming for CUDA. TNL is one such library. The theoretical part ends with the description of the finite difference method. At first, it defines the essential terms needed for understanding the numerical method. Then it describes the process of the finite difference method and applies it to find the solution to the heat equation.

The practical part of the thesis begins with the description of the structured orthogonal grid implementation in the TNL. At the end of the grid design description, the thesis covers the process of testing the implementation. Then the thesis will describe the implementation of the convolution operator and the techniques used in its implementation on CUDA. Afterwards, the thesis will describe the results of the improvements in the TNL library. Finally, it will discuss the benchmark results of the heat equation numerical solver. Additionally, the heat equation has a pseudo-analytical solution, which uses the convolution operator. In the end, the thesis will show the similarities between the heat equation numerical solution and the heat equation pseudo-analytical solution.

Goals

The main goal of the thesis is to improve the implementation of structured orthogonal grids in the Template Numerical Library (TNL). The list of issues that will be addressed as a part of the improvement in the grid implementation in the TNL is the following:

1. TNL implements one-dimensional, two-dimensional and three-dimensional orthogonal grids, but each of them is implemented independently of the other. However, some grid properties don't depend on the grid dimension. Therefore, the goal is to extract dimension-independent properties of the grid to reduce the repetitiveness in the codebase.

2. The main operation performed on the grid is the parallel traversal of the grid entities. The current grid implementation in the TNL uses separate classes to define the traversal operation. Later, the TNL team introduces the optimal parallel for loop algorithm. The issue is that the traverse classes implement a similar algorithm for the traverse, which makes their implementation redundant. The goal is to update traverser classes to use the parallel algorithm.

3. Various numerical solvers will use grid classes to perform computations. The usage of grid classes may add significant run-time overhead to the numerical solver. Therefore, the goal is to estimate the overhead added by the grid environment. To measure it, the numerical solver of the heat equation using the Finite Difference Method (FDM) based on the grid class will be implemented. It will be compared with the numerical solver, which solves the same equation, but uses only primitive functions provided by the TNL.

4. Current grid implementation in the TNL is poorly tested. In addition to that, it must be noted that the TNL actively uses template metaprogramming. The compiler uses templates to generate specific data types at compile time, i.e. the source code will be generated and compiled by the end users of the library. As a result, different specializations of the templates must be tested because of the error possibility in the generated code. The goal is to cover the grid classes with the unit tests.

The compulsory goal of the thesis is to learn CUDA programming. It will be achieved by implementing CUDA kernels for the convolution operator. There are several requirements for the implementation:

1. The convolution operator must be optimal for the small kernels.

2. The convolution operator must support one-, two- and three-dimensional kernels.

3. The convolution operator must be generic, i.e. the user can define any convolution mask.

To fulfill the requirements I will implement different versions of the convolution operator. The goal is to learn CUDA primitives and to understand their effect on the computation efficiency, therefore not all variants of the convolution operator algorithms will be implemented.

Finally, the heat equation has a pseudo-analytical solution, which uses the convolution operator. The goal is to use the implementation of the convolution operator and show the similarity between the numerical solution obtained with FDM and the pseudo-analytical solution to the heat equation.

Chapter 1

CUDA programming

Nowadays, computer systems are heterogeneous, i.e. they consist of the central processing unit (CPU) complemented with a graphics processing unit (GPU), each designed and optimized for specific tasks with separate memory spaces and a shared memory bus.

CPU is a complex device with a complex instruction set architecture designed for the operating system's needs [1]. A modern operating system kernel manages hundreds of user threads concurrently by arranging their instructions in heavy kernel threads. Then kernel assigns them to the CPU core. The parallelism between threads on a single core is achieved by switching between them in short periods of time [2]. Therefore, to be efficient CPU core must run as many instructions as possible with complex and unpredictable branching. To excel at this task CPU core contains specialized circuits, which significantly increase the size of each core on the die. Overall, the CPU has a limited number of cores and can run fewer threads concurrently.

In comparison to CPUs, GPUs are designed for data-parallel tasks. They are standalone coprocessors, thus they can't start the execution of tasks. GPU cores are simpler than CPU ones, which makes them significantly smaller and allows to increase their count on silicon. For example, a modern high-end GPU has 10496 cores [3] and a modern server CPU has only 64 cores [4].

1.1 Hardware Architecture

The architecture of the GPU was firstly introduced with the Tesla model in 2006 by Nvidia. Since then GPU architecture has significantly evolved to excel at various kinds of tasks, but its main components haven't changed much [5]. In system architecture, GPU is a standalone coprocessor, which must schedule and execute workloads defined by the CPU.

1.1.1 Workload execution

GPU architecture is designed around Streaming Multiprocessors (SM). Each streaming multiprocessor is built to execute concurrently hundreds of threads. The GPU achieves a higher level of parallelisms by increasing the count of the SMs on board. Such architecture is easily scalable and allows the execution of thousands of concurrent threads at the same period of time. The essential components of the SM are instruction cache, warp scheduler, register files, shared memory/L1 cache and CUDA cores.

The SM creates, manages, schedules and executes threads in groups of 32 parallel threads called warps. SM uses a warp scheduler to instantiate, terminate and manage warps concurrently. Each warp has a dedicated execution context, which consists of program counters, register file

and shared/L1 memory. Only one warp is executed at the time. The execution context of each warp is persisted on the SM during its lifetime, therefore there is no cost for switching between warps. This allows the warp scheduler to efficiently utilize the SM whenever the execution of warp stalls for any reason (memory load, instruction fetch, synchronization and others). However, this limits the count of warps managed by the multiprocessor.

Individual thread in a warp starts their execution at the same instruction address, but each of them has an individual instruction counter and register state, therefore they can branch and execute themselves independently [6]. Nvidia calls this single instruction, multiple threads (SIMT) architecture, which is the extension of the single instruction, multiple data (SIMD) system architecture by Flynn's taxonomy [7]. The SIMT architecture can be ignored by the developer, then it will be the same as SIMD architecture. However, a significant performance improvement can be achieved by the efficient utilization of it.

To achieve maximal instruction throughput on the warp, each thread in the warp must follow the same execution path. When a warp of threads diverges by a different data-dependent conditional path, the warp will execute each path separably by disabling threads on each path. This can occur only within a single warp. Distinct warps execute independently.

1.1.2 Memory architecture

GPU implements a multi-level memory hierarchy similar to modern CPUs. Global memory is DRAM, which is slow and large. SM can access global memory with memory transactions of 32-, 64- or 128-bits wide. To optimize memory access time GPU architecture uses the principles of spatial locality by implementing a two-level cache. GPU hardware tries to group memory accesses in a single cache line of L2 memory. L2 cache has a cache line of 128 bits wide and can be accessed by multiple SMs simultaneously.

Each SM has a shared memory and L1 cache. The size of both is limited and specified by the GPU architecture. Shared memory is programmable, this makes it extremely useful for various use cases. It is possible to disable shared memory, and then shared memory will be used as L1 cache. Shared memory consists of 32 memory banks, which can be accessed concurrently. Each bank has a bandwidth of 32-bits per clock cycle. Multiple banks boost the efficient memory bandwidth by a multiple of the memory banks count [6]. The size of the element in the bank can be set either to 32-bits or to 64-bits. Each successive word in the shared memory is mapped to the successive memory bank [8]. The index of the bank for a specific word can be calculated using the next equation:

$$\text{bank index} = \frac{\text{address of word} - \text{shared memory address}}{\text{bank size}} \mod \text{banks number}$$

When multiple threads in the warp access elements in the same bank, the memory conflict arises. Hardware will serialize memory requests as if no conflicts occur. This reduces memory bandwidth by a factor of colliding memory requests. Bank conflict doesn't occur when multiple threads in a warp read from the same address. Programs should be carefully designed in order to avoid bank conflicts.

1.2 CUDA

Compute Unified Device Architecture (CUDA) is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in Nvidia GPU. CUDA API is available for multiple languages, such as C, C++, Fortran, Python, Java and others. In addition to this, Nvidia provides developers with a vast collection of libraries for different applications. The whole documentation for the CUDA platform can be found in [6].

CUDA programming model assumes that the program runs on a heterogeneous system architecture composed of a host and a device [6]. The host represented by the CPU performs sequential code, whereas the device represented by the GPU executes tasks specified by the host. Host and device have their own managed memory space. In addition to that, there is no binary compatibility between instructions compiled for each device.

The CUDA toolkit provides developers with NVCC to compile programs defined in C++. NVCC aims to hide the process of compiling CUDA source code from the developers. NVCC is a compiler-driver, therefore it requires a host C++ compiler to compile the host source code [9].

1.2.1 Kernel thread hierarchy

CUDA programming model allows developers to specialize C++ functions called kernels. To specify kernel developers must add `__global__` specifier in function declaration. Inside the kernel, the developer can launch other kernels or call functions specified with `__device__` specifier. Each kernel is executed in parallel by CUDA threads. CUDA thread is the lowest level of abstraction for doing computations and memory operations. CUDA API exposes a two-level thread hierarchy. At the bottom of the hierarchy is a thread. A group of threads organized in one, two or three dimensions forms a block and a group of blocks organized in one, two or three dimensions forms a grid. To launch a kernel the programmer must specify the number of thread blocks and the number of threads per block.

When a kernel is launched, the GPU block scheduler enumerates and assigns a thread block to available SMs. When the SM receives a thread block, it partitions it into successive warps with the first warp containing a thread with zero index [6]. A single thread block can be processed only by a single multiprocessor, therefore there is a limit of 1024 threads per thread block. The size of the warp of 32 threads suggests that multiprocessors can achieve full utilization when thread blocks have the size of a multiple of 32. When the thread block finishes its execution, the next thread block is assigned to the freed multiprocessor. From the host's perspective the kernel is launched asynchronously, i.e. the host doesn't wait for kernel execution. To wait for kernel execution the developer must call `cudaDeviceSynchronize()` function.

In the SIMT architecture threads can cooperate by using either global or shared memory. However, threads don't run the same way physically. Thus, read-after-write, write-after-read or write-after-write hazards may occur, when multiple threads access the same address in global or shared memory. To mitigate hazards within a thread block the developer can specify a synchronization point by calling `__syncthreads()` function. Whenever a thread arrives at the synchronization point, it waits until all threads within a thread block reach the same point [6].

CUDA programming model implements support for vector types. One of such types is `dim3`, which is a vector of 3 integers used to specify dimensions. To identify a thread in the thread hierarchy, the programmer can access `threadIdx`, `blockIdx`, `blockDim` and `gridDim` of `dim3` type in the kernel code. `threadIdx` identifies a thread inside the thread block. `blockIdx` identifies the block in the grid and `blockDim` is the dimension of the block. `gridDim` identifies the dimension of the grid in blocks.

One of the most common operations is the mapping of the n-dimensional thread index to index inside the 1-dimensional global memory block. This can be performed by the next equations:

$$\begin{aligned} ix &= \text{threadIdx}.x + \text{blockIdx}.x * \text{blockDim}.x \\ iy &= \text{threadIdx}.y + \text{blockIdx}.y * \text{blockDim}.y \\ iz &= \text{threadIdx}.z + \text{blockIdx}.z * \text{blockDim}.z \\ \text{index} &= ix \ + iy * \text{blockDim}.x * \text{blockDim}.x + \\ &\quad iz * \text{blockDim}.x * \text{blockDim}.x * \text{gridDim}.y * \text{blockDim}.y \end{aligned} \tag{1.1}$$

1.2.2 Memory

CUDA assumes that the host and device have different memory spaces. CUDA raises an exception when either host or device tries to access the memory space of another. The developer can manage memory allocation of CUDA with functions similar to CPU memory management functions (`malloc`, `memcpy`, `free`, ...): `cudaMalloc`, `cudaMemcpy`, `cudaFree`. `cudaMemcpy` can transfer memory between host and device global memory.

CUDA programming model allows to allocate shared memory either statically or dynamically. The compiler allocates shared memory for all variables declared in the device code with `__shared__` specifier. However, the size of each variable should be known at the compile time. To allocate variables in shared memory dynamically they should be declared with `extern __shared__` keywords. Then the size of the shared memory must be specified at the kernel launch. However, it is only possible to allocate dynamically 1-dimensional arrays [6].

1.2.3 Profiling

CUDA offers different tools to help developers analyze CUDA kernel performance issues. NVProf is one such tool. It can evaluate the runtime of the CUDA kernels inside the CUDA program. The result of the analysis is the list of metrics. Metrics extensively characterize the properties of the CUDA program. The full list of metrics with their description can be found in [10].

Chapter 2

Template Numerical Library

Writing optimal CUDA programs isn't trivial. Firstly, the CUDA programming model provides only intrinsic operations. Secondly, GPUs computation capability has a wide application in different areas, but there aren't any basic data containers, e.g. arrays, matrices, vectors and others. Finally, the developer should determine manually the best launch configuration of the kernel, data alignment in GPU memory, memory access patterns, and thread synchronization within the kernel as these all affect performance. However, it isn't necessary to build everything from scratch. There are a lot of open-source libraries, which aim to simplify CUDA programming. One of such libraries is Template Numerical Library [11]. TNL provides developers with the essential building blocks for high-performance computing on modern hardware, including CPUs, GPUs and distributed systems. Furthermore, TNL actively uses template metaprogramming, therefore the user can tune most of the components for its needs.

2.1 Arrays and vectors

Template Numerical Library offers a large collection of generic data structures. `Array<Value, Device, Index, Allocator>` defined in `TNL::Containers` namespace is one of them. Developers can easily use `Array` data structure to allocate and prepare memory and transfer it between host and device. `Value` parameter specifies the type of the data stored in the array. `Device` parameter can be specified with `TNL::Devices::Host` and `TNL::Devices::Cuda` to declare where data is located. `Index` parameter specifies the index type of the array. `Allocator` parameter specifies the type, which should acquire and release memory. The developer can manage memory block contents by using either `operator[]` or `operator()`. In addition to that, the developer can fill an array with the same value using `operator=`. To transfer data between the host and device the developer must use copy constructors of `Array` [12].

Array data structure manages the ownership of the memory block, i.e. it ensures correct deallocation of it. It is non-copyable, therefore each pass by value will result in an expensive deep copy of the memory block. To safely pass by value `Array` data structure allows to create an `ArrayView` on itself. The `ArrayView` allows accessing the memory block of the `Array`, but it can't manage it. Thus, the developer can perform the same set of operations as the Array, which doesn't manage the original memory block. `ArrayView` consists only of the pointer to the original memory block and its size. Thus, it is lightweight and passing by value will result in shallow copy.

`TNL::Containers::Vector` extends `Array` data structure and adds the requirement of arithmetic type of `Value` parameter. As a result, it can perform element-wise mathematical operations: `+=`, `-=`, `*=`, `/=` and `%=` [13].

2.2 Lambdas

TNL offers different generic algorithms and data structures, which can be extended with user-defined functions. The most common approach is to specify an anonymous function using lambda expressions. Lambda expression constructs a closure, i.e. an unnamed function object, which can capture the environment. Under the hood, the C++ compiler for each lambda expression generates a uniquely named structure with `operator()` containing the body of the lambda. For argument-dependent lookup lambda expression type is declared in the smallest block, class or namespace scope, that contains the lambda expression [14].

CUDA kernel can execute only `__device__` functions, therefore it can't launch functions generated from lambda expressions, because they represent host code. Only with CUDA 7.5 it is possible to declare `__device__` lambda expressions. However, `__device__` lambda expressions can't be called from a host. And in CUDA 8.0 it is possible to specialize `__host__` and `__device__` lambda expressions, which the host and device can call [15]. TNL aims to provide unified interface for both CPU and GPU, therefore, both host and device must call the lambda expression function. Hence, TNL introduces `__cuda_callable__` specifier, which is the macro on `__host__` and `__device__` specifiers.

▶ **Definition 2.1** (Extended lambda expression)**.** *Lambda expression is extended, when it is annotated with either `__device__` or with both `__host__` and `__device__` specifiers and defined in the immediate or block nested scope of the `__host__` or `__host__ __device__` function [6].*

CUDA programming model restricts extended lambda expressions. Firstly, an extended lambda expression must capture the environment by value. Secondly, an extended lambda expression can't be generic. There are more restrictions and the whole list of them can be found in [6].

`__host__ __device__` lambda expressions may reduce the performance when they are called from host. The main reason is that the CUDA compiler wraps them in the `std::function`. The host compiler may not be able to inline the body of such lambda expression. Thus, it will be evaluated as a separate function call [15].

2.3 Parallel for loops

Loop-level parallelism is one of the simplest forms of parallelism. For loop can be executed in parallel when there are no dependencies between iterations in the loop. TNL implements a parallel for loop algorithm for both CPU and GPU and provides a simple way to use it. The implementation of parallel for loop uses OpenMP library on the host and uses CUDA on the device.

To use the algorithm the developer must call the `exec()` function in the `ParallelFor{1|2|3}D <Device, Mode>` defined in the `TNL::Algorithms` namespace. `Device` template parameter of the structure declares the device, which should run the algorithm. `Mode` template parameter can be either `SynchronousMode` or `AsynchronousMode`. In the synchronous mode, the program will return only after successful algorithm execution. In the asynchronous mode, the program will receive control immediately. `exec()` function must be called with the `[start, end)` iteration range for each dimension and the body function object to execute in parallel. The algorithm will call body function with the iteration index of parallel for loop. Optionally, additional parameters can be passed to the `exec()` function, which will be passed to the body function call [16].

2.4 Parallel reduction

The reduction operator produces a single value by applying a binary operator to the input sequence. Since the binary operator can be any binary function, the reduction operator has a wide range of applications. Its sequential implementation is simple though the parallel one is more complicated. TNL implements parallel reduction for both CPU and GPU and provides a simple way to use it. The implementation of parallel for loop uses OpenMP library on the host and uses CUDA on the device.

The developer must call `reduce()` function defined in the `TNL::Algorithms` namespace to perform parallel reduction. The `reduce()` function can be specialized with the `Device`, `Index`, `Result`, `Fetch` and `Reduce` template parameters. `Device` template parameter declares the device to execute parallel reduction. `Index` declares the sequence index type and `Result` declares the result type. `Fetch` declares the function type, which fetches the element at the index from the input sequence. `Reduce` declares a function type, which applies a binary operator on the two values. TNL defines a set of the most common `Reduce` function objects. Some of which are `TNL::Plus`, `TNL::Minus`, `TNL::Multiply` and `TNL::Divide`. The `reduce` function must be called with the start and the end indices of the input range, fetch and reduce function objects and the identity element. The identity element is the neutral element, which can't affect the result of the reduce operation.

Sometimes, with the reduce operator we need to determine the index of the element in the sequence. Therefore, TNL offers `reduceWithArgument()` function, which will call `Reduce` function with the reference on the accumulator and the current index with the index of the variable and variable itself [17].

Chapter 3

Finite Difference Method

In various industrial and scientific areas, the understanding of dynamic processes is essential. A process is dynamic when it constantly changes and evolves. In general, most dynamic processes are complicated. Most of them, therefore, don't have any analytical solutions. Hence different numerical methods were created to approximate the solution. One of such methods is the Finite Difference Method (FDM). The FDM approximates the solution for problems, which involves differential equations.

3.1 Differential Equations

The derivative of the function is one of the most basic operations of calculus. The intuition behind it is that it expresses the change rate of the original function at each point.

▶ **Definition 3.1** (Ordinary derivative)**.** *Let f be a real function of a real variable defined in the neighbourhood of the point x_0. The value of the following limit (if it exists and is finite)*

$$\lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} \tag{3.1}$$

is the derivative of the function f at the point x_0. It is written as $f'(x)$ [18].

Most processes include multiple variables, therefore a function of a single variable is not sufficient to describe the process. The definition of the function and the derivative of the function can be extended to work with the multiple variables.

▶ **Definition 3.2** (Multivariate function)**.** *A mapping f: $\mathbb{R}^n \to \mathbb{R}$ is a real function of n real variables.*

The derivative of the multivariate function must be taken for function variables. It expresses the change rate of selected variables assuming that all other variables are fixed, i.e. they are constant.

▶ **Definition 3.3** (Partial derivative)**.** *Let f be a real function of $(x_0, x_1, ..., x_n)$ real variables defined in the neighbourhood of the point $a = (a_0, a_1, ..., a_n)$. Then the partial derivative of the first order with the respect to the variable x_i is the value*

$$\lim_{x_i \to a_i} \frac{f(x_0, \ldots, x_i, \ldots, x_n) - f(x_0, \ldots, a_i, \ldots, x_n)}{x_i - a_i} \tag{3.2}$$

if the limit exists. It is written as $\frac{\partial f}{\partial x_i}$ [19].

▶ **Definition 3.4** (Differential equation)**.** *A differential equation is an equation containing the derivatives of one or more unknown functions (or dependent variables), for one or more independent variables [20].*

▶ **Definition 3.5** (Partial differential equation)**.** *A differential equation is partial if it contains partial derivatives of one or more unknown functions for two or more variables [20].*

3.2 Numerical Grids

The FDM uses numerical grids to discretize continuous PDEs. A grid is a geometric entity defined in the geometric space. The most common geometric space is the Euclidean. One of the characteristics of the geometric space is its dimensionality. Informally, the dimension represents a minimum number of parameters to identify the geometric entity.

The grid is formed by covering the n-dimensional space with the n-dimensional geometric entities. The geometric entities are polyhedrons, polygons, line segments and vertices, which represent three, two, one and zero-dimensional entities accordingly. Informally, the n-dimensional geometric entity is formed by the closed arrangement of entities of lower dimensions. For example, the cube is formed by 8 vertices, 12 edges and 6 faces or the square is formed by 4 vertices and 4 edges. A numerical grid uniquely identifies its entities using coordinates, i.e. defines a coordinate system. Depending on the numerical the method grid must identify components of entities. For example, the 3-dimensional grid formed by 3-dimensional cubes must uniquely identify the edges of each cube.

Grids can be classified as structured or unstructured, orthogonal or non-orthogonal. The grid is structured if each grid entity has a consistent number of adjacent entities, otherwise, it is unstructured. Structured grids have simple indexing, therefore the entity address in memory can be often calculated in a constant time. Unstructured grids have complex indexing of their entities, therefore the entity address in memory must be additionally stored. The grid is orthogonal if all grid lines intersect at a right angle, otherwise, it is non-orthogonal.

One of the elementary grid types is a structured orthogonal grid formed by covering the n-dimensional space with the n-dimensional hypercubes. Such grids use a Cartesian coordinate system to identify their entities. The Cartesian coordinate system identifies points by the pair of numerical coordinates. The coordinate origin is the intersection of the oriented orthogonal lines and is defined with the $(0, \ldots, 0)$ coordinate pair.

The FDM replaces the continuous domain with the grid. The example of the two-dimensional grid, where n is the width of the grid and m is the height of the grid, is in Figure 5.1. The grid entities are the vertices identified with the (i, j)-coordinates. The h_x and h_y determine the spacing between vertices in the grid.

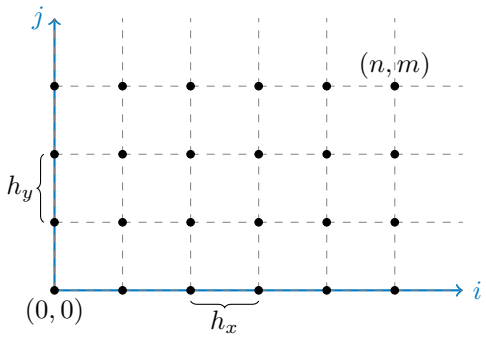


Figure 3.1 The two-dimensional finite difference grid

3.3 Finite Difference Method

Most of the physical processes can be represented as the system of PDEs or ODEs in a continuous domain. The process of solving PDEs using FDM consists of 2 steps. The first step is the discretization of the continuous PDEs using numerical grids. As a result, each continuous differential equation can be approximated as the set of the difference equations for each entity. The second step is to solve the obtained system of algebraic equations [21].

3.3.1 Taylor series

The FDM process obtains the set of difference equations using Taylor series expansion.

▶ **Definition 3.6** (Taylor polynomial). *Let f be a real function of a real variable and f is n-times differentiable at the point x_0. Then there exists exactly one polynomial $T_{n,a}$ of degree at most n such that*

$$T^{(k)}_{n,x_0}(x_0) = f^{(k)}(x_0),\ \forall k = 0, 1, \ldots, n. \tag{3.3}$$

The polynomial $T_{n,x_0}(x)$ is the n-th Taylor's polynomial of function f expanded about the point x_0 and has the following form

$$T_{n,x_0}(x) = \sum_{k=0}^{n} \frac{(x - x_0)^k}{k!} f^{(k)}(x_0). \tag{3.4}$$

For further steps, the alternative form of the Taylor polynomial can be obtained by substitution $h = x - x_0$ and $x = x_0 + h$. The Taylor polynomial transforms into the next equation:

$$T_{n,x_0}(x_0 + h) = \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0) = f(x_0) + \sum_{k=1}^{n} \frac{h^k}{k!} f^{(k)}(x_0). \tag{3.5}$$

▶ **Definition 3.7** (Taylor formula). *Let f be a real function of a real variable and f is n-times differentiable at the point x_0. Let $\forall x \in D(f)$ be $R_{n,a}(x) = f(x) - T_{n,a}(x)$, then the relation*

$$f(x) = T_{n,a}(x) + R_{n,a}(x) \tag{3.6}$$

is the Taylor formula and the $R_{n,a}(x)$ is the n-th remainder [22].

▶ **Definition 3.8** (Langrange's form of remainder). *Let f be a real function of real variable and function f is $(n+1)$-times differentiable on the interval containing x_0 and x. Then the remainder $R_{n,a}(x)$ can be expressed as follows*

$$R_{n,x_0}(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1} \tag{3.7}$$

where ξ depends on x and n and ξ lies between x_0 and x. This form is called n-th Langrange's form of the remainder.

In the following text, the main requirement on the function f is that it has n-th derivative on some open interval, which contains x and x_0. To not repeat it, the function of the differentiability class is defined in the following way.

▶ **Definition 3.9** (The function of differentiability class). *Let f be a real function of real variable and $n \geq 0$. The function f is of differentiability class n on the interval (a, b) if and only if it has continuos 0-, 1-, …, n-th derivatives on the interval (a, b). It is written using the following notation.*

$$f \in C^n(a,b) \tag{3.8}$$

The function f of differentiability class n on any interval containing $x_0, x_1, \ldots$ points is written using the following notation.

$$f \in C^n[x_0, x_1, \ldots] \tag{3.9}$$

In Section 3.3.2, the Lagrange's form of the remainder will be approximated using the theorem 3.10.

▶ **Theorem 3.10.** *Let $f \in C^{n+1}[x_0, x]$, then the Langrange's form of remainder is at most of order $(x - x_0)^{n+1}$ or*

$$R_{n,x_0}(x) = O((x - x_0)^{n+1}). \tag{3.10}$$

Proof. The n-th Lagrange's form of the remainder is the following.

$$R_{n,x_0}(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1} \tag{3.11}$$

To prove the theorem it is enough to show, that the absolute value of the remainder is upper bounded with the constant multiple of the function $(x - x_0)^{n+1}$.

$$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+, \forall n \geq n_0 \colon \left|R_{n,x_0}(x)\right| \leq c\left|(x - x_0)^{n+1}\right| \tag{3.12}$$

Function f has continous $(n+1)$-th derivative on the interval between x_0 and x, therefore its values can be upper-bounded with the constant.

$$\exists c \in \mathbb{R}^+, \left|f^{(n+1)}(\xi)\right| \leq c \tag{3.13}$$

In the end, it is enough to show, that the value of the remainder is at most of the order of the $(x - x_0)^{n+1}$.

$$\left|R_{n,x_0}(x)\right| = \left|\frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}\right| \leq \frac{c}{(n+1)!}\left|(x - x_0)^{n+1}\right| \leq k\left|(x - x_0)^{n+1}\right|, k \in \mathbb{R}^+ \tag{3.14}$$

◀

3.3.2 Derivatives approximations

Let $f \in C^2[x, x+h]$, then first Taylor polynomial expanded at point x at $x + h$ is

$$f(x+h) = T_{1,x}(x+h) + R_{1,x}(x+h) = f(x) + hf'(x) + R_{1,x}(x+h). \tag{3.15}$$

The solution for $f'(x)$ from (3.15) is the

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{R_{1,x}(x+h)}{h}. \tag{3.16}$$

Following the theorem (3.10) the first derivative approximation of function f using Taylor expansion is

$$f'(x) \approx \underbrace{\frac{f(x+h) - f(x)}{h}}_{d(x)} + O(h). \tag{3.17}$$

The equation (3.17) is called the forward difference approximation. The first-order derivative of function f is approximated as the sum of the difference term $d(x)$ and the error term $O(h)$.

In practice, only the difference term $d(x)$ is calculated. The difference term $d(x)$ is the slope of the function f.

Let $f \in C^2[x, x-h]$, then first Taylor polynomial expanded at point x at $x-h$ is

$$f(x-h) = f(x) - hf'(x) + R_{1,x}(x-h). \tag{3.18}$$

The solution from (3.18) is the following.

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{R_{1,x}(x-h)}{h} \approx \frac{f(x) - f(x-h)}{h} + O(h) \tag{3.19}$$

As a result, the first-order derivative is approximated with values of the function at x and $x-h$ points. The equation (3.19) is called the backward difference approximation.

Let $f \in C^2[x, x+h, x-h]$ and let substract (3.18) from (3.15).

$$f(x+h) - f(x-h) = 2hf'(x) + R_{1,x}(x+h) - R_{1,n}(x-h) \tag{3.20}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \frac{R_{1,x}(x+h) - R_{1,x}(x-h)}{2h} \approx \frac{f(x+h) - f(x-h)}{2h} + O(h) \tag{3.21}$$

In addition to this, it can be shown, that the error term in the same finite difference approximation is $O(h^2)$ for $f \in C^3[x, x-h, x+h]$. To show this the second Taylor polynomial expanded at x_0 must be considered and a similar procedure must be performed. It is omitted from the text because of its repetitiveness. Overall, the final approximation for $f \in C^3[x, x-h, x+h]$ is known as the central difference approximation and has the following form.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \tag{3.22}$$

The best approximation is achieved when the error term is minimal. The error term is minimized by reducing the grid step size h. However, a smaller step size implicates a larger number of grid entities to cover the same domain, i.e. it is more computationally expensive. The error term in the forward and backward difference approximations scales linearly. In comparison the error term in the central difference approximation has asymptotic quadratic scaling, e.g. if the grid step size is divided by two, the error term is proportionally divided by 4. Hence, the central difference approximation is asymptotically more accurate, than the forward and backward difference approximations.

Let $f \in C^3[x, x+h, x+2h]$, then the forward difference approximation of the second-order derivative is obtained by considering the second Taylor polynomial of function f expanded at x at $x+h$ and $x+2h$.

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!} f^{(2)}(x) + R_{2,x}(x+h) \tag{3.23}$$

$$f(x+2h) = f(x) + 2hf'(x) + \frac{4h^2}{2!} f^{(2)}(x) + R_{2,x}(x+2h) \tag{3.24}$$

Then the equation (3.23) is multiplied by 2 and substracted from the equation (3.24).

$$-2f(x+h) + f(x+2h) = -f(x) + h^2 f^{(2)}(x) + R_{2,x}(x+2h) - 2R_{2,x}(x+h) \tag{3.25}$$

$$\begin{aligned} f^{(2)}(x) =& \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} + \frac{R_{2,x}(x+2h) - 2R_{2,x}(x+h)}{h^2} \\ \approx& \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} + O(h) \end{aligned} \tag{3.26}$$

The equation (3.26) is the forward difference approximation for the second-order derivative of function f. The backward difference approximation can be obtained by performing similar steps for $f \in C^3[x, x-h, x-2h]$.

Let $f \in C^3[x, x-h, x+h]$, then the central difference approximation of the second-order derivative is obtained by considering the second Taylor polynomial of function f expanded at x at $x-h$ and $x+h$.

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f^{(2)}(x) + R_{2,x}(x+h) \tag{3.27}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f^{(2)}(x) + R_{2,x}(x-h) \tag{3.28}$$

Then by adding the equation (3.27) to the equation (3.28) the following result is received.

$$f(x+h) + f(x-h) = 2f(x) + h^2 f^{(2)}(x) + R_{2,x}(x+h) + R_{2,x}(x-h) \tag{3.29}$$

$$\begin{aligned} f^{(2)}(x) =& \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \frac{R_{2,x}(x+h) + R_{2,x}(x-h)}{h^2} \\ \approx& \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h) \end{aligned} \tag{3.30}$$

In addition to this, it can be shown, that the error term in the same finite difference approximation is $O(h^2)$ for $f \in C^4[x, x-h, x+h]$. To show this the second Taylor polynomial expanded at x_0 must be considered and a similar procedure must be performed. It is omitted from the text because of its repetitiveness. Overall, the final approximation for $f \in C^4[x, x-h, x+h]$ is known as the central difference approximation of the second-order derivative and has the following form.

$$f^{(2)}(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2) \tag{3.31}$$

The approximations of the higher-order derivatives can be obtained by performing similar procedures. For multivariate functions, the partial derivatives are approximated the same way. In Section 3.5, it will be shown by applying the FDM to obtain the solution of the heat equation.

3.4 Convolution

▶ **Definition 3.11** (Convolution operator)**.** *Let f and g be real functions of n real variables. Then the function h defined by*

$$h(\mathbf{x}) = \int_{\mathbb{R}^n} f(\mathbf{x}-\mathbf{y})g(\mathbf{y})d\mathbf{y} = \int_{\mathbb{R}^n} f(\mathbf{y})g(\mathbf{x}-\mathbf{y})d\mathbf{y} \tag{3.32}$$

is the convolution of functions f and g. It is denoted by symbol $f * g$ [23].

The convolution operator has a wide range of applications in statistics, signal and image processing and other areas. In most cases there is no analytical solution for the integral, therefore different numerical methods are applied to approximate convolution operation. One of the ways to approximate the integral is by using the Rectangle rule [24].

At first, to calculate the convolution operator using the Rectangle rule the finite intervals over which the operator is calculated should be chosen. Let $\forall i \in \{1, \ldots, n\}[a_i, b_i]$ be a finite intervals over which the convolution operator is calculated. Then the integral is discretized on the chosen intervals. The Rectangle rule explicitly states, that there must be equal spacing between the discrete observations. In other words, the interval is split into equal subintervals and the length

of the subinterval can be calculated in the following way. Let M_i represents the number of the subintervals of the interval $[a_i, b_i]$. Then the length of subinterval h_i is the following

$$h_i = \frac{b_i - a_i}{M_i}. \tag{3.33}$$

Finally, the discrete convolution obtained with the Rectangle rule has the following form

$$(\prod_{k=1}^{n} h_k) * \sum_{i_1=1}^{M_1} \cdots \sum_{i_n=1}^{M_n} f(\mathbf{x} - (\mathbf{a} + \mathbf{hi}))g(\mathbf{a} + \mathbf{hi}). \tag{3.34}$$

The $\mathbf{a}$ or $(a_1, \ldots, a_n)$ is the vector with the beginnings of the intervals and has the following form. The $\mathbf{i}$ or $(i_1 - 1, \ldots, i_n - 1)$ is the index vector, which represents the indices of the sums. The $\mathbf{h}$ or $(h_1, \ldots, h_n)$ represents the vector of subinterval lengths.

In practice, f is chosen the way, that it has a significant value near the coordinate origin and fastly converges to zero by moving away from it. Therefore, these functions can be numerically approximated with a small number of discrete observations and they are called kernels. The g function represents the data on which the convolution is calculated. Because of the data function finite representation, the convolution operator is undefined for boundary elements. This is called a boundary value problem. There are several methods to address the boundary value problem. One of the simplest solutions for the boundary value problem is to consider, that the data function is zero when it is undefined.

3.5 Heat equation

▶ **Definition 3.12** (Heat equation)**.** *For the space $\Omega \subset \mathbb{R}^n$ the heat equation is the PDE defined as*

$$\begin{aligned} \frac{\partial u(\boldsymbol{x}, t)}{\partial t} &= \Delta u(\boldsymbol{x}, t) \text{ on } \Omega \times (0, T\rangle, \\ u(\boldsymbol{x}, t) &= u_0(\boldsymbol{x}) \text{ on } \Omega, \\ u(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t) \text{ on } \partial\Omega \times \langle 0, T\rangle, \end{aligned} \tag{3.35}$$

where the Laplacean operator Δ is defined as

$$\Delta u(\boldsymbol{x}) = \sum_{i=1}^{n} \frac{\partial^2 u(\boldsymbol{x})}{\partial x_i^2}. \tag{3.36}$$

The function $u_0(\boldsymbol{x})$ is the initial condition and the function $g(\boldsymbol{x}, t)$ is the Dirichlet boundary condition defined on the $\partial\Omega$.

The heat equation describes the diffusion of heat in material over time. The initial condition $u_0(x)$ describes the initial state of the process. The boundary condition denotes, how the PDE is solved on the boundary of the space Ω. The selection of the appropriate boundary condition is essential for obtaining the correct numerical solution of the PDE.

Beforehand only the approximations for the derivatives of single-variable functions were obtained. The same principles apply to partial derivatives to a single variable. In the heat equation, there are two kinds of derivatives. The time derivative ∂t denotes the change of the function over time. The space derivative ∂x^2 denotes the change of the function to the space. To obtain a numerical solution both time and space derivatives are approximated independently.

The first step in the FDM is to replace the continuous domain with the finite difference grid. The h_i denotes the space step of the grid along i-th axis and Δt is the time step for the numerical method. The second step is to obtain difference equations using Taylor series expansion. The time derivative difference equation is obtained using forward difference approximation of the first-order derivative (3.17) and is the following.

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} \approx \frac{u(\boldsymbol{x},t+\Delta t)-u(\boldsymbol{x},t)}{\Delta t} \tag{3.37}$$

The space derivative difference equation is obtained using central difference approximation of the second-order derivative (3.31) and is the following.

$$\forall i \in \{1,\dots,n\} : \frac{u(\boldsymbol{x},t)}{\partial x_i^2} \approx \frac{u(\boldsymbol{x}+\boldsymbol{h},t)-2u(\boldsymbol{x},t)+u(\boldsymbol{x}-\boldsymbol{h},t))}{h_i^2}, \boldsymbol{h}=(0,\dots,h_i,\dots,0) \tag{3.38}$$

By restricting to the $\mathbb{R}^2$ space the following numerical scheme of the heat equation is obtained.

$$\frac{\mu_{i,j}^{k+1}-\mu_{i,j}^{k}}{\Delta t} = \frac{\mu_{i+1,j}^{k}-2\mu_{i,j}^{k}+\mu_{i-1,j}^{k}}{h_x^2} + \frac{\mu_{i,j+1}^{k}-2\mu_{i,j}^{k}+\mu_{i,j-1}^{k}}{h_y^2} \tag{3.39}$$

The $\mu_{i,j}^{k}$ is the value of the heat function at the iteration k with the coordinates (i,j) in the finite difference grid with dimensions (n,m). By denoting the $\mu_{i,j}^{k+1}$ from the equation (3.39) and by adding the initial and boundary conditions the numerical scheme for the heat equation using FDM is the following.

$$\begin{aligned}
&\mu_{i,j}^{k+1} = \mu_{i,j}^{k} + \frac{\Delta t}{h_x^2}(\mu_{i+1,j}^{k}-2\mu_{i,j}^{k}+\mu_{i-1,j}^{k}) + \frac{\Delta t}{h_y^2}(\mu_{i,j+1}^{k}-2\mu_{i,j}^{k}+\mu_{i,j-1}^{k}),\\
&\mu_{i,j}^{0} = u((i,j),0), && \text{(Initial condition)}\\
&\mu_{p,q}^{k} = g((p,q),k), \forall p,q : p \in \{0,n-1\}, q \in \{0,\dots,m-1\}, && \text{(Horizontal boundaries)}\\
&\mu_{p,q}^{k} = g((p,q),k), \forall p,q : p \in \{0,\dots,n-1\}, q \in \{0,m-1\}. && \text{(Vertical boundaries)}
\end{aligned} \tag{3.40}$$

The numerical scheme starts at iteration $k=0$ with the initial values of the function. Then it calculates the $\mu_{i,j}^{k+1}$ using the values from the k-th iteration until it reaches the final iteration defined by the user.

In the $\mathbb{R}^2$ space the heat equation has the following pseudo-analytical solution [25].

$$u(\boldsymbol{x},t) = \int_{\mathbb{R}^2} G_{\sqrt{2t}}(\boldsymbol{x}-\boldsymbol{y})u_0(\boldsymbol{y})\mathrm{dy} = \big(G_{\sqrt{2t}} * u_0\big)(\mathbf{x}) \tag{3.41}$$

The G_σ is the Gaussian kernel, which has the following form.

$$G_\sigma = \frac{1}{2\pi\sigma^2}\exp(-\frac{|\boldsymbol{x}|^2}{2\sigma^2}) \tag{3.42}$$

The pseudo-analytical solution can be interpreted as the smoothing of the heat function by the Gaussian kernel.

Chapter 4
Grid

The TNL provides `Grid` and `GridEntity` class templates to work with structured orthogonal grids. From the implementation perspective, the library aims to minimize the runtime overhead by implementing statically resolved types. As a result, the compiler performs more compile-time optimizations. In addition to that, both `Grid` and `GridEntity` aim to be lightweight and store only essential information. The TNL supports one-, two- and three-dimensional grids. The main operation performed on the grid is the parallel traverse of its entities. The grid can traverse all, boundary and interior entities concurrently. Additionally, the grid positions entities in space and allows them to operate with the entity neighbours. The grid entity is not only the cell or the vertex of the grid, but the grid supports any subentity of the grid cell. The TNL implements a projected coordinates system. Each entity is indexed with the Cartesian coordinate, and then the coordinate is projected to its position in space.

Most of the values in the grid and grid entity are represented as vectors of the grid dimension length. Hence, grid defines `Coordinate` and `Point` types, which represent the grid dimension-length vector of integer and real values accordingly.

4.1 Grid entity

The `GridEntity<Grid, EntityDimension>` class template represents the entity of a structured grid. The `EntityDimension` template parameter specifies the dimension of the grid entity. Possible dimensions of the grid entity are zero or the value lower or equal to the dimension of the grid. The zero-dimensional grid entity represents a vertex, the one-dimensional grid entity represents an edge and so on. The `GridEntity` is highly coupled with the grid it is defined in. Therefore, the `Grid` template parameter references the type of the `Grid` class. From the `Grid` class type the `GridEntity` gets the information about the dimension of the grid, index and real types. The grid entity stores only essential information needed: the reference on the grid, the coordinates of the entity in the grid, its global index, orientation and basis (described in Section 4.1.3).

4.1.1 Coordinates

The grid indexes every entity using the Cartesian coordinate system. In every cell, a coordinate is assigned to the vertex in the cell, which is the closest to the coordinate origin. And the coordinate is the index of the vertex in the sequence of cells. The example of the coordinates is shown in Figure 4.1 The type of the coordinate is the `Coordinate`. However, the coordinate is not capable of identifying all grid entities. To show this let's consider 2-dimensional grid from Figure 4.1. $(0, 0)$ is the coordinate, which will uniquely identify both the vertex and the grid

cell. However, there are two types of edges which go from vertex $(0, 0)$, therefore the coordinate is not sufficient, which led to the introduction of the orientation.

The user can get coordinates from both the device and the host using the `getCoordinates()` method. Additionally, the grid entity coordinates can be updated using the `setCoordinates()` method.

4.1.2 Orientation

The orientation represents the position of the entity relative to grid axes and is of integer type. For example, let's consider a 3-dimensional grid. There is only one way to position cell and vertex to grid axes, therefore they have the orientation 0. There are 3 ways to position edges and faces to axes, therefore there are 3 possible orientations.

From the other point of view, the orientation characterizes the linear subspace to which the entity belongs. To understand this more, let's show how the linear subspace is denoted. The Cartesian coordinate system is a linear space, which is spanned by a set of linearly independent basis vectors. All vectors in the set are unique, otherwise, they aren't linearly independent. The selection of k-vectors from the basis forms a linearly independent set of vectors, which spans a linear subspace. Using previous observations to generate all linear subspaces the enumeration of $(0, 1, \ldots, n)$-combinations is performed. Each n-dimensional entity of the grid either belongs to the linear subspace of the corresponding dimension or the linear manifold collinear to it. The orientation of n-dimensional entity identifies the combination index in the lexicographically sorted sequence of n-combinations from basis vectors. All possible orientations are represented in Figure 4.1.

The entity dimension and the orientation uniquely identify the *entity type*. Thus, both entity dimension and the orientation should be the template parameters in the grid entity type. However, this affects the traverse operations. The traverse operations must accept the lambda function by design, which will be called with the grid entity. At the moment of writing, the CUDA programming model didn't support generic lambdas. Therefore to traverse entities, the user must specify a lambda function for every orientation. However, entities of different orientations are generally processed similarly. Hence the grid entity class stores its orientation.

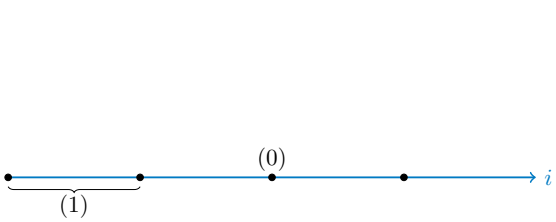
The user can access the orientation of the entity from both the device and the host by using the `getOrientation()` method of the `GridEntity`. In addition to that, the grid entity orientation can be updated by the `setOrientation()` method.

4.1.3 Basis

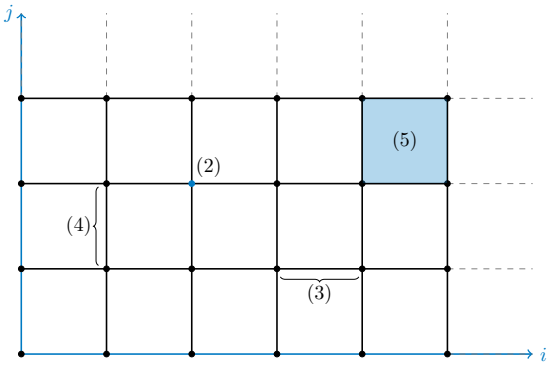
The basis is the n-dimensional boolean vector, which characterizes the grid entity type. The value at index i in the vector is 0 if the entity spans along the i-th axis, otherwise, it is 1. The examples of bases are listed in Figure 4.1. The type of basis is the `Coordinate`.

It must be mentioned, that the basis is superfluous to the entity dimension and the entity orientation. Nonetheless, the basis has an important role in the traversal operations, verifying if the entity is a boundary and accessing the neighbour entities. The main reason is the relationship between the entity type and the number of corresponding entities in the grid. The grid stores its dimensions as the number of edges along its axes. The basis possesses the following property: the grid dimensions are formed by entities of a specific dimension and the number of entities along grid axes is given by summing up the basis and grid dimensions. In the following text, the term *grid entities dimensions* will mean the dimension of the grid aligned with the basis. For example, the dimensions of a 2-D grid on Figure 4.1 are $(5, 3)$. The basis of the vertex is $(1, 1)$ and the number of vertices along axes is the following.

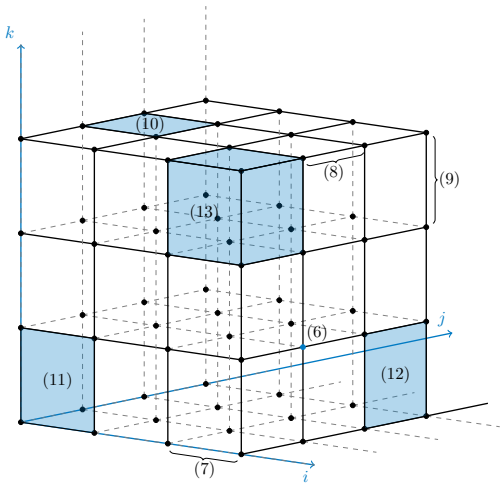
$$(5, 3) + basis = (5, 3) + (1, 1) = (6, 4) \tag{4.1}$$



a. One-dimensional grid



b. Two-dimensional grid



c. Three-dimensional grid

ID	Grid Dim.	Ent Dim.	Coordinate	Orientation	Basis
0	1	0	(2)	0	(1)
1	1	1	(0)	0	(0)
2	2	0	(2, 2)	0	(1, 1)
3	2	1	(3, 1)	0	(0, 1)
4	2	1	(1, 1)	1	(1, 0)
5	2	2	(4, 2)	0	(0, 0)
6	3	0	(3, 1, 1)	0	(1, 1, 1)
7	3	1	(2, 0, 0)	0	(0, 1, 1)
8	3	1	(3, 1, 3)	1	(1, 0, 1)
9	3	1	(3, 3, 2)	2	(1, 1, 0)
10	3	2	(0, 1, 3)	0	(0, 0, 1)
11	3	2	(0, 0, 0)	1	(0, 1, 0)
12	3	2	(3, 2, 0)	2	(1, 0, 0)
13	3	3	(2, 0, 2)	0	(0, 0, 0)

Figure 4.1 All grid entities types for one-, two- and three-dimensional grids

In the implementation, the basis is generated using template metaprogramming. The problem of generating bases is reduced to generating types, which represent the k-combinations. In addition to that, the implementation of the basis generation is dimension independent. The developer can generate `constexpr` basis value by using the static `getBasis<EntityOrientation>()` method in the `BasisGetter<EntityDimension>` class. However, the compiler must know the orientation and dimension at the compilation phase. Hence the grid additionally stores bases, which can be accessed dynamically by using `getBasis<EntityDimension>(orientation)` defined in the `Grid` class.

The user can access the basis of the entity from both the device and the host by using the `getBasis()` method of given `GridEntity`. In addition to that, the grid entity basis can be updated by using the `setBasis()` method.

4.1.4 Global index

The global index is the grid entity integer index calculated from the entity coordinate and the orientation. To calculate the index the grid entities counts are obtained from the basis and the dimensions of the grid following the method from Section 4.1.3. Then the index is calculated from the coordinate of the entity the following way.

$$\text{index} = c.x() + \dim.x() * c.y() + \dim.x() * \dim.y() * c.z() \tag{4.2}$$

The c is the coordinate of the grid entity and the dim is the grid entity dimensions.

For the entities with multiple orientations, this method will result in similar indices for different orientations. However, entities of different types are processed independently, therefore it is enough to correctly align the entities' indices. The index of the entity with the orientation k is adjusted by adding entity counts of lower orientations. The example is on the Figure 4.2, where (0), (1), (2) are orientations of the edges in the 3-dimensional grid.

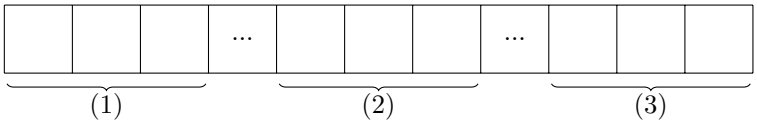


Figure 4.2 The edge index layout in two-dimensional grid

To access the global index of the entity the user can use `getIndex()` method.

4.1.5 Boundary entities

The grid entity is a boundary if all of its vertices are located on the surface of the grid. The user can check if the entity is the boundary by using the `isBoundary()` method of the `GridEntity`.

4.1.6 Neighbours

In most cases, numerical methods perform operations on the entity and its neighbours. The process of obtaining neighbour entities has two edge cases. The first case is when the neighbour of the same dimension and the orientation to the parent entity are needed. Then to get a neighbour it is enough to adjust the coordinate and update the global index because the entity of the same type has the same basis and orientation. The second case is when the dimension and the orientation differ from the parent entity. Then the coordinate and the global index are updated the same way, but in some cases, the neighbour entity doesn't support the parent orientation. In these cases, the orientation of the neighbour entity is set to the maximally supported orientation and the basis is updated.

To get the neighbour entity the user can use any of the following methods:

- `getNeighbourEntity<Dimension, Steps...>()` returns the entity of the `Dimension` with the coordinates adjusted by `Steps...`
- `getNeighbourEntity<Dimension, Orientation, Steps...>()` returns the entity of the `Dimension` and `Orientation` with the coordinates adjusted by `Steps...`
- `getNeighbourEntity<Dimension>(const Coordinate& offset)` returns the entity of the `Dimension` with the coordinates adjusted by `offset`.
- `getNeighbourEntity<Dimension, Orientation>(const Coordinate& offset)` returns the entity of the `Dimension` and `Orientation` with the coordinates adjusted by `offset`.

When the method is called with `Dimension` and `Orientation`, it will use the static basis value. Otherwise, it will access the basis dynamically from the grid. The same is true for the `Steps...` variadic parameter pack.

4.1.7 Measure and center

The grid entity implements `getMeasure()` and `getCenter()` methods to get the information about the entity position in the domain. The measure represents the volume of the hypercube, which the entity represents, and the center represents the position of the grid entity center in the domain.

In general, both the measure and the center can be implemented using bases, because the basis represents the directions in which the entity spans. However, the implementation will be less efficient, because the basis must be accessed dynamically.

4.2 Grid

The `Grid<Dimension, Real, Device, Index>` class template represents an orthogonal structured grid. It inherits from `NDimGrid<Dimension, Real, Device, Index>`, which implements dimension independent operations on the grid. The `Dimension` template parameter represents the dimension of the grid. The `Device` represents the device, on which parallel traverse of grid entities will be performed. The `Index` and `Real` represent index and real value correspondingly. The grid stores information about entity counts, configurations of domain and bases of entities.

4.2.1 Dimensions

The dimension of the grid along the axis is the number of edges along the corresponding axis. When the dimension of the grid is updated, the grid recalculates the count of every entity type in the grid and the proportions of the grid. The entity counts are obtained the same way described in Section 4.1.3.

The user can use `getDimensions()` to get the dimensions of the grid. The user can use `setDimensions()` to set the dimensions of the grid.

4.2.2 Domain

The implementation of the grid only supports cells of equal dimensions. The user can specify the dimension of the cell by using `setSpaceSteps()` method. The method accepts on the input the vector, where each real element at i-th index represent the dimension of the cell along the corresponding axis. It is often needed to calculate the coefficients in the following form.

$$h_x^i * h_y^j * h_z^k \tag{4.3}$$

The h_* represents the space step along the $*$-th axis and i, j, k are integer powers of space steps. For that reason, the grid calculates the coefficients for all i, j, k from range $[-2, 2]$. The user can get the coefficient by using `getSpaceStepsProducts(Powers...)` and `getSpaceStepsProducts<Powers...>()` method. By using the second method, the grid will calculate the index of the coefficient in coefficient lists at the compile time.

Additionally, the grid supports positioning in given space. The origin represents the point, where the zero coordinate is located. The type of origin is the `Point`. The user can use `getOrigin()` to get the origin of the grid. The user can use `setOrigin()` to set the origin of the grid. The `setOrigin()` accepts on the input the vector of real values.

Finally, the user can specify the size of the grid and the origin using `setDomain()` method. The grid will calculate the space steps based on their dimension and update the origin.

4.2.3 Traverse

The grid implements parallel traverse of all, boundary and interior entities. To traverse grid entities the developer must use `forAll`, `forBoundary` and `forInterior` methods. Because of the Cartesian coordinate system, the traversal of grid entities is the same as the iterating of the grid entities' coordinates. The problem of iterating all coordinates for entities with a single orientation type is solved with several nested for loops. It is because all entities are identified with the coordinates and all coordinates form rectangle. For entities with multiple orientations grid entities dimensions differ for each orientation, but it can be resolved by processing each orientation independently. In the following text, the *start index* term will represent the initial values of nested for loops and the *end index* will represent the end values of nested for loops.

The traverse of entities with multiple orientations is used in the traverse of all, boundary and interior entities. Therefore static (compile-time) for loop `ForEachOrientation` was implemented using template metaprogramming. The `exec()` function from `ForEachOrientation` accepts the lambda function on the input, which will be called with orientation and basis of entity. The orientation has `std::integral_constant` type, therefore it can be used in type definition. Additionally, the `ForEachOrientation` can skip one orientation value.

Finally, the implementation of all entities of any dimension traversal is the following.

```
1 Coordinate from = ..., to = ...;

2 auto exec = [&](const Index orientation, const Coordinate& basis) {
3   ParallelFor<...>::exec(from, to + basis, ...);
4 };

5 ForEachOrientation<Index, EntityDimension, Dimension>::exec(exec);
```

Code listing 1 The implementation of all grid entities traversal

The traversal of interior (non-boundary) entities is similar to the traversal of all entities, but the start and the end indices must be adjusted. For entities with multiple orientations, the start index is adjusted by adding the basis vector. By doing this the start index is moved along axes, such that the entity doesn't span along with them, therefore it isn't a boundary. For entities with a single orientation, the start and end indices are adjusted depending on if the coordinate represents the grid vertex or cell. The implementation of interior entity traversal is the following.

```
Coordinate from = ..., to = ...;

auto exec = [&](const Index orientation, const Coordinate& basis) {
  if (Dimension == EntityDimension) {
    ParallelFor<...>::exec(from + Coordinate(1), to - Coordinate(1), ...);
    return;
  }

  Templates::ParallelFor<...>::exec(from + basis, to, ...);
};

ForEachOrientation<Index, EntityDimension, Dimension>::exec(exec);
```

Code listing 2 The implementation of interior grid entities traversal

The traversal of boundary entities is built on the fact, that the surface of the n-dimensional hypercube is formed by the $(n-1)$-dimensional linear spaces. There are exactly $2n$ boundary surfaces of the hypercube. Each surface is spanned by the set of $(n-1)$ independent vectors and the other basis vector is orthogonal to the surface. Therefore, to iterate all surfaces it is enough to iterate all basis vectors or all orientations of grid surface dimension entity. The traversal of the intersections formed by boundaries is an important implementation aspect because intersections belong to multiple surfaces. Each entity must be traversed only once, therefore, entities, which belong to the intersections, must be addressed by the traversal algorithm. The implementation of boundaries traversal is the following.

Entities of the grid surface dimension form intersections of lower dimensions than the dimension of the entity. Therefore, each entity belongs only to one surface and can be traversed the following way.

```
auto exec = [&](const auto orientation, const Coordinate& basis) {
  constexpr int orthogonalOrientation = EntityDimension - orientation;

  forBoundary(orthogonalOrientation, orientation, basis);
};

ForEachOrientation<Index, EntityDimension, Dimension>::exec(exec);
```

Code listing 3 The traversal of boundary entities of the grid surface dimension

For all other entities, intersections are formed. In addition to this, there are two other cases. The first case is when the entity type has multiple orientations. Then all entity orientations belong to the surface except the orientation which is orthogonal to the surface. Hence, the orthogonal orientation must be omitted in the boundary entities traversal. The second case is when the entity type has a single orientation, then all entities belong to the surface. For both cases boundary intersection issue occurs, therefore the implementation remembers all traversed surfaces in the `isBoundaryTraversed` vector. The implementation is presented in the Code Listing 4.

```
auto exec = [&](const auto orthogonalOrientation) {
  auto exec = [&](const auto orientation, const Coordinate& basis) {
    forBoundary(orthogonalOrientation, orientation, basis);
  };

  ForEachOrientation<
    ..., isDirectedEntity ? orthogonalOrientation : -1
  >::exec(exec);

  isBoundaryTraversed[orthogonalOrientation] = 1;
};

DescendingFor<orientationsCount - 1>::exec(exec);
```

Code listing 4 The traversal of other boundary entities

In both cases, the implementation calls `forBoundary` lambda functions, which start traversal of bottom and upper surface along with orthogonal orientation. On lines 3-10 the program adjusts indices to not iterate intersection entities several times. On lines 12, 15-16 start and end indices are adjusted to traverse only surface entities.

```
Coordinate start = from;
Coordinate end = to + basis;

if (EntityDimension != Dimension - 1) {
  #pragma unroll
  for (Index i = 0; i < Dimension; i++) {
    int offset = (!isDirectedEntity || basis[i]) && isBoundaryTraversed[i];

    start[i] = offset;
    end[i] = end[i] - offset;
  }
}

start[orthogonalOrientation] = end[orthogonalOrientation] - 1;

Templates::ParallelFor<Dimension, Device, Index>::exec(start, end, ...);

// Skip entities defined only once
if (!start[orthogonalOrientation] && end[orthogonalOrientation]) return;

start[orthogonalOrientation] = 0;
end[orthogonalOrientation] = 1;

Templates::ParallelFor<Dimension, Device, Index>::exec(start, end, ...);
```

Code listing 5 The traversal of boundary entities along orthogonal axis

4.3 Unit testing

After the implementation, several problems with unit testing of the solution arose. Firstly, there are hundreds of edge cases for different grid dimensions and configurations. Secondly, the implementation must be tested on multiple device architectures. Thirdly, the solution actively uses template metaprogramming, therefore, multiple specializations of methods must be tested. To address all these issues several test environments were written to test the implementation of the grids. In general, the test environment must be additionally tested to ensure the validity of the test.

All tests were written using the modern C++ unit testing framework GTest [26]. GTest provides the solution to implement generic test cases by using `TYPED_TEST` macro. The whole documentation on the `TYPED_TEST` can be found in [26].

4.3.1 Test of grid properties

The grid templates provide multiple getter and setter methods to configure the grid. To test these methods, the test environment uses the following approach. It configures the grid and instantiates a `TNL::Array` container. The test environment traverses the container in parallel and on multiple devices. During the traversal, the test environment gets the value from the grid using the index in the container and stores it in the container. Finally, the test environment traverses the container once more, but the traversal is done sequentially and the values are verified.

4.3.2 Test of traversal and grid entities

To test traversal methods, the test environment uses the following approach. It configures the grid and instantiates entity storage. Then it calls the tested traverse method and stores all information about the entity in the entity storage. Finally, it traverses all grid entities once more, but the traversal is done sequentially and the entity values are verified. Additionally, the entities of the grid possess the following property: the set of all entities is formed by the union of interior and boundary entities. Using this property the traversal methods were additionally tested.

4.3.3 Test of neighbours

To test the neighbour entity access, the test environment uses the following approach. It configures the grid and instantiates entity storage. Then it calls the tested traverse method with the body, inside which the grid accesses the specific grid entity neighbour. The neighbour entity is stored in the entity storage. Finally, the test environment traverses all grid entities once more, but the traversal is done sequentially and the entity values are verified.

Chapter 5
Convolution

The implementation of the convolution operator was a preliminary to adding the algorithm to the TNL. Hence, it is designed as several benchmarks, which measure the speed of the convolution operator in multiple configurations. In addition to this, the implementation has a flexible design, which can be expanded with additional test cases and versions of the algorithm. According to the goals of the thesis, the convolution operator must be generic, therefore the following interface was chosen.

```
static void execute( const Vector< Index >& dimensions,
                     const Vector< Index >& kernelSize,
                     FetchData&& fetchData,
                     FetchBoundary&& fetchBoundary,
                     FetchKernel&& fetchKernel,
                     Convolve&& convolve,
                     Store&& store );
```

The `dimensions` and `kernelSize` parameters represent the size of the domain and the kernel accordingly. The `fetchData` and `fetchKernel` parameters are lambda functions, which must return the data and kernel element from index on the input. For boundary elements, the convolution operator will call `fetchBoundary` lambda function with the index of the boundary element. To define the convolution operation the user must specify `convolve` lambda function, which will be called with the accumulator, data element and the kernel. The lambda function must return the result of the operation. Finally, the `store` lambda function will be called with the index and the result of the convolution.

5.1 Basic implementation

The first step in the CUDA programming model is to define the thread hierarchy. In our implementation, each thread calculates the convolution for one data index. All threads are aligned in the thread blocks with the kernel dimensions and blocks are aligned in the grid with data dimensions. The main advantages of this implementation are the simplicity and the minimal number of idle threads on the data boundaries. The main disadvantage is that the size of the thread block is limited with the 1024 threads, therefore the volume of the kernel is limited to 1024 elements. Additionally, the block size is not aligned with the warp size, which may reduce performance. However, the alignment to warp size may significantly increase the number of boundary entities reads and significantly reduce performance.

The implementation of the one-dimensional convolution is the following.

```
Index ix = threadIdx.x + blockIdx.x * blockDim.x;
Real result = 0;

for( Index i = -kernelRadius; i <= kernelRadius; i++ ) {
  Index elementIndex = i + ix, kernelIndex = i + kernelRadius;

  result = convolve( result,
    elementIndex < 0 || elementIndex >= endX ?
      fetchBoundary( elementIndex ) : fetchData( elementIndex ),
    fetchKernel( kernelIndex ) );
 }

store( ix, result );
```

Code listing 6 The basic implementation of one-dimensional discrete convolution

There are two significant problems with such implementation. Firstly, each thread reads the kernel, data and boundary from the global memory. However, each thread uses the same kernel and some of them share data and boundary elements. The second problem is the ternary if-clause inside for loop, which causes branch divergence. The GPU will process each branch serially, which will significantly reduce performance.

5.2 Shared Memory

My implementation uses shared memory to address the problem of repeated reads from the global memory. The shared memory is allocated dynamically as a one-dimensional memory block, which every thread can access.

5.2.1 Kernel in the shared memory

The first optimization is to store the kernel in the shared memory, because it is the same for every thread. The kernel will always fit in the shared memory because its volume is limited to 1024 elements.

The implementation must be updated in the following way to support shared memory. At the beginning of the thread block execution, each thread reads the value of the kernel at thread index and stores the value in the shared block. Then it waits for all other threads read in the thread block. Finally, instead of the `fetchKernel`, the convolution will access elements from the shared memory.

```
Real* shared = TNL::Cuda::getSharedMemory< Real >();

// The size of the block is equal to the kernel size
shared[ threadIdx.x ] = fetchKernel( threadIdx.x );

__syncthreads();
```

Code listing 7 The implementation of the kernel load to the shared memory for one-dimensional discrete convolution

5.2.2 Data in the shared memory

The other optimization is to store the data block needed for convolution in the shared memory. The main reason is that it is significantly larger than the kernel, therefore fewer reads from the global memory are needed. The size σ of the data block in the shared memory can be calculated in the following way.

$$\sigma = \prod_{i=1}^{n}(2k_i - 1) \tag{5.1}$$

where n is the dimension of the kernel and k_i is the size of the kernel along i-th dimension. It is important to understand if the data fits in the shared memory. For the purpose of calculations, the size σ of the data block can be approximated in the following way.

$$\sigma = \prod_{i=1}^{n}(2k_i - 1) \leq \prod_{i=1}^{n} 2k_i \leq 2^n * 1024 \tag{5.2}$$

where n is kernel dimension. Using the observation from the equation (5.2), maximal sizes of the data blocks are listed in Table 5.1.

Dimension	Type	Elements[1]	Size[2]	Fit[3]
1	float	2048	8192	yes
1	double	2048	16384	yes
2	float	4096	16384	yes
2	double	4096	32768	yes
3	float	8192	32768	yes
3	double	8192	65536	no[4]

Table 5.1 The table of required shared memory size for storing data block

Based on the observations the data block will fit in the shared memory in most cases. To load data block into shared memory each thread splits data into 2^n tiles, where the n is the dimension of the convolution. The example of the tiling is presented in Figure 5.1. Each thread loads one value for each group and stores it in the shared memory.

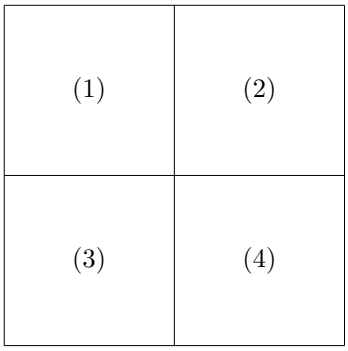


Figure 5.1 Thread loads of two-dimensional convolution

[1] Maximal number of elements in data block
[2] Maximal size in bytes
[3] The value is calculated by assuming the size of shared memory is 32 KB
[4] The data block will fit, if L1 cache is disabled

Each data block has the size of the kernel, therefore the data is stored by adjusting the index by the dimensions of the block.

```
1 Index ix = threadIdx.x + blockIdx.x * blockDim.x;
2 Index x = ix - kernelWidth / 2;

3 shared[ threadIdx.x ] = x < 0 || x >= endX ? fetchBoundary(x) : fetchData(x);

4 x = ix + kernelWidth / 2;

5 shared[ threadIdx.x + blockDim.x ] = x < 0 || x >= endX ?
6   fetchBoundary(x) : fetchData(x);

7 __syncthreads();
8 if( ix >= endX ) return;
```

Code listing 8 The implementation of the data load to the shared memory for one-dimensional discrete convolution

Storing data in the shared memory solved the problem with the branch divergence. Additionally, the for loop from Code Listing 6, line 3 can be unrolled with the pragma directive.

```
1 #pragma unroll
2 for( Index i = 0; i < kernelWidth; i++ ) {
3   Index elementIndex = i + threadIdx.x;
4   result = convolve( result, data[ elementIndex ], fetchKernel(i) );
5 }
```

Code listing 9 The implementation of one-dimensional discrete convolution with data in the shared memory

5.2.3 Kernel and data in the shared memory

Finally, both the kernel and the data block can be stored in the shared memory. However, in some cases the GPU must be additionally configured to prefer shared memory over L1 cache. Then the maximal size of the shared memory block is 48 KB. By adding the kernel elements the following table is obtained.

Dimension	Type	Elements[1]	Size[2]	Fit[3]
1	float	3072	12288	yes
1	double	3072	24576	yes
2	float	5120	20480	yes
2	double	5120	40960	yes
3	float	9216	36864	yes
3	double	9216	73728	no

Table 5.2 The table of required shared memory size for storing data block and kernel

[1] Maximal number of elements in data block
[2] Maximal size in bytes
[3] The value is calculated by assuming the size of shared memory is 48 KB

Chapter 6
Evaluation results

6.1 Environment

The evaluation and measurement of implementations were performed on the `gp1` machine provided by the Faculty of Nuclear Sciences and Physical Engineering, CTU Prague. The specification of the machine and the environment is the following.

Element	Specification
OS	Arch Linux 5.17.4-arch1-1
CPU	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
GPU	Nvidia Quadro P6000
NVCC	11.6.112
GCC	11.2.0

Table 6.1 The specification of test environment

The code was compiled with the `nvcc` and `gcc` using the TNL build system. The `./build benchmark` command was used to build benchmarks and `./build tests` was used to build unit tests.

6.2 Grid implementation

6.2.1 Refactoring results

Using the techniques described in Section 4 the size of the codebase was significantly reduced. The original implementation was 7949 lines long. In comparison to it, the new implementation is only 3716 lines long. In addition to this, the new implementation has everything prepared to make the grid orientation static. By doing this, some grid entity operations and statements can be evaluated at the compilation phase, which will result in better performance. In addition to this, the basis and the orientation can be evaluated statically, which will reduce the grid entity memory footprint. However, to add this the limitation of the CUDA lambda expressions must be overcome. Finally, the new implementation is tested on multiple devices and with multiple specializations of grid templates.

6.2.2 Comparison of the heat equation numerical solvers

One of the goals was to evaluate the overhead added by the use of grid classes. To evaluate it the following benchmarks were implemented.

1. The heat equation numerical solver using the `ParallelFor` algorithm in the TNL is the baseline benchmark. It uses only essential primitives, therefore it is the fastest of all benchmarks.
2. The heat equation numerical solver, which uses `Grid` classes, is the benchmark, which is used to measure the overhead added by using grids.
3. The heat equation numerical solver with the emulated grid entity is the benchmark, which implements the same class hierarchy as the implementation of grids in the TNL. The main reason for adding this benchmark was to evaluate if grid class hierarchy adds runtime overhead to the implementation.
4. The last numerical solver implements dimension independent grid prototype. The numerical solver traverses entity global indices and then calculates the coordinates of the entity in the grid, which is opposite to what the TNL grids do. The main reason for adding this benchmark was to test if the described approach is suitable for grid improvement.

All of these benchmarks were compiled with the NVCC and were evaluated on the host and the device in multiple grid configurations. The results were obtained the following way. Each benchmark calculated the solution of the heat equation 10 times. For the host, the benchmark executed 1000 iterations of the numerical scheme and for the device, the benchmark performed 10000 iterations of the numerical scheme. The results are listed in Table 6.2.

			Grid width x Grid height			
ID[1]			**100x100**	**400x400**	**1600x1600**	**3200x3200**
1. Parallel for	host	time[2]	0.047	0.134	1.448	-
		difference[3]	-	-	-	-
	device	time	0.169	0.267	3.410	13.175
		difference	-	-	-	-
2. TNL	host	time	0.070	0.437	6.662	-
		difference	33.3	69.4	78.2	-
	device	time	0.194	0.294	3.440	13.207
		difference	12.9	9.1	0.9	0.2
3. Emul. ent.	host	time	0.044	0.137	1.448	-
		difference	-6.9	2.8	0.1	-
	device	time	0.172	0.274	3.417	13.181
		difference	2.4	2.7	0.2	0.05
4. N-dim.	host	time	0.047	0.221	3.375	-
		difference	1.4	39.3	57.1	-
	device	time	0.402	0.490	5.190	15.281
		difference	58	46	34.3	13.7

Table 6.2 The table of the heat equation numerical solvers evaluation results. All benchmarks were compiled with the NVCC and the double-precision operations were used for calculations

[1] The ID of the benchmark in order of description on the top of the page
[2] Execution time measured in seconds
[3] Difference to the ”parallel for” benchmark in percents

It can be seen that the use of TNL grids doesn't add significant overhead for large grids, which were executed on the GPU. However, for the small grid and host execution, the overhead is significant.

The main reason for significant overhead measured for small grids is the execution time of numerical scheme iteration. For small grids, the iteration of the numerical scheme is significantly shorter than for large grids. Hence, the ratio of the overhead added by the grid to the iteration execution time is bigger than the equivalent ratio for large grids.

For host execution, the significant overhead is the result of the compiler's inability to perform lambda expression inlining. The problem was described in Section 2.2. However, the issue only occurs, when the numerical solver is compiled with the NVCC. To evaluate the numerical solvers on the host the additional measurements were performed for benchmarks compiled with GCC. On the host the numerical solvers traverse grid entities in parallel by using the OpenMP library. The results are listed in Table 6.2.

		Grid width x Grid height			
ID[1]		**100x100**	**400x400**	**800x800**	**1600x1600**
1. Parallel for	time [2]	1.330	2.970	4.701	22.838
2. TNL	time	1.340	2.691	4.883	22.532
	difference [3]	0.7	-10.3	3.7	-1.3
3. Emul. ent.	time	1.317	3.283	5.065	23.205
	difference	-0.9	9.5	7.1	1.5
4. N-dim.	time	1.404	5.790	20.936	86.710
	difference	5.2	48	77	73

Table 6.3 The table of the heat equation numerical solvers evaluation results. All benchmarks were compiled with the GCC and the double-precision operations were used for calculations

By results evaluation, it can be seen, that the use of grid classes doesn't add significant overhead when the numerical solver was compiled with the GCC.

6.3 Convolution

6.3.1 Comparison of the convolution operators

All versions of the convolution operator described in Section 5 were evaluated on different kernel and domain sizes. To measure the computation time the convolution operator is calculated 10 times, and then the average of all times is taken. The obtained results showed, that one-, two- and three-dimensional convolution operators behave similarly with the increasing kernel volume. Therefore, only the measurements for the two-dimensional convolution are presented in Figure 6.1. The basic implementation is predictably the slowest one. By storing the data block in the shared memory, insignificant performance improvement is achieved for small kernels. However, for large kernel sizes almost 2 times performance improvement is achieved. In comparison to this, by storing the kernel in the shared memory the opposite behaviour is obtained. For small kernel sizes, the implementation is significantly faster, however, with the increasing kernel size the performance degrades. Finally, the implementation with storing both data and kernel in the shared memory is the fastest. The main reason is that after loading data no reads from the global memory are needed.

[1] The ID of the benchmark in order of description on the top of the previous page
[2] CPU time measured in seconds
[3] Difference to the "parallel for" benchmark in percents

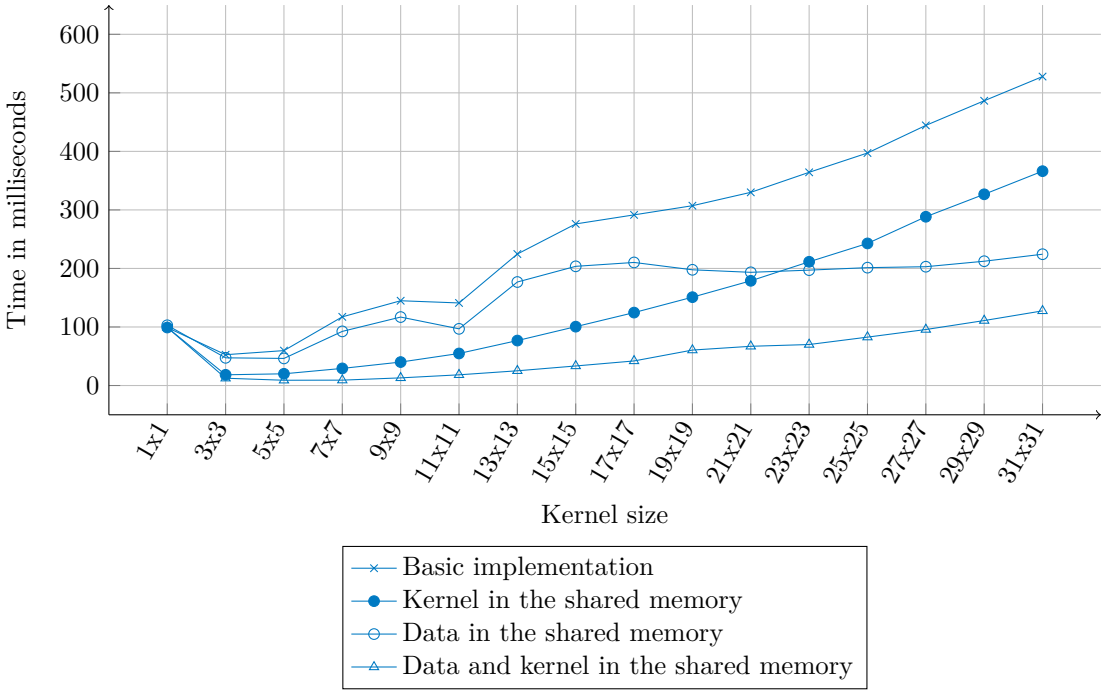


Figure 6.1 The evaluation times of different convolution operator implementations for different kernel sizes on the 8192x8192 domain

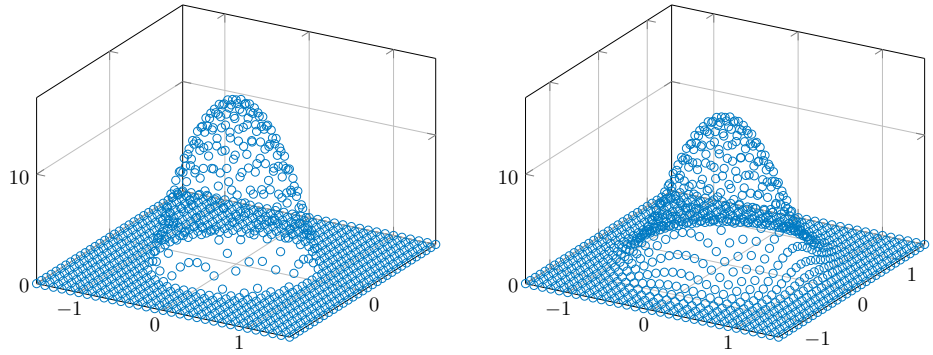
The convolution operator was compared to the implementation proposed in the following article [27]. The authors of the article propose the implementation of the convolution operator and compare it to different public versions of it. Their implementation is similarly limited to the thread block size. However, their implementation has one significant drawback. The performance of the convolution operator significantly degrades with the 15x15 and larger kernel sizes. In comparison to them, the implementation proposed in the thesis doesn't have such drawbacks. And for small kernels, the implementation has a similar performance shown in the article.

6.3.2 Comparison between the FDM approximation of the heat equation and the pseudo-analytical solution

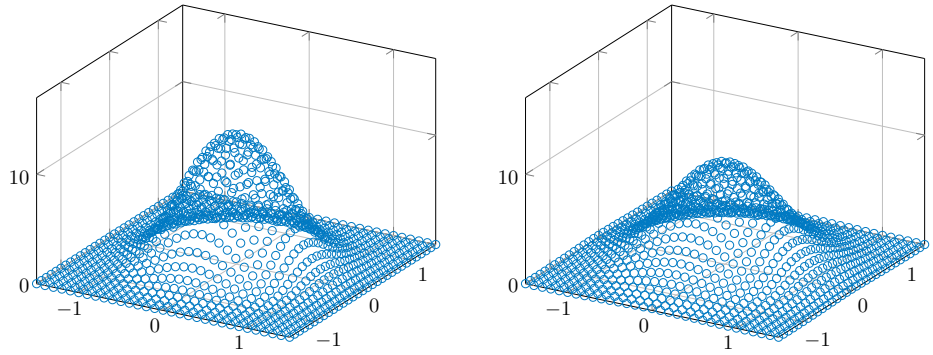
The numerical scheme obtained in Section 3.5 iteratively approximates the solution of the heat equation. In comparison, the pseudo-analytical solution calculates the exact state of the heat equation at a specific time. The pseudo-analytical solution can't be computed in the continuous domain, therefore the implementation numerically approximates it. The following initial condition μ was chosen to compare the pseudo-analytical solution of the heat equation and the numerical approximation obtained with the FDM.

$$\mu = \frac{x^2}{\alpha} + \frac{y^2}{\beta} + \gamma \tag{6.1}$$

where α, β, γ are user-defined variables. To evaluate the results the values -0.05, -0.05 and 15 were chosen for α, β and γ accordingly. The results of the FDM approximation are illustrated in Figure 6.2 and the results of the pseudo-analytical solution approximation are illustrated in Figure 6.3. By observing the results it can be seen, that both the approximation obtained with FDM and the approximation of pseudo-analytical solution behave similarly to each other.

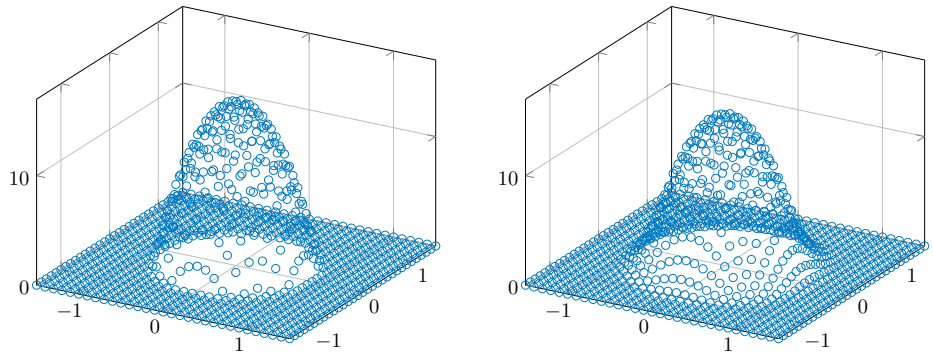


a. t=0 **b.** t=0.02

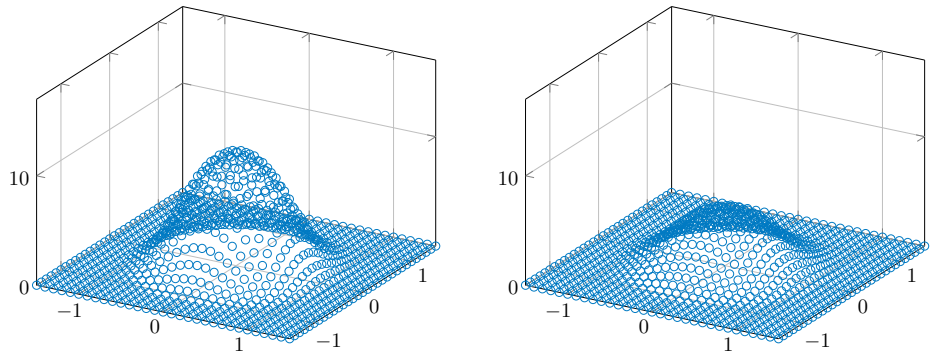


c. t=0.04 **d.** t=0.08

Figure 6.2 The heat equation FDM numerical approximation at different times



a. t=0 **b.** t=0.02



c. t=0.04 **d.** t=0.08

Figure 6.3 The heat equation pseudo-analytical solution at different times

6.4 Discussion

There are several improvements, which can be added to the grid implementation. Firstly, there must be a solution, which will make it possible to add the orientation to the entity type and wouldn't require the generic lambda function. It will allow to remove the basis from the entity and compute it statically. One of the possible ways to achieve it is to use a type erasure pattern [28] for the grid entity. Secondly, the implementation of boundary entity traversal can be improved with CUDA streams. To traverse the boundaries of the grid a large number of small CUDA kernels should be executed. Each of them wouldn't use all resources of the GPU, which will add significant overhead by scheduling and waiting for their execution. These kernels can be launched concurrently in multiple CUDA streams, which will improve the performance of the traversal. Finally, the heat equation numerical solver doesn't use all methods of grid and grid entity. Therefore there is a possibility of hidden performance issues in some of them. However, these can be only revealed by using a grid in various numerical solvers.

The implementation of the convolution operator can be improved by considering other algorithms. One of such algorithms is the convolution operator, which is based on the Fast Fourier transform. It is the algorithm of lower asymptotic time complexity, therefore it can calculate the convolution faster on larger domains. The other way to improve the convolution operator implementation is to add support for large kernels.

Conclusion

The main goal of the thesis was to improve the implementation of grids in the TNL library. With the help of the combinatorics features of the grid data structure and modern C++ language features I significantly reduced the size of the grid codebase. In addition to this, the implementation aims to be dimension-agnostic, which leaves open the possibility of implementing grids of higher dimensions. I met with the obstacles of lambda expression implementation in the CUDA programming model, which blocked the desire to make the grid entity as lightweight as possible. However, C++ 17 standards introduce generic lambdas and I expect that the future versions of CUDA will add support to this feature. As a result, the compiler will generate all operations of the grid entity during the compilation phase, which will significantly reduce the runtime overhead added by the grid class.

Then I covered the implementation with unit tests. I tried to make them as dimension-agnostic as possible, to follow the design principles of the grid. I made them scalable, i.e. it is easy to add different launch configurations and add specific edge cases. One of the main operations is the grid traversal. I tested it on small and large grids and to reinforce the confidence even more I tested the properties of the traversal operations.

After proper testing, I investigated the overhead added by the grid implementation. I implemented simple heat equation solvers and benchmarks to measure it. As a result, I found out that the grid overhead wasn't significant in most cases. In addition to that, I understood the nature of the overhead and explained it. In the end, I updated the design of benchmarks implementation to make it simple to expand them with any implementation.

My compulsory goal was to learn CUDA programming. I fulfilled it during the implementation of the convolution operator. I tried using different techniques to make it as efficient as possible. I compared the most efficient implementation with the others and found out that it was outperforming them. In the end, I implemented benchmarks following the same design principles as the heat equation benchmarks. Thus, other implementations of the convolution operator can be added to evaluate the performance.

Finally, I compared the convolution operator with the Gaussian kernel and the pseudo-analytical solution to the heat equation. I received similar results, which shows the correctness of the pseudo-analytical solution.

Bibliography

1. HENNESSY, John L.; PATTERSON, David A. *Computer architecture: a quantitative approach*. 5th ed. Amsterdam: Elsevier, c2011. ISBN 012383872x.
2. TANENBAUM, Andrew S.; BOS, Herbert. *Modern operating systems*. Fourth edition. Boston: Pearson, [2014]. ISBN 1292061421.
3. NVIDIA CORPORATION. *Nvidia Ampere GA102 GPU architecture* [online] [visited on 2022-04-23]. Available from: `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`.
4. ADVANCED MICRO DEVICES, Inc. *AMD Ryzen™ Threadripper™ 3990X specifications* [online] [visited on 2022-04-23]. Available from: `https://www.amd.com/en/product/9111`.
5. *A history of NVIDIA stream multiprocessor* [online] [visited on 2022-04-23]. Available from: `https://fabiensanglard.net/cuda/`.
6. *CUDA C++ programming guide* [online] [visited on 2022-04-23]. Available from: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
7. FLYNN, Michael. Flynn's Taxonomy. In: *Encyclopedia of Parallel Computing* [online]. Boston, MA: Springer US, 2011, pp. 689–697 [visited on 2022-04-23]. ISBN 978-0-387-09765-7. Available from DOI: `10.1007/978-0-387-09766-4_2`.
8. HARRIS, Mark. *Using Shared Memory in CUDA C/C++* [online] [visited on 2022-04-23]. Available from: `https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/`.
9. NVIDIA CORPORATION. *CUDA toolkit documentation* [online] [visited on 2022-04-23]. Available from: `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html`.
10. NVIDIA CORPORATION. *Profiler User's Guide* [online] [visited on 2022-05-07]. Available from: `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`.
11. OBERHUBER, Tomáš; KLINKOVSKÝ, Jakub; FUČÍK, Radek. *Template Numerical Library* [online] [visited on 2022-04-24]. Available from: `https://tnl-project.org/`.
12. *Arrays tutorial* [online] [visited on 2022-04-23]. Available from: `https://mmg-gitlab.fjfi.cvut.cz/doc/tnl/md_Tutorials_Arrays_tutorial_Arrays.html`.
13. *Vector tutorial* [online] [visited on 2022-04-23]. Available from: `https://mmg-gitlab.fjfi.cvut.cz/doc/tnl/md_Tutorials_Vectors_tutorial_Vectors.html`.
14. *Lambda expressions* [online] [visited on 2022-04-23]. Available from: `https://en.cppreference.com/w/cpp/language/lambda`.

15. GROVER, Vinod; MARATHE, Jaydeep. *New Compiler Features in CUDA 8* [online] [visited on 2022-04-23]. Available from: https://developer.nvidia.com/blog/new-compiler-features-cuda-8/.
16. *For loop tutorial* [online] [visited on 2022-04-23]. Available from: https://mmg-gitlab.fjfi.cvut.cz/doc/tnl/md_Tutorials_ForLoops_tutorial_ForLoops.html.
17. *Flexible (parallel) reduction and prefix-sum tutorial* [online] [visited on 2022-04-23]. Available from: https://mmg-gitlab.fjfi.cvut.cz/doc/tnl/md_Tutorials_ReductionAndScan_tutorial_ReductionAndScan.html.
18. *Derivative* [online] [visited on 2022-04-23]. Available from: https://encyclopediaofmath.org/wiki/Derivative.
19. *Partial derivative* [online] [visited on 2022-04-23]. Available from: https://encyclopediaofmath.org/wiki/Partial_derivative.
20. ZILL, Dennis G. *A First Course in Differential Equations with Modeling Applications*. 10th. Boston: Cengage Learning, 2013. ISBN 1285401107.
21. ROYCHOWDHURY, D. G. *Computational Fluid Dynamics for Incompressible Flows* [online]. CRC Press, 2020-8-20 [visited on 2022-04-23]. ISBN 9780367809171. Available from DOI: 10.1201/9780367809171.
22. *Taylor formula* [online] [visited on 2022-04-23]. Available from: https://encyclopediaofmath.org/wiki/Taylor_formula.
23. *Convolution of functions* [online] [visited on 2022-04-23]. Available from: https://encyclopediaofmath.org/wiki/Convolution_of_functions.
24. *Rectangle rule* [online] [visited on 2022-04-23]. Available from: http://encyclopediaofmath.org/index.php?title=Rectangle_rule.
25. EVANS, Lawrence. *Partial Differential Equations*. 2nd ed. Providence, Rhode Island: American Mathematical Society, 2010-3-2. ISBN 9780821849743. Available from DOI: 10.1090/gsm/019.
26. GOOGLE LLC. *GoogleTest* [online] [visited on 2022-05-04]. Available from: https://github.com/google/googletest.
27. CHEN, Peng; WAHIB, Mohamed; TAKIZAWA, Shinichiro; MATSUOKA, Satoshi. Pushing the liits for 2D convolution computation on CUDA-enabled GPUs. *IPSJ SIG Technical Report*. 2018, no. 22.
28. O'DWYER, Arthur. *What is Type Erasure?* [Online] [visited on 2022-05-09]. Available from: https://quuxplusone.github.io/blog/2019/03/18/what-is-type-erasure/.

Contents of enclosed medium

- `README.md` ..... The description of media contents
- `src`
  - `impl` ..... The TNL repository versions
    - `convolution.zip` ..... The version with the convolution benchmarks
    - `grid.zip` ..... The version with grid improvements and heat equation benchmarks
  - `vis` ..... The visualization of the results
    - `requirements.txt` ..... The environment specification for the Jupyter notebook
    - `Evaluation.ipynb` ..... The Jupyter notebook with the visualizations
    - `data` ..... The data for visualizations
  - `thesis` ..... The source code of the text in LaTeX
- `text` ..... The text of the thesis
  - `thesis.pdf` ..... The text of the thesis in PDF format