TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DIPLOMARBEIT

The Finite Element Method On Massively Parallel Computing Architectures

Ausgeführt am

Institut für Mikroelektronik
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Erasmus Langer
und
Dr.techn. Karl Rupp, MSc

durch

Florian Rudolf

Pamessergasse 7, 2103 Langenzersdorf

Studienkennzahl 869
Matrikelnummer 0326156

Datum

Unterschrift

Abstract

The finite element method is a widespread tool for solving boundary value problems approximately. These problems often arise in physics, for example in electrodynamics and mechanics. To get accurate approximations to the true solution, fine discretizations are needed leading to large systems of linear equations. To accelerate the solution of these systems, the algorithms are parallelized by the use of multiprocessors. In particular, graphics adapters are employed for parallel processing and are investigated for their potential within the finite element method.

This work makes use of the Open Computing Language, in short OpenCL, which is a platform independent framework for programming multiprocessor computing architectures, including an application programming interface and a programming language definition. Graphics adapters are programmed by using the OpenCL framework. Different types of sparse matrix storage schemes, which can be used for finite element solving, are presented. Matrix-vector multiplications of these implementations are benchmarked. Theoretical algorithm and memory requirement complexities are discussed and implementations using OpenCL are presented. ViennaCL, a C++ Basic Linear Algebra Subprograms (BLAS) implementation, which makes heavy use of OpenCL, is presented. A mathematical introduction to boundary value problems and the finite element method is given. Finite element implementations are proposed and benchmarked. The benchmark results are presented and discussed.

Contents

List of Figures

List of Tables

Listings

Acknowledgment

Writing this diploma thesis was challenging at times and I learned a lot in the process, thanks in great part to all the people helping me and giving me key insights in times of need. Now it is time to pay credit where credit is due and say thank you.

First of all, I would like to thank Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Erasmus Langer and the Institut für Mikroelektronik of the Vienna University of Technology for giving me the opportunity to write this diploma thesis, the necessary hardware support, technical information and all required resources.

I want to express my sincere gratitude to my advisor Dr.techn. Karl Rupp, MSc for supporting me and my work from the very beginning. Thank you a lot for your advice, personal and technical guidance, motivation, patience, forbearance (especially with the English language), proofreading and much more! This work wouldn't be as it is without your support and assistance. Thank you a lot!

I'd also like to thank my classmates from university, especially Dipl.-Ing. Andreas Morhammer and Dipl.-Ing. Klaus Neuwirth, with whom I spent lots of lectures and courses. Thank you for your support during study and for this diploma thesis.

Last but not least I want to thank all the people close to me for giving me motivation and stability, support and guidance for my life and my study. Thank you for believing in me, Rafaela, Maria, Hansi, Alex and Mäx!

Chapter 1

Introduction

Boundary value problems often arise in physics, for example in electrodynamics and mechanics. A popular tool for a solving these problems is the finite element method, which is a numerical algorithm. As most numerical methods, approximations of the exact solution of the boundary value problem get more accurate, the finer the discretization is. Fine discretizations are needed leading to large systems of linear equations. The solving algorithm for a linear system of equations is parallelized to take advantage of the use of multiprocessors, especially graphics processors. Graphics processors are evaluated for their performance in the finite element method.

Chapter 2 gives an introduction to the Open Computing Language, in short OpenCL. OpenCL is a platform independent framework for programming multiprocessor computing architectures, including an application programming interface (API) and a programming language definition. Especially graphics adapters can be programmed by using the OpenCL framework.

Chapter 3 introduces different types of sparse matrix storage schemes which can be used for finite element solving. Theoretical algorithm and memory requirement complexities are discussed and implementations using OpenCL are presented. The last section of this chapter covers benchmark results of different implementations of sparse matrix-vector multiplication.

In Chapter 4 the C++ library ViennaCL is presented. ViennaCL is a Basic Linear Algebra Subprograms (BLAS) implementation which makes heavy use of OpenCL. Some extra features like iterative solver for systems of linear equations, preconditioners and fast Fourier transformation are supported by ViennaCL.

The finite element method is described in Chapter 5. Boundary value problems and their weak formulation are defined in this chapter. A mathematical background for this chapter is given in the Appendix A. Based on this definition, the finite element method algorithm is derived. A short example using the Poisson equation is presented in the last section of this chapter.

Chapter 6 combines the finite element method with the power of graphics adapters. A boundary value problem is defined and different implementations of the finite element method for solving this problem using the CPU, ViennaCL and OpenCL are described and benchmarked. The results are presented and discussed in the last section of this chapter.

Chapter 2

OpenCL

In this chapter the Open Computing Language framework, in short OpenCL, is presented [6]. The history of OpenCL is presented in Section 2.1. Section 2.2 presents the OpenCL framework and its model layout.

OpenCL is a platform and programming language independent framework specification for parallel programming on heterogeneous collections of processors. The framework is designed for SPMD (single program, multiple data) algorithms. The specification was initially written by Apple Inc. and consists of two major parts: the OpenCL computing language, which is a programming language similar to C, and the programming API for common programming languages like C or C++. OpenCL is based on a client-server architecture where the host is the application running on the central processing unit (CPU) and the client is the computing device, for example a graphics adapter.

2.1 History of OpenCL

There are two primary ways of increasing the power of a computing architecture: increasing the clock frequency or adding more processors. Because of technical and physical limitations the first option became much more difficult in the past years. Therefore, CPUs increased their performance by adding more cores and making them more flexible instead of primarily increasing the speed. Beside common CPUs, another processor type became very powerful: the graphics processing unit, in short GPU. Initially only used for graphics processing, graphics processors got much more flexible due to the introduction of programmable shaders in 2000. Since then, graphics adapters have evolved very fast, actually faster than predicted

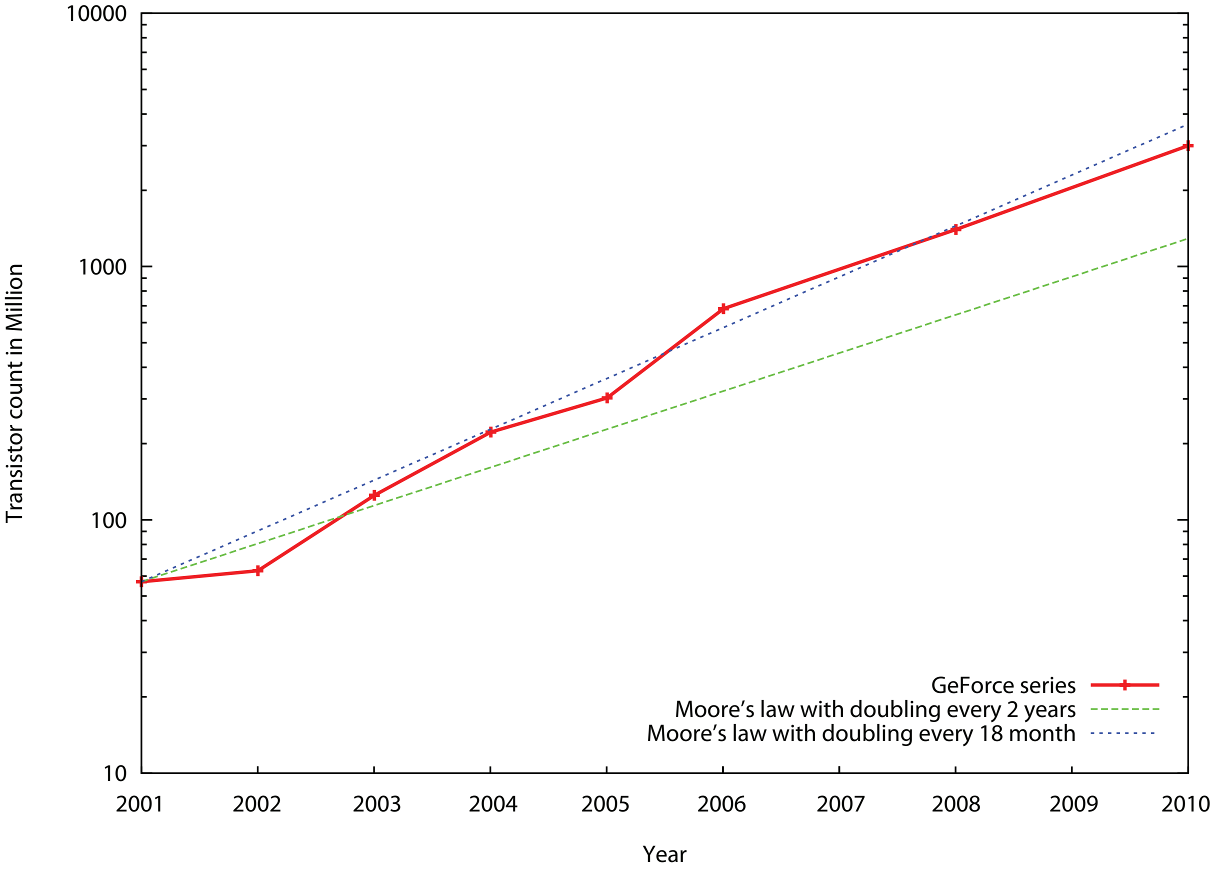


Figure 2.1: Moore's law compared to NVIDIA GeForce graphic processor

by Moore's original law [1] [2] [3]. Figure 2.1 shows the transistor growth of NVIDIAs graphic processors from 2001 to 2010.

With increasing flexibility and performance, graphics processors have started to be used for general purpose computations. First implementations used the graphics API DirectX or the Open Graphics Library (OpenGL) and shader for getting hardware independent access to the GPU [4]. In 2007 NVIDIA published the first version of CUDA (Compute Unified Device Architecture), a platform independent software development kit which allows developer to directly use the graphics processors functionality. Later in 2007 ATI released their development kit which could be used to access ATI graphics hardware. Only a year later, the first specification of OpenCL was published. Apple released the first OpenCL implementation in August 2009 with their operation system Mac OS X 10.6. In the same year, Microsoft introduced DirectCompute with an API similar to that of OpenCL, as a part of DirectX. NVIDIA and AMD/ATI both offer, besides their own development kits,

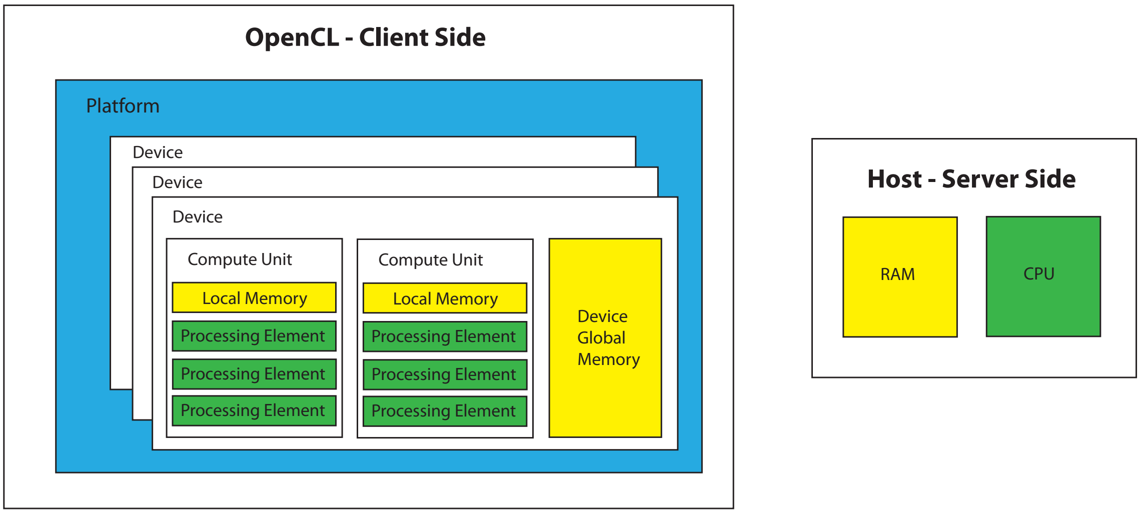


Figure 2.2: The OpenCL platform model

implementations for OpenCL and DirectCompute. OpenCL is now maintained and advanced by the industry consortium Khronos.

2.2 The OpenCL Framework

According to the OpenCL specification, the OpenCL framework can be split into three main parts:

1. Platform model
2. Execution model
3. Memory model

These three models are discussed in the following.

2.2.1 Platform Model

The OpenCL platform model is shown hierarchically in Figure 2.2. On top there is the application using the OpenCL implementation, called the host. The host is connected to the OpenCL platform, which is a set of OpenCL devices. An OpenCL device is a collection of so-called compute units, which again consists of processing elements. A processing element is a simple processor, executing scalar operations. Many processing elements in combination with a local memory form a compute unit. The actual OpenCL computations are carried out in the processing elements, where a stream of instructions is executed as SIMD units (single instruction, multiple data) or SPMD units (single program, multiple data). For example, an OpenCL device can be a multi-processor computer, in which case a compute unit is a CPU die and the processing elements are the cores of this CPU.

Asynchronous tasks are passed to the OpenCL implementation through commands. The host application submits commands to an OpenCL context. A context is a visibility region including one or more devices from one platform. Within such a context, command queues can be created, which manage execution on the OpenCL implementation.

2.2.2 Execution Model

An OpenCL program consists of two execution units: the host program, written in a standard programming language and executed on the host, and the OpenCL kernels, written in the OpenCL/C programming language and executed on one or more OpenCL devices. In the host program the OpenCL API is used to create the OpenCL objects, especially the context, and to manage them. Execution of OpenCL kernels are initiated by the host program.

The most important part in the execution model are the OpenCL kernels. An OpenCL kernel is a small executable which is run in parallel on a OpenCL device. This program is mostly written in the OpenCL/C programming language, but native kernels can also be used. Native kernels, however, are highly hardware-dependent. Every instance of such a kernel, called work-item, executes the same source code and is identified by a global unique ID. Depending on the source code and the global ID, the program flow may vary in work-items. Multiple work-items are organized in a work-group. Every work-group has a unique group ID and within a work-group every work-item has a unique local ID. Hence, every work-item can be uniquely identified by either its global ID or a combination of its local ID within the current work-group and the work-group ID. A work-item is executed on a processing element, while a work-group corresponds to a compute unit. A kernel is able to synchronize data within a work-group.

The previously presented IDs are available as scalars, two- or three-dimensional tuples. With this attribute it is possible to easily write OpenCL kernels operating on an image. For example, a two-dimensional ID is handy for a blur filter kernel on an image. From this point of view, a work-item can be seen as a point in an index space. This index space in OpenCL

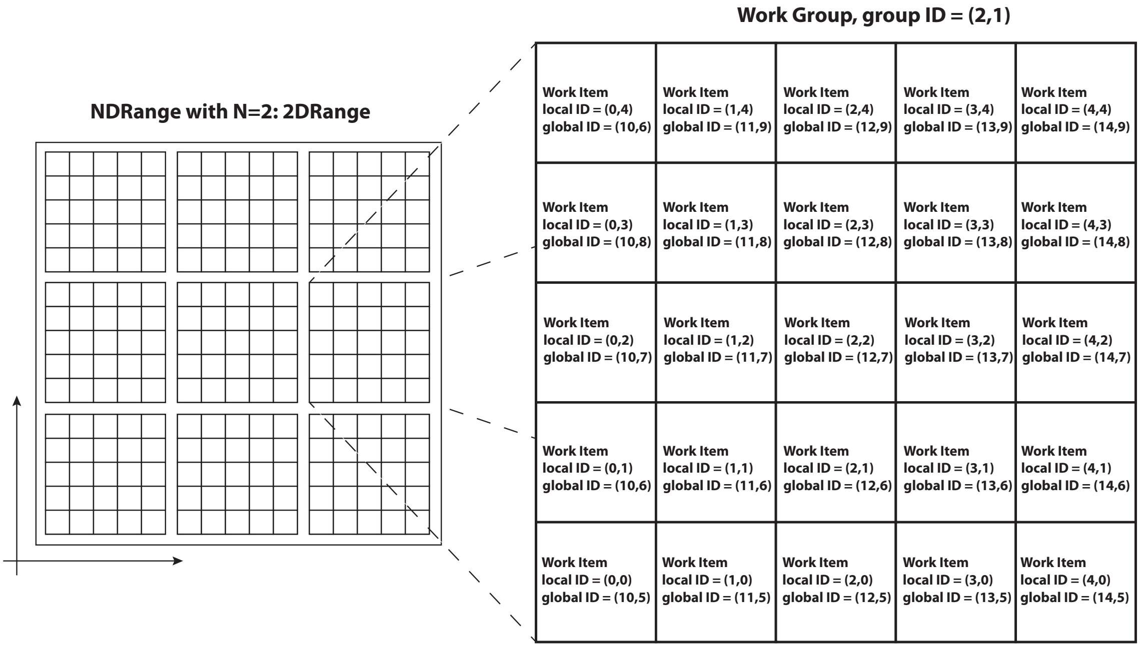


Figure 2.3: A 2D index space

is called NDRange, where N refers to the dimension (1, 2 or 3). Figure 2.3 shows an example of a two-dimensional index space.

As previously mentioned, command queues are used to manage the kernel execution in OpenCL. A command queue is associated with a single device and coordinates the execution of OpenCL commands processed by this OpenCL device. The host program appends these commands to a command queue and the OpenCL library takes care of that. Execution within a command queue can be in-order or out-of-order. In the first case the commands are executed one after another in the order pushed to the queue. In the second case the commands are also launched in the order pushed to the queue, but the device may not wait for the commands to finish before launching the next command. It is possible to have more than one command queue on a device, but these queues run completely independent of each other and synchronization has to be done manually by using OpenCL events.

OpenCL specifies the following set of commands:

Kernel Execution Commands. These commands will execute an OpenCL kernel

Memory Commands. These commands will perform memory manipulation tasks, like copying memory from or to OpenCL memory objects, transferring memory between them, or mapping OpenCL memory objects into host memory.

Synchronization Commands These commands perform synchronizations on the command queue.

Kernel Execution Commands and Memory Commands generate event objects when they are put into a command queue. These event objects can be used to synchronize commands in this queue and manage execution from the host program.

2.2.3 Memory Model

OpenCL distinguishes four different types of memory accessible to kernels. The main OpenCL device memory is termed global memory. Global memory is visible to all work-items and can be accessed for read and write operations. Depending on the device, read and write operations may be cached. Constant memory is also a part of the global device memory, but this region is constant to the work-item. The host application is responsible for filling this memory with data. With constant memory the OpenCL device may be able to perform optimizations within the kernel. Local memory is a memory block which is local within one work-group. Variables allocated in this memory section are shared within a work-group. The last memory type is private memory, which is a memory accessible to one single work-item. No other work-items are able to access the local memory of another work-item. Figure 2.4 illustrates the memory model for work-groups and work-items.

The individual memory types also differ in access times. The following information is valid for NVIDIA processors of the Fermi architecture. Private memory is used for temporary variables in a kernel and commonly resides in the registers of the processor. In this case there is no latency for accessing this memory. When accessing local memory, however, there are roughly 4 to 6 clock cycles of memory latency [7]. Global memory generates even more memory latency. A memory access in global memory has about 400 to 600 clock cycles of latency [5]. However, the scheduler is able to hide these latencies in most cases by switching

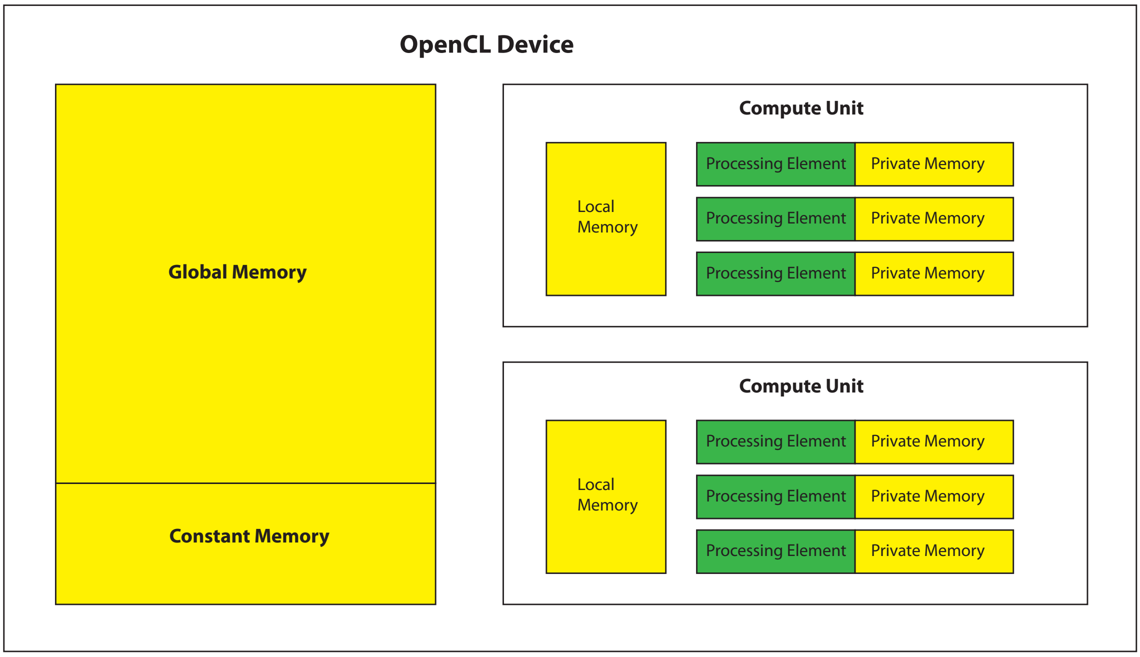


Figure 2.4: The OpenCL memory model

blocked threads with some other threads.

Memory is managed by memory objects. These objects are created by the host program and represent regions in the global memory of the device. There are two ways of transferring data between the OpenCL device and the host program: Either by using transfer commands, or by mapping device memory into host memory. For transferring data, a data copy command is added to a command queue. Three ways of data transfer are supported: copying data from the host program to a device memory object, copying from one device memory object to another device memory object, and copying data from a device memory object back to the host program. Copy operations can be blocking, which means that the host programs flow is stopped until the transfer is completed and the memory is ready for further use. Mapping device memory into host memory also needs do be accomplished via a command queue. Instead of actually transferring the data, the content of the memory is available in host memory where the host program has direct access to. When the host program has completed all manipulations of the mapped memory, it has to be unmapped so that OpenCL is able to synchronize the mapping memory region in host memory to the device memory. The mapping and unmapping commands can also be used in blocking or non-blocking manner.

Two types of memory objects are specified: buffer objects and image objects. While buffer objects define a sequential memory array, image objects are two- or three-dimensional pictures. Every element of a buffer object array can be any type specified by OpenCL: a basic type, a vector type or a struct. Elements are stored sequentially in device memory which allows simple offset access using pointer arithmetics. Image objects support predefined image types only and may not be stored sequentially. Access to image data from within an OpenCL kernel is only possible using lookup functions. The use of pointers or typical array arithmetic is not allowed.

OpenCL buffer memory objects are restricted in use compared to host random access memory (RAM). There is no clear way to use pointers in OpenCL kernels because it is not possible to reference one element in one memory object by an absolute pointer from within another memory object. Additionally, the number of arguments passed from the host program to the OpenCL kernel is limited. Therefore, it is also not possible to use a list of arguments, especially a list of memory objects, for a kernel. Within a kernel the memory size and location from a memory object is constant. It is not possible to allocate, free or resize device memory within a kernel.

This leads to some basic design limitations when implementing algorithms using OpenCL. Dynamic data structures are difficult to implement and very restricted in use. The only way to use dynamic data structures in OpenCL device memory is to allocate a big memory object and use the offset to the beginning of the block for access. It is therefore theoretically possible to write a memory management within this memory block. Such a custom memory management has some major disadvantages. First of all, it has to be thread-safe which can only be achieved by synchronizing all threads. This is very impractical and usually leads to poor performance.

Chapter 3

Sparse Matrix-Vector Multiplication

Linear mappings are a central topic in mathematics. In this work, only linear mappings from finite dimensional vector spaces to itself are relevant. Given a basis, such a linear mapping can be uniquely identified with a matrix-vector multiplication, making matrix-vector multiplication one of the most important applications of linear algebra. Linear equation systems, a special case of the inverse linear mapping, are used very often as well. There are mainly two algorithm types for solving linear equation systems: direct solvers and iterative solvers. Direct solvers for matrices with a lot of zero entries do not perform well. Under certain circumstances, iterative solver compensate these problems, especially for large sparse matrices. For an iterative solver the matrix-vector multiplication is a central operation in each cycle. Linear systems arising from the discretization of partial differential operations are usually sparse, which means that most of the entries are equal to zero. In such cases it is more efficient to store non-zero elements only such that less memory is required and the matrix-vector multiplication requires less operations. On the other hand, the matrix-vector multiplication algorithm has to be adapted for the storage scheme. These changes will lead to an overhead when storing the matrix and performing the matrix-vector multiplication. This overhead should be minimized.

n_r	number of rows of the matrix
n_c	number of columns of the matrix
n_z	number of non-zero elements of the matrix
n_{zr}	maximum number of non-zero elements per row

Table 3.1: Shortcuts for sparse matrix-vector multiplication algorithm analysis

In this chapter the shortcuts in Table 3.1 will be used. The memory requirement complexity of a dense matrix is $O(n_r \times n_c)$ and accessing specific elements is done in constant time. The runtime complexity of the matrix-vector multiplication algorithm for dense matrices is $O(n_r \times n_c)$.

3.1 Sparse Matrix Layouts

As described in Section 2.2.3, OpenCL buffer memory objects are very restricted in use. Because of this, all memory layouts of data structures and algorithms should be adapted to a style with no dynamic data structures and as less memory indirections as possible. It is recommended to design the data structures in a way that is suitable for the algorithm. Especially it is advantageous to pack topologically close data together. When focusing on the matrix-vector multiplication for example, the elements of a vector next to each other should be located next to each other within the memory as well. Depending on the hardware, caching might further improve the performance in these cases.
In this second two matrix storage schemes are presented: the coordinate scheme and the compressed scheme. There are much more storage schemes than these two, which are also suitable for OpenCL usage but are not covered in this work [22].

3.1.1 Coordinate Scheme

The simplest way to store a sparse matrix is by a sequence of triples: a row index, a column index and the (non-zero) value. This storing scheme is called coordinate scheme. The non-zero values may be sorted by their coordinates within the matrix. As described in the introduction, these triples are packed into linear arrays and therefore no dynamic data structures is used. Figure 3.1 shows the data scheme of the coordinate matrix.

The memory required for storing a coordinate matrix is linear in the number of non-zero elements. Accessing a specific entry of the matrix is quite slow. If an element is requested, the non-zero value array has to be searched for that element. If the elements in the array are sorted, this search has a logarithmic complexity, otherwise the complexity is linear. In both cases there is additional work to do if there is a write access to an element not yet present within the array. In case of an unsorted array the element can simply be pushed to the back of the array. In contrast, the insertion of a new element in a sorted array requires that all successive elements are moved. Apart from memory management, the complexity

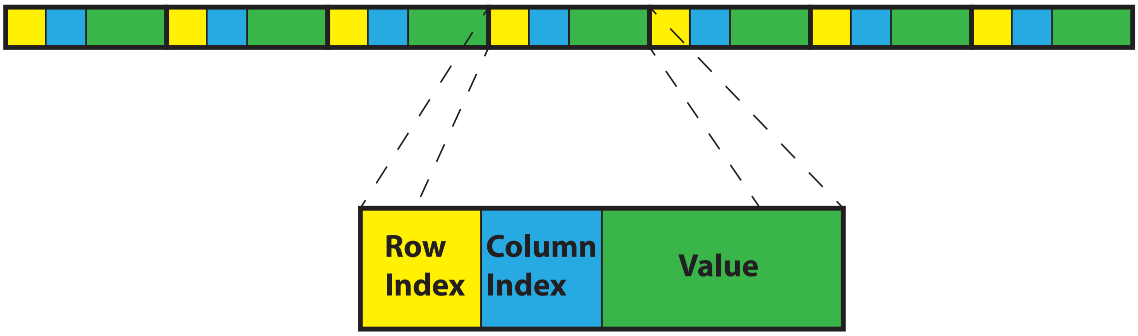


Figure 3.1: Storage scheme of the coordinate matrix

	Sorted Array	Unsorted Array
Read Only Access	$O(\log(n_z))$	$O(n_z)$
Write Access	$O(n_z)$	$O(1)$

Table 3.2: Element access complexity for coordinate matrices

of an insertion is linear for sorted arrays and constant for unsorted arrays. Table 3.2 gives an overview of access complexities.

Iterating over all non-zero elements reduces to an iteration over an array. All non-zero elements store their own location within the matrix, so no additional effort is required for iterating over the entries, no matter if the array is sorted or not. Depending on the matrix non-zero structure there might be a caching advantage for some algorithms if the array is sorted. For example, the matrix-vector multiplication algorithm may benefit from a sorted array because it is likely that neighbor elements of the vector and the result vector are accessed. For parallelization, the sorting of the array is important for good performance. If the array is unsorted, different threads might access the same entry in the result vector and race conditions appear.

The matrix-vector multiplication algorithm is quite simple. The pseudo code for that algorithm is presented in Listing 3.1. The complexity of this algorithm is linear in the number of non-zero values. There is no difference if the array is sorted or not, but a sorted array might have caching advantages. If write access to the result vector within every loop step is not atomic, it is difficult to parallelize the matrix-vector algorithm of the coordinate matrix.

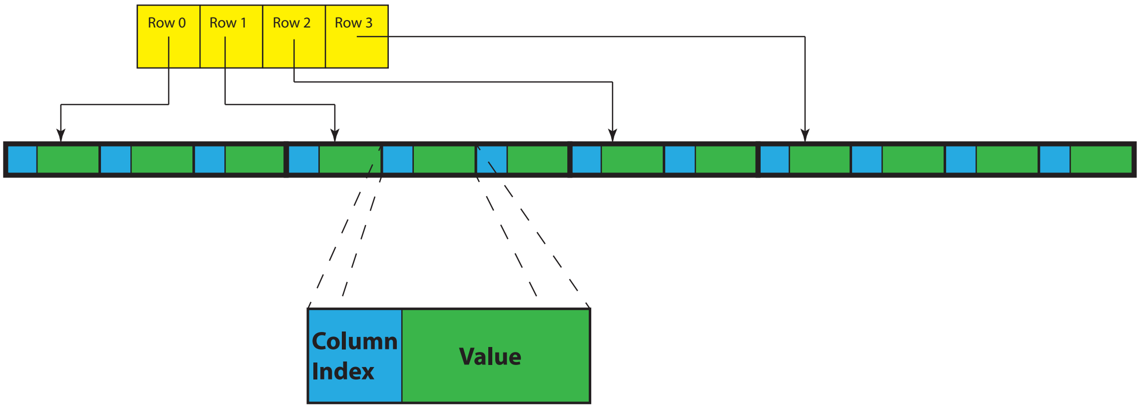


Figure 3.2: Storing scheme of the compressed matrix

Listing 3.1: Matrix-vector multiplication for coordinate matrices

```
set all entries in result_vector to zero

for each non-zero value NZ in matrix
    result_vector [ NZ.row_index ] += NZ.value * vector[ NZ.column_index ]
end
```

3.1.2 Compressed Scheme

In the compressed memory scheme, the data is stored in two arrays. All non-zero elements and their column index are stored ordered by their row index in an array. Additionally there is a second array, called row jumper, which holds the start index of every row in the first array. The memory layout is presented in Figure 3.2. It is also common that the column indices and the values are split into two separate arrays instead of one interlaced array. The values within a row may also be sorted by their column index, similar to the coordinate scheme.

Compared to the coordinate scheme, the compressed scheme usually requires less memory to store. The memory complexity is $O(n_z + n_r)$. The number of non-zero elements is greater than the number of rows for all practical sparse matrices, so the memory consumption is linear in the number of non-zero values.

The complexity of accessing an element of the matrix depends on the sorting of the array. Due the fact that the every row has an entry in the row jumper array leading to the start index of the column indices and values, the complexity of retrieving the row is constant. Within a row the runtime depends on sorting of the elements by the column index. If the

	Sorted Row	Unsorted Row
Read Only Access	$O(\log(n_{zr}))$	$O(n_{zr})$
Write Access	$O(n_z)$	$O(n_z)$

Table 3.3: Element access complexity for compressed matrices

elements are sorted, the complexity of a read-only access is logarithmic in the number of non-zero values in that row, otherwise it is linear. Write access time does not depend on whether the values within a row are sorted or not. This complexity is always linear in the number of non-zero values because a memory move is always required when inserting a new element. An overview of the access complexity is presented in Table 3.3.

A pseudo code of the matrix-vector multiplication algorithm for the compressed scheme is presented in Listing 3.2. The algorithm is the same no matter if the values within a row are sorted or not. Due to the fact that every iteration of the inner loop operates on one non-zero element, the algorithm has a complexity of $O(n_z + n_r)$. As described above, the number of rows is often less than the number of non-zero values, so in most cases the complexity is linear in the number of non-zero values. Since every iteration of the outer loop is independent, this algorithm can be parallelised easily.

Listing 3.2: Matrix-vector multiplication for compressed matrices

```
for each row R in matrix
    tmp = 0

    for each element E int row R
        tmp += E.value * vector[ E.column_index ]
    end

    result[ R.row_index ] = tmp
end
```

OpenCL has built-in support for fixed size vector types (sizes of 2, 3, 4, 8 and 16 are supported) including memory functions and arithmetics. Memory access is usually more efficient when reading little memory blocks instead of single elements. There is a minor adaption to the compressed scheme which benefits from these facts. This modification allocates the number of elements within a row to a multiple of an alignment number. If the number of non-zero values in a row is already a multiple of that alignment, no changes have to be made. Otherwise, additional zeros are added in the row. This modification requires slightly more memory in most cases, but the matrix-vector algorithm can be adapted in a way where more than one element is read in each iteration. Instead of iterating over every single element within a row in the inner loop, multiple non-zero values are read in every step depending on the alignment. Good numbers for the alignment for most graphics hardware are the sizes of the built-in vector types, especially 4 or 8.

3.2 Implementation with OpenCL

OpenCL object management is done via the OpenCL API. Since creating a context, command queues, memory objects and so on is straight-forward, these technical details are not covered in this work.

3.2.1 Memory object creation and transfer

Memory objects can only be created by the host via OpenCL API functions. Within an OpenCL kernel there is no way to create a memory object. Those objects are constant in structure for the whole lifetime, which means that the size of an memory object can not be changed after creation. As stated in the Section 3.1, dynamic data structures are difficult to implement and very restricted in their use.

OpenCL provides four methods for managing data: copy data from host memory to device memory, copy data from device memory back to host memory, copy data from one memory object on a device to another memory object of probably another device and fill data of a memory object. All methods enqueue a memory transfer job to a command queue. The memory transfer takes place as soon as the command queue launches the job. There are special versions of the memory copy commands which add support for transferring regions. For example it is possible to transfer a sub-matrix block of a dense matrix stored in one memory object in just one call using rectangular memory transfer functions. But those functions only support regular rectangular regions and it is not possible to manage random accesses from the host with only one OpenCL function call. An access to three random elements in a memory buffer requires at least two memory transfer function calls: one striding call for accessing two elements and one call for the third access. The memory creation also supports copying data from host to the created memory object, but this function does not support OpenCL events and therefore can not be synchronized.

Transferring the matrix storage schemes from Section 3.1 to OpenCL leads to some major complications. Memory structures can not be changed from within an OpenCL kernel, the kernel is restricted to read-only accesses to the compressed or coordinate matrix. Iterating over all elements on the other hand is possible for both matrix types presented previously. Therefore, the matrix-vector multiplication algorithm can be ported to OpenCL easily. Accessing elements from the host is possible, but may result in heavy memory management operations. With the finite element method the matrices would be set up once and not modified after creation. In those cases the matrix is created by the host and then transferred to OpenCL, where only the matrix-vector multiplication is performed.

3.2.2 The OpenCL kernel

In Section 3.1.2 the compressed matrix was presented. The focus of this section is on an implementation of this matrix-vector multiplication using OpenCL. To perform a task on an OpenCL device a kernel has to be created. The algorithm has to be ported to the OpenCL C programming language and this source code is passed to OpenCL as a string. There are also possibilities for using native kernels, but those are highly hardware dependent. The OpenCL implementation compiles and links the kernel source code on the fly. This ensures that a task written for OpenCL can be used on every OpenCL implementation.

The kernel source code for the matrix-vector multiplication is presented in Listing 3.3. The algorithm specified in Listing 3.2 is split up in a way that every thread processes a number of rows in the matrix. In multi-threaded programming on the CPU an application uses only a few threads, mostly one thread per core. OpenCL was made for architectures with many more cores than a common CPU.

The *__kernel* keyword specifies that this function can be used as a kernel. A kernel function has no return value and can be called by the host using the OpenCL API. Every kernel consists of a set of parameters provided by OpenCL. The matrix-vector multiplication kernel has five arguments. Memory buffer arrays are represented via pointers. An array argument must have an address space qualifier so that the OpenCL kernel knows which type of memory to expect. In this case the matrix arrays, the vector and the result vector are specified as *__global*, which means that the arrays are memory blocks in the global OpenCL device memory and suitable for read and write access. All pointers, except the pointer to the result vector, are marked *const* because the kernels do not require write accesses to these arrays.

Listing 3.3: OpenCL kernel source code for the matrix-vector multiplication of a compressed matrix

```
__kernel void packed_compressed_matrix_vector_multiplication(
    __global const unsigned int * row_jumper,
    __global const unsigned int * column_indices,
    __global const float * element_buffer,
    __global const float * vector,
    __global float * result_vector,
    unsigned int num_rows)
{
    for (int row = get_global_id(0); row < num_rows; row += get_global_size(0))
    {
        unsigned int idx = row_jumper[row];
        const unsigned int stop_idx = row_jumper[row+1];
        float tmp = 0.0f;
        for (; idx < stop_idx; ++idx)
        {
            tmp += element_buffer[idx] * vector[ column_indices[idx] ];
        }
        result_vector[row] = tmp;
    }
}
```

The kernel source code represents one thread that will operate on a number of rows. Each kernel instance is uniquely identified by its global ID which will be used to identify the start row index on which the kernel operates. The total number of threads is given by the function *get_global_size*. The outer loops iterates over all rows, where each thread may processes several rows. Inside the outer loop, the first and last index in the column index and element buffer array for the current row are looked up in the row jumper array. Then the entry of the result vector is calculated by using a temporary variable. This temporary variable is not required, but saves some expensive array accesses to the result vector in global memory.

As presented above, there is a modification to the compressed storage scheme which uses an alignment. Listing 3.4 presents the source code of the alignment version of the compressed matrix matrix-vector multiplication with an alignment of 4. It is assumed that the entries of the row jumper array are the number of quadruples within one row instead of the number of elements. The OpenCL C programming language supports additional built-in vector types. The algorithm iterates over all quadruples of non-zero elements within a row and loads the values into a temporary vector. Those values are used to calculate the local dot product of the matrix-vector multiplication algorithm.

Listing 3.4: OpenCL kernel source code for matrix-vector multiplication of a compressed matrix using an alignment of 4

```
__kernel void packed_compressed_matrix_vector_multiplication(
    __global const unsigned int * row_jumper,
    __global const unsigned int * column_indices,
    __global const float * element_buffer,
    __global const float * vector,
    __global float * result_vector,
    unsigned int num_rows)
{
    for (int row = get_global_id(0); row < num_rows; row += get_global_size(0))
    {
        unsigned int idx = row_jumper[row];
        const unsigned int stop_idx = row_jumper[row+1];
        float tmp = 0.0f;
        for (; idx < stop_idx; ++idx)
        {
            float4 vals = vload4(idx, element_buffer);
            uint4 col_idx = vload4(idx, column_indices);
            tmp += vals.x * vector[col_idx.x];
            tmp += vals.y * vector[col_idx.y];
            tmp += vals.z * vector[col_idx.z];
            tmp += vals.w * vector[col_idx.w];
        }
        result_vector[row] = tmp;
    }
}
```

3.2.3 Launching the kernel

After the kernel has been successfully passed to OpenCL and the compilation of the source code is completed, the kernel is ready to use. Before launching the kernel on an OpenCL device, the kernel parameters have to be set to associate the memory objects with the kernel parameters. Then, the kernel can be launched by using an NDRange (see Section 2.2.2). For this NDRange the size and number of a work-group has to be specified. As stated in Section 2.2.2 a work-group operates on a compute unit and a work-item operates on a single processing element. The OpenCL function for launching a NDRange is able to determine a proper work-group size automatically, but in many cases it is preferable to specify the size and number of work-groups by hand. For example the device has 16 compute units and 250 threads should be started. Threads should be split equally on up to 16 compute units. The best possible choice for that is to launch 25 threads on 10 compute units, in which case 6 compute units are idle, while 10 compute units execute 25 threads each. In this case it is better to launch 256 threads with 6 threads idle, but 16 threads are launched on each compute unit with 16 threads each. This results in much better load balance. To achieve this flexibility, the kernel source code has to handle situations where extra idle threads are launched. This method provides a flexible way to choose the number of work-groups as well as the work-group size, which is very helpful for optimizing kernels. In the case of the kernel sources presented in Listing 3.3 and Listing 3.4, no modifications have to be made.

Finding ideal values for the work-group size depends on the hardware, the data, and on the kernel. In general there are a few criteria for this value. The number of idle threads should be kept to a minimum. The number of work-groups should be greater or equal to the number of compute units. This ensures that all compute units are used for calculation. The work-group size should be greater or equal to the number of processing elements per compute unit, otherwise there are processing elements which execute no kernel instance. For most graphics adapters the processing elements within one compute unit execute the same kernel [5]. All other processing elements are idle. So in this case the work-group size should be a multiple of the number of processing elements. If there are more work-items than processing elements, the compute units scheduler might be able to balance the work load and compensate memory latencies.

3.3 Matrix-Vector Multiplication Benchmarks

In this section the performance of the matrix-vector multiplication on different hardware is discussed. The benchmarks are carried out using randomly generated quadratic sparse matrices with sizes from $4^4 = 128$ to $4^{10} = 1048576$ and one additional size of $2^{21} = 2097152$. The same matrix is used for benchmarking different matrix types on different hardware for fair comparison. The number of non-zero values per row is constant 32 for all matrix sizes. Each benchmark consist of an average over 20 matrix-vector multiplications and is are carried on a workstation with Linux 3.1.6. The benchmark programs are written in C++ and compiled with gcc 4.4.5 using the '-O3' optimization. OpenMP (Open Multi-Processing) is used to parallelise the CPU implementation. The IEEE 754 single precision float type is used as a data type for the matrix values.

The target processor hardware is an Intel Core i7-960 CPU. For benchmarking with OpenCL a NVIDIA GeForce GTX 470 graphics adapter and a NVIDIA Tesla C2050 is used. Table 3.4 gives an overview of the hardware used for the benchmarks. For benchmarking

	Number of cores	Theoretical processing power using single precision float in GFLOPs	Theoretical memory bandwidth in GB/s
Intel Core i7-960 CPU [11] [12]	4	102.4	76.8
NVIDIA GeForce GTX 470 graphics card [9] [10]	448	1088.64	133.9
NVIDIA Tesla C2050 [8]	448	1030	144

Table 3.4: Benchmark hardware

	Memory throughput in bytes per matrix-vector multiplication for single floating point values	Processing power in operations per matrix-vector multiplication
Dense matrix	$(n_r \times n_c \times 2 + n_c) \times 4$	$n_c \times n_r \times 2$
Coordinate matrix	$n_{nz} \times 5 \times 4$	$n_{nz} \times 2$
Compressed matrix	$(n_{nz} \times 3 + n_r \times 3) \times 4$	$n_{nz} \times 2$

Table 3.5: Complexity for memory throughput and processing power

the OpenCL implementations, a work-group size of 8 is used, because it results in the best performance for those implementations.

The benchmark results presented in Table 3.6 and depicted in Figure 3.3 clearly show that the complexity of the dense matrix-vector multiplication is quadratic and the complexity of the coordinate and compressed matrix-vector multiplication is linear in the matrix size. The benchmarks with OpenMP and OpenCL are not representative for lower matrix sizes because of the OpenCL API or OpenMP thread overhead. The compressed scheme with an alignment of four, benchmarked on the GeForce 470, shows best performance over all benchmarks. Based on the values in Table 3.6, memory throughput and computing power can be estimated. Estimated complexities for memory throughput and processing power are presented in Table 3.5. For the largest matrix size in the benchmark, the actual memory throughput is about 8GB/s on the GeForce 470, which is about 6% of the theoretical power. This poor value can be explained with the randomness of the indices in the sparse matrix and confirms that peak bandwith can only be obtained with regular accesses.

Table 3.7 presents benchmark results with different local and global work sizes. A matrix with 262144 rows and columns containing 8388608 non-zero elements is used to perform a matrix-vector multiplication using a compressed matrix scheme with alignment of four. Only the NVIDIA GeForce GTX 470 was tested and a local work size of 16 has the best performance. A benchmark with different number of non-zero elements per row is presented

Matrix size	256	1024	4096	16384	65536	262144	1048576	2097152
Number of non-zero values	8192	32768	131072	524288	2097152	8388608	33554432	67108864
Dense matrix on Intel Core i7-960	1.65e-04	0.001295	0.020432	0.32622				
Dense matrix with OpenMP on Intel Core i7-960	5.47e-05	0.000322	0.005129	0.08200				
Coordinate matrix on Intel Core i7-960	4.38e-05	0.000078	0.000438	0.00150	0.0065	0.0327	0.180	0.59
Compressed matrix on Intel Core i7-960	2.81e-05	0.000110	0.000380	0.00104	0.0054	0.0282	0.146	0.58
Compressed matrix with OpenMP on Intel Core i7-960	2.00e-05	0.000050	0.000093	0.00022	0.0016	0.0095	0.052	0.14
Dense matrix on NVIDIA GeForce GTX 470	7.02e-05	0.000289	0.004905	0.08218				
Dense matrix on NVIDIA Tesla C2050	7.56e-05	0.000418	0.005272	0.08594				
Compressed matrix on NVIDIA GeForce GTX 470	2.13e-05	0.000033	0.000159	0.00078	0.0035	0.0154	0.067	0.13
Compressed matrix on NVIDIA Tesla C2050	2.34e-05	0.000033	0.000153	0.00072	0.0031	0.0141	0.072	0.15
Compressed matrix with alignment of 4 on NVIDIA GeForce GTX 470	1.79e-05	0.000027	0.000122	0.00062	0.0028	0.0123	0.052	0.10
Compressed matrix with alignment of 4 on NVIDIA Tesla C2050	2.02e-05	0.000028	0.000121	0.00058	0.0025	0.0113	0.059	0.13

Table 3.6: Benchmark result in seconds per matrix-vector multiplication

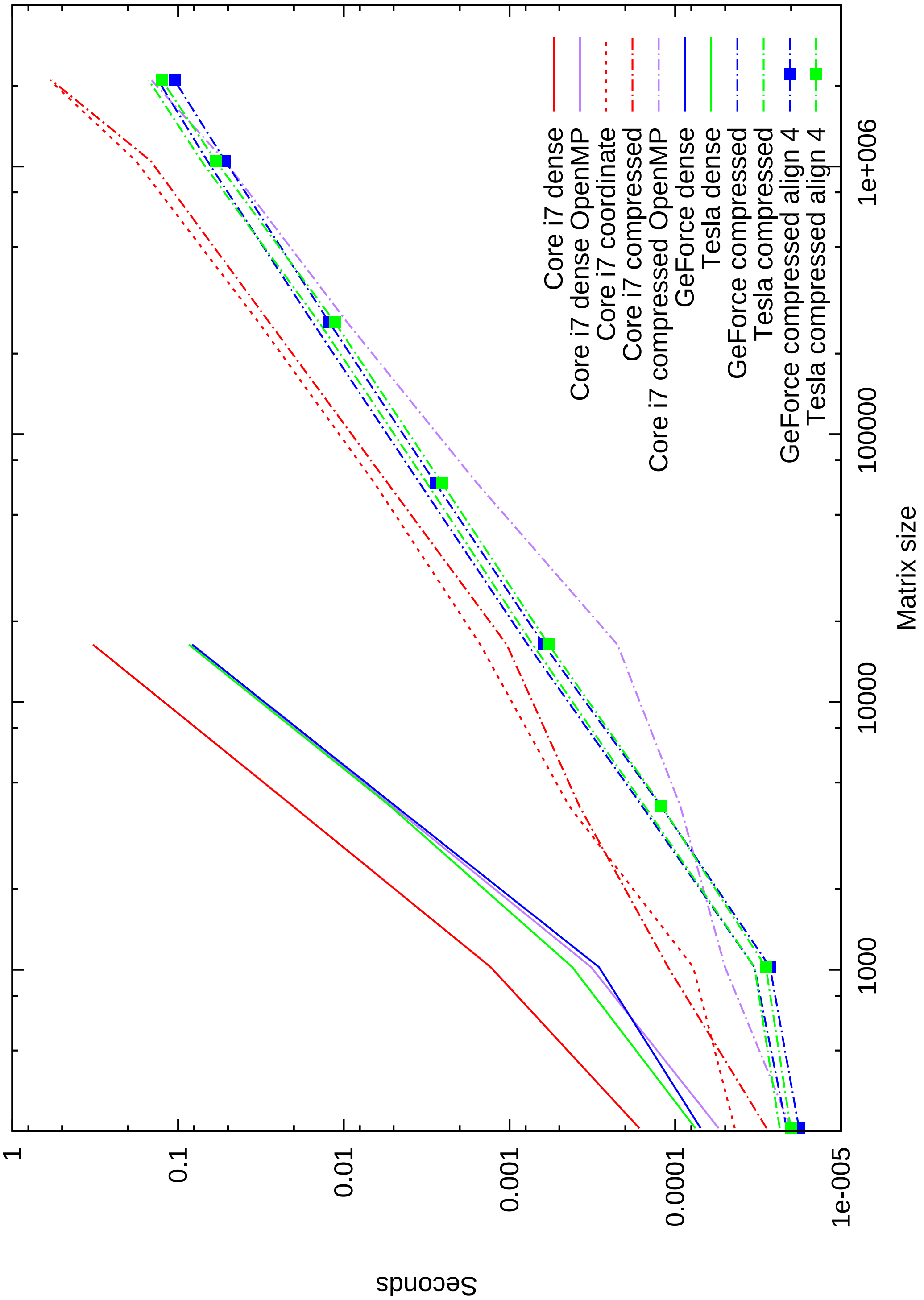


Figure 3.3: Execution times

		local work size			
		16	32	64	128
global work size	32	0.0121	0.0132	0.0152	0.0172
	64	0.0115	0.0134	0.0154	0.0154
	128	0.0131	0.0146	0.0161	0.0161
	256	0.0123	0.0141	0.0160	0.0160
	512	0.0124	0.0142	0.0160	0.0160
	1024	0.0122	0.0140	0.0161	0.0161

Table 3.7: Benchmark result in seconds per matrix-vector multiplication with different worksizes

Hardware	local work size	global work size	16 non-zero elements per row	32 non-zero elements per row	64 non-zero elements per row	random number of non-zeros per row, with mean = 40 and deviation = 4	random number of non-zeros per row, with mean = 40 and deviation = 8	random number of non-zeros per row, with mean = 40 and deviation = 20
GeForce GTX 470	16	64	0.0043	0.011	0.023	0.014	0.014	0.014
Tesla C2050	16	64	0.0040	0.011	0.021	0.013	0.014	0.013
GeForce GTX 470	16	1024	0.0043	0.012	0.024	0.016	0.016	0.015
Tesla C2050	16	1024	0.0039	0.011	0.022	0.015	0.014	0.015
GeForce GTX 470	32	1024	0.0047	0.014	0.029	0.018	0.018	0.018
Tesla C2050	32	1024	0.0044	0.015	0.030	0.020	0.019	0.019
GeForce GTX 470	128	128	0.0056	0.018	0.035	0.023	0.023	0.023
Tesla C2050	128	128	0.0061	0.022	0.044	0.029	0.029	0.029

Table 3.8: Benchmark result in seconds per matrix-vector multiplication with different worksizes and different number of non-zero entries per row

in Table 3.8. It can be seen that small number of non-zero elements per row perform better than greater numbers. The affect of a randomness in the number of non-zero elements per row is very small.

Chapter 4

ViennaCL

In this chapter, the C++ library ViennaCL (Vienna Computing Library) is presented [17]. Section 4.1 gives a short introduction to the Basic Linear Algebra Subprograms. In Section 4.2 the library is introduced. The OpenCL management of ViennaCL is presented in Section 4.3. The programming technique expression templates, which is used by ViennaCL is discussed in Section 4.4. Pros and cons of the library are presented in Section 4.5.

4.1 BLAS and uBLAS

The Basic Linear Algebra Subprograms, in short BLAS, is a software library which covers basic linear algebra operations. BLAS implements a lot of different vector and matrix operations. The library is initially written in Fortran, but many ports and bindings for other programming languages are available. BLAS is widely spread and a de-facto standard interface for linear algebra computing libraries. Because of its popularity there are many highly optimized implementations for different platforms available. BLAS is split up into 3 levels, each covering different linear algebra operations. BLAS level 1 contains vector operations with complexity $O(N)$: operation of the form $\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}$, dot products and vector norms. BLAS level 2 contains operations with complexity $O(N^2)$: matrix-vector operations of the form $\mathbf{y} \leftarrow \alpha A \times \mathbf{x} + \mathbf{y}$ and the solution of linear equation systems given by triangular matrices. BLAS level 3 contains operations with complexity $O(N^3)$: matrix-matrix operations of the form $C \leftarrow \alpha A \times C + \beta C$. The library supports single and double precision floating point numbers as well as complex data types in either single or double precision.

BLAS functionality is among other libraries provided by uBLAS for the programming language C++. uBLAS is a part of the boost C++ libraries which are a de-facto standard extension for C++. One of the primary goals of uBLAS is mathematical notation. Classical BLAS consists of a set of low-level functions, resulting in code which is hard to read. The operator overloading in C++ enables the use of a mathematical notation. For example a vector addition can be written as $z = x + y$ instead of using function calls. These abstractions in uBLAS rely on heavy use of C++ templates. In contrast to virtual function calls, templates can be used to eliminate penalty at runtime at the cost of increased compilation times. Additionally uBLAS uses expression templates to reduce the number of temporary objects. Expression templates will be discussed in Section 4.4. Due the use of templates the uBLAS classes are very flexible. Many different storage layouts for matrices and vectors are supported. The default storage layout is compatible to C arrays, but it is also possible to use Fortran data layout. Due to some remaining overhead of C++ classes, uBLAS is slightly slower than the classic BLAS implementation written in Fortran.

4.2 Introduction and Motivation

As presented in Section 3.3, GPUs can be used for high-performance linear algebra operations. The idea behind ViennaCL is to provide a BLAS library using the power of parallel computing architectures to perform such tasks. Like uBLAS, ViennaCL is an implementation of BLAS. The library is written in C++, but instead of executing the linear algebra operations on the CPU natively, OpenCL is used to perform the operations on an OpenCL device. ViennaCL is designed to be as much as compatible to uBLAS as possible. In addition, ViennaCL provides iterative solvers for linear equation systems including some preconditioners. Like uBLAS, ViennaCL uses C++ templates and therefore it is possible to use those solvers with other libraries. The user of the library does not need to explicitly call an *init()* function. ViennaCL automatically manages OpenCL objects in the background.

Listing 4.1 shows a simple scaled vector addition. ScalarType can be either float or double in ViennaCL 1.2.1. *viennacl::vector* and *viennacl::matrix* provide uBLAS-conforming iterators. Transferring memory from CPU to GPU, from GPU to GPU or from GPU to CPU is accomplished using standard copy functions with iterators provided by ViennaCL. For scalar types, *operator=* is overloaded to transfer values between CPU and GPU. The syntax for the linear algebra operation is the same as with uBLAS, which is straightforward, and easy to use.

Listing 4.1: Scaled vector addition example

```
ublas::vector<ScalarType> ublas_vector_x(SIZE);
ublas::vector<ScalarType> ublas_vector_y(SIZE);
ScalarType ublas_scalar;

viennacl::vector<ScalarType> vcl_vector_x(SIZE);
viennacl::vector<ScalarType> vcl_vector_y(SIZE);
viennacl::scalar<ScalarType> vcl_scalar;


// copy the data from the CPU to the OpenCL device
viennacl::copy(ublas_vector_x.begin(), ublas_vector_x.end(), vcl_vector_x.begin());
viennacl::copy(ublas_vector_y.begin(), ublas_vector_y.end(), vcl_vector_y.begin());
vcl_scalar = ublas_scalar; // implicit transfer for scalars


ublas_vector_y += ublas_scalar * ublas_vector_x;    // performing the calculation on
    the CPU
vcl_vector_y += vcl_scalar * vcl_vector_x;  // using OpenCL for calculations
```

ViennaCL provides classes for vectors, dense matrices and sparse matrices. The coordinate matrix and the compressed matrix presented in Section 3.1 are supported. All vector and matrix types have built-in support for alignment of the data arrays. Besides basic BLAS functionality, ViennaCL provides additional algorithms from the field of linear algebra. These algorithms are implemented generically so they can be used with data types from ViennaCL as well as with uBLAS or similar libraries. For directly solving systems of linear equations, a LU factorization and substitution are provided. In ViennaCL 1.2.1 pivoting is not included so numerical errors might give results of poor accuracy. Three iterative solvers are included: Conjugate Gradient (CG) [13], Stabilized Bi-CG [18] and Generalized Minimum Residual [19]. These iterative solvers can be configured using tags. Listing 4.2 provides a short example of how to use these solvers with types from ViennaCL or uBLAS. For accelerating these linear equation solvers, ViennaCL also provides three types of preconditioners. An incomplete LU factorization preconditioner with threshold is available but completely computed on the host CPU which might lead to low performance when used with ViennaCL data types. Moreover, ViennaCL provides a Jacobi preconditioner and a row-scaling preconditioner with native OpenCL support. New features in ViennaCL 1.2.1 are: algebraic multigrid preconditioners, sparse approximate inverse preconditioners and fast Fourier transform. All preconditioners work with uBLAS as well.

Listing 4.2: Matrix solver example

```
viennacl::matrix<ScalarType> vcl_matrix(SIZE, SIZE);
viennacl::compressed_matrix<ScalarType> vcl_sparse_matrix(SIZE, SIZE);

viennacl::vector<ScalarType> vcl_rhs(SIZE);
viennacl::vector<ScalarType> vcl_scalar;

ublas::compressed_matrix<ScalarType> ublas_matrix(SIZE, SIZE);
ublas::vector<ScalarType> ublas_rhs(SIZE);

// Solving a linear equation system with a dense matrix directly using LU factorization
viennacl::lu_factorize(vcl_matrix);
viennacl::lu_substitute(vcl_matrix, vcl_rhs);

// Solving a linear equation system with a sparse matrix using conjugate gradient algorithm
// A custom tag is used: tolerance is set to 1e-4 and a maximum of 10 iterations are used
viennacl::linalg::cg_tag tag(1e-4, 10);
viennacl::vector<ScalarType> vcl_result = viennacl::linalg::solve(
            vcl_matrix, vcl_rhs, tag);

// The same algorithm can be used to solve a linear equation system using uBLAS data types
ublas::vector<ScalarType> ublas_result = viennacl::linalg::solve(ublas_matrix, ublas_rhs, tag);
```

OpenCL initialization and OpenCL object management is automatically done in the background when using ViennaCL. The first device of the first platform found on the computer is used by ViennaCL by default. In addition, ViennaCL supports custom OpenCL contexts. Such a context can be created by directly using OpenCL functions and then connecting that context to ViennaCL. All ViennaCL operations can then be used in that context. This mechanism is important when using ViennaCL in an existing OpenCL environment. ViennaCL has support for custom kernels enabling a simple way to extend the library. For some complex algorithms ViennaCL might not provide the desired functionality efficiently, in which case an extra kernel can be written to perform that algorithm directly. ViennaCL provides a *kernel* class to enable user-provided kernels to interact with ViennaCL. Besides custom kernels ViennaCL data types can be configured to use custom memory objects. Both custom kernels and custom memory objects are important for custom extensions of the ViennaCL library.

4.3 OpenCL Management

The OpenCL kernels for linear algebra operations in ViennaCL are included in C++ source code. There are no extra files required for including the kernel source code. At the compilation process of the C++ program the kernel source is directly packed into the executable. At the first instantiation of a ViennaCL linear algebra object, the library passes the kernel sources belonging to that object to OpenCL. Thus, only those kernel sources which are possibly required are actually compiled by the OpenCL compiler. ViennaCL provides kernels for all basic linear algebra operations for all objects as well as specialized kernels. The operation in Listing 4.1 requires only one kernel although there are actually 2 operations: one scalar multiplication and one addition. More complex operations typically require more than one kernel.

To avoid memory leaks, all OpenCL objects have to be released after their usage. OpenCL provides a simple reference counting mechanism used by ViennaCL. For all OpenCL objects there is a *retain()* function increasing the internal reference counter and a *release()* function which decreases the internal reference counter. The internal reference counter starts with a value of one when the object is created and if the reference counter is zero, the object is destroyed by OpenCL. This reference counting mechanism is like a smart pointer using C functions for increasing and decreasing the internal reference counter. All OpenCL objects used by ViennaCL are managed with a handle object. A handle object encapsulates a basic OpenCL object and uses the retain and release functions of that object to perform reference counting. In each constructor call of a handle the reference count is incremented and in each destructor call it is decremented. Custom objects created by the user and passed to ViennaCL as custom objects are also managed by ViennaCL handles.

4.4 Expression Templates

When writing a vector class in C++, operator overloading can be used to ensure clarity and readability of mathematical instructions. However, operator overloading can also lead to poor performance due to the generation of temporary objects. For example, the operation presented in listing 4.1 would generate one temporary object. With classic operator overloading the return type of an operator is typically an object. So the operation *scalar * vector* generates a temporary vector object which will be added in-place to the other vector. Listing 4.3 provides an example for simple operator overloading for vector addition. The operation at the end of the example generates two temporary vectors, one for each addition.

Listing 4.3: Operator overloading example

```cpp
// a simple vector class
template<class TYPE>
class Vector : public std::vector<TYPE>
{};

// operator+ take two Vector objects and generates a temporary Vector object as
    result
template<class TYPE>
Vector<TYPE> operator+(
    const Vector<TYPE> & lhs,
    const Vector<TYPE> & rhs)
{...}

Vector<ScalarType> v1, v2;
v2 = v1 + v1 + v1; // this operation generates 2 temporary Vector objects, one for
    each addition
```

Expression Templates are a C++ meta programming mechanism to avoid certain types of temporary objects. With expression templates an operator returns an operator expression object instead of a whole vector object. A vector expression is an object representing a specific mathematical expression. It provides a read-only access operator evaluating the vector expression. For example, *operator+* returns an addition vector expression which holds both vectors of the addition. The access operator of this addition vector expression evaluates the operation. An *operator=* can be implemented to assign a vector expression to a vector object. This operator iterates over all elements and uses the access operator of the vector expression to evaluate the expression. With this mechanism no temporary object is required. Using C++ templates, vector expressions can be created recursively and complex vector operations are supported.

In Listing 4.4 an example of the expression template technique is shown. The addition at the end of the example generates a recursive *VectorAddExpression* object holding a *Vector* object and another *VectorAddExpression* object holding two *Vector* objects. This recursive *VectorAddExpression* object is evaluated in the *operator=* of the *Vector* class. The object tree of this operation is presented in Figure 4.1. Except for the *VectorAddExpression* objects this operation does not generate any other temporary objects, particularly no temporary *Vector* object with large overhead in construction.

Listing 4.4: Expression template example

```
// a simple vector class
template<class TYPE>
class Vector : public std::vector<TYPE>
{
public:

    // the operator= assigns an expression to the vector, the very type of the
    // expression is unknown. The expression object has to provide an operator[]
    template<class EXPRESSION>
    Vector<TYPE> & operator=(const EXPRESSION & expr)
    {
        for (size_t i = 0; i < size(); ++i)
            (*this)[i] = expr[i];
        return *this;
    }
};

// An object which represents a vector addition expression.
// The operator[] evaluates the addition
template<class EXPRESSION1, class EXPRESSION2>
class VectorAddExpression
{
public:
    typedef typename EXPRESSION1::value_type value_type;

    VectorAddExpression(
        const EXPRESSION1 & _lhs,
        const EXPRESSION2 & _rhs)
        : lhs(_lhs), rhs(_rhs) {}

    value_type operator[] (const size_t i) const { return lhs[i] + rhs[i]; }

private:
    const EXPRESSION1 & lhs;
    const EXPRESSION2 & rhs;
};

// The operator+ takes two expression (a Vector or any other expression) and returns
// a VectorAddExpression object
template<class EXPRESSION1, class EXPRESSION2>
VectorAddExpression<EXPRESSION1, EXPRESSION2> operator+(
    const EXPRESSION1 & lhs,
    const EXPRESSION2 & rhs)
{   return VectorAddExpression<EXPRESSION1, EXPRESSION2>(lhs, rhs); }


Vector<ScalarType> v1, v2;
v2 = v1 + v1 + v1;  // this operation generates 2 VectorAddExpression objects and
                    // the operator= of the Vector class triggers the evaluation of
                    // the VectorAddExpressions. No large temporary objects are
                    // generated
```

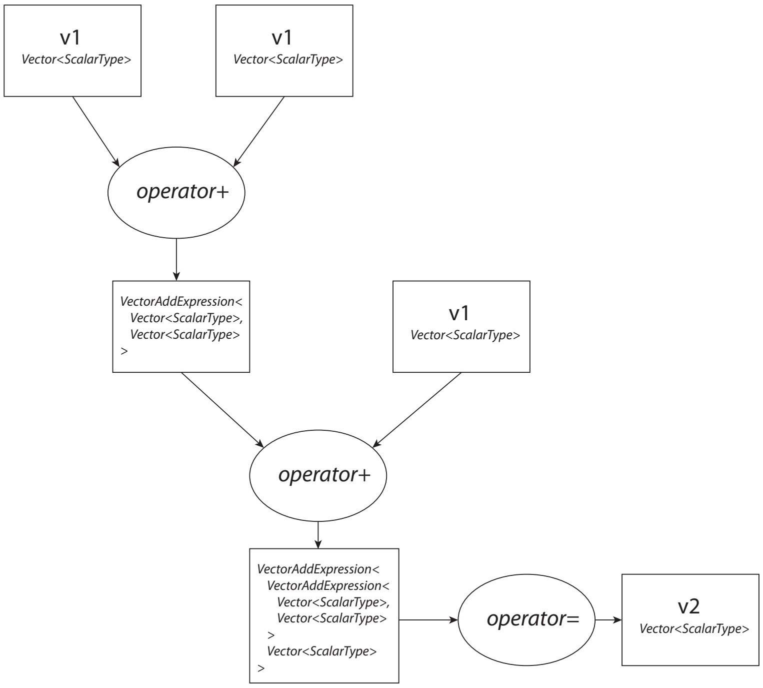


Figure 4.1: Object tree of the operation *v2 = v1 + v2 + v2;* with expression templates

Operator overloading, with or without expression templates, leads to bad performance in some cases. When using operator overloading for matrices and vectors, the operation $\mathbf{y} \leftarrow A \times B \times \mathbf{x}$ uses more operations than required. If no parentheses are used, C++ generates a matrix-matrix multiplication expression and then a matrix-expression vector expression. From the calculation complexities point of view is this the same as performing a matrix-matrix multiplication and then a matrix-vector multiplication, instead of two matrix-vector multiplications. This leads to a complexity of $O(N^3)$ instead of $O(N^2)$. To avoid such cases, explicit parentheses are required as shown in Listing 4.5. The expression template mechanism does not compensate this problem.

Listing 4.5: Expression template problem example

```
    VectorType v1, v2;
    MatrixType m1, m2;

    v1 = m1 * m2 * v2;          // cubic complexity
    v1 = m1 * (m2 * v2);        // quadratic complexity
```

In addition expression templates have some additional disadvantages. Due to template usage, the expression template mechanism is a compile-time technique. When using dynamic polymorphism, expression templates are not able to evolve their full potential. Expression templates make heavy use of the C++ template technique which might lead to longer compilation times. In addition the expression template mechanism has some restrictions when used in separate compile units. A separate compilation unit has to provide compiled code for all possible definitions, which is essentially impossible with expression templates.

uBLAS makes heavy use of expression templates as well. All vector and matrix operations are implemented using this technique. ViennaCL also uses expression templates but beside basic operations only a few more complex operations are explicitly mapped to kernels. All other operations use expression templates but may create temporary objects. For example the in-place addition with scalar multiplication in Listing 4.1 is an operation which does not require any temporary. Full expression template support for ViennaCL is difficult to implement because for every complex operation a custom kernel has to be generated. The source code of this kernel has to be written entirely automatically. For complex operations this requires tricky code generation and might be limited due the restriction of kernel parameters [20].

4.5 Pros and Cons

ViennaCL provides a simple way to perform BLAS operations on GPUs. OpenCL management, object creation and deletion is completely handled internally and hidden from the user. ViennaCL is mostly syntax-compatible to uBLAS, which allows generic algorithms to work with ViennaCL as well as with other libraries like uBLAS. Some algorithms, like iterative solvers and preconditioners, provided by ViennaCL already work with uBLAS, Eigen [25] and MTL4 [26]. On the other hand, generic algorithms designed for uBLAS usually work with ViennaCL as well. ViennaCL is a header-only library, which means that no extra compilation units nor compiled binaries are required.

In contrast to uBLAS, coding optimized code with ViennaCL is not simple. Because the OpenCL management is hidden to the user it is often not clear which OpenCL kernels are used and optimization is not straightforward. For example, accessing a single element of a matrix or a vector is a simple operation, which is actually quite slow. To access one element, a memory transfer operation has to be enqueued, which is, compared to the size of one element, a huge overhead. Setting all elements of a vector in a loop will thus result in bad performance. In this case it is better to create a temporary vector in host memory, set the elements of this vector and then copy the vector to a ViennaCL vector. Listing 4.6 provides an example.

Listing 4.6: Performance problems with ViennaCL

```cpp
viennacl::vector<ScalarType> vcl_vector;

// for each element in vcl_vector a single transfer job is enqueued
for (viennacl::vector<ScalarType>::size_type i = 0; i < vcl_vector.size(); ++i)
    vcl_vector[i] = 0;

// a temporary vector is needed but only one transfer job is enqueued
std::vector<ScalarType> std_vector;
for (std::vector<ScalarType>::size_Type i = 0; i < std_vector.size(); ++i)
    std_vector[i] = 0;
viennacl::copy(std_vector.begin(), std_vector.end(), vcl_vector.begin());
```

All ViennaCL objects require OpenCL calls for data manipulation. For each operation a kernel has to be enqueued. Thus, every ViennaCL operation results in OpenCL API overhead. Compared to the performance gain of the operation, the OpenCL API overhead is too large for small matrices and vectors. This overhead is especially significant for ViennaCL scalar objects which represent one single scalar. Those objects are very small and operations are costly. On NVIDIA architecture, a single scalar read operation of a vector takes about 91 μs [21]. So it is not recommended to use ViennaCL for small matrices and vectors and the user has to be aware of the performance issues when using ViennaCL scalars.

Chapter 5

The Finite Element Method

Partial differential equations, in short PDEs, are widely used in physical applications. The most prominent representative is the Poisson equation

$$\Delta u = \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2} = f(x), \qquad x \in \Omega, \tag{5.1}$$

which is fulfilled for example by the electric potential in an electronic device. Equations of this type are described in Section 5.1 [15] [16]. The finite element method, a mathematical framework for a solution of these differential equations, is described in Section 5.2 [13] [14]. An example of a numerical solution of a partial differential equation of elliptic type is given in Section 5.3. The mathematical background is provided in Appendix A.

5.1 Boundary Value Problems given by Elliptic Partial Differential Equations

Let $\Omega \subset \mathbb{R}^d, (d \geq 1)$ be an open set and $u \in \mathcal{C}^k(\Omega)$ for some $k > 0$. The partial derivative of u is given by

$$D^\alpha u = \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \dots \partial x_d^{\alpha_d}},$$

where $\alpha = (\alpha_1, \ldots, \alpha_d) \in \mathbb{N}$ is a multi-index and $|\alpha| = \alpha_1 + \cdots + \alpha_d$. A linear partial differential equation of order k is given by

$$\sum_{|\alpha| \leq k} a_\alpha(x) D^\alpha u = f(x), \qquad x \in \Omega, \tag{5.2}$$

where $a_\alpha : \Omega \to \mathbb{R}, f : \Omega \to \mathbb{R}$ are arbitrary functions.

Definition 1 (Elliptic partial differential equation). *1. A linear partial differential equation is called* elliptic *in a point $x \in \Omega$, if the matrix $(a_{ij}(x))$ is positive or negative definite.*

2. A linear partial differential equation is called elliptic *in Ω if the equation is elliptic in almost all $x \in \Omega$.*

Poisson's equation (5.1) is a partial differential equation of elliptic type. The equation is obviously linear and the matrix (a_{ij}) is equal to the identity matrix, which is positive definite.

A function $u : \Omega \to \mathbb{R}$ is called (classical) solution to (5.2) if $u \in \mathcal{C}^k$ and u satisfies (5.2). Without additional constraints, PDEs usually lead to an infinite number of solutions. To get a unique solution, boundary condition need to be imposed. A boundary condition prescribes values of the solution or the derivative of the solution at the boundary of the domain where the differential equation is defined. Two common boundary condition are:

1. A *Dirichlet boundary condition* specifies the values of the solution on the boundary of the domain.
2. A *Neumann boundary condition* specifies the values of the derivate of the solution on the boundary of the domain.

Typically, the boundary of the domain has to be continuous or continuously differentiable except for a finite number of points. It is also possible to use boundary conditions of different type on disjoint continuously differentiable segments. A (partial) differential equation together with one or more boundary conditions is called a boundary value problem. Similar to a differential equation, a function $u : \Omega \to \mathbb{R}$ is called (classical) solution to the boundary value problem, if $u \in \mathcal{C}^k$ and u satisfies the differential equation as well as all boundary conditions. Some boundary value problems do not have classic solutions. Therefore, the definition of a weak solution is motivated.

5.1.1 Weak derivative and weak formulation

Definition 2 (Weak derivative)**.** *Let* $u \in L^p(\Omega), 1 \leq p \leq \infty$ *and* $\alpha \in \mathbb{N}^d$ *be a multi-index. The function* $D^\alpha u \in L^p(\Omega)$ *is called weak derivative of* u*, if*

$$(D^\alpha u, v)_{L^2} = (-1)^{|\alpha|}(u, D^\alpha v)_{L^2}, \qquad \forall v \in C_0^\infty(\Omega) \tag{5.3}$$

For $u \in C^\alpha(\Omega)$ the weak derivative and the classical derivative is the same. This can be shown easily by partial integration. Thus, the weak derivative is a true generalization of the classical derivative. For example the function $f(x) = |x|$ does not have a classical derivative, but a weak derivative $f'(x) = \operatorname{sgn}(x)$. Actually, a weak derivative is not unique. Each function, which is equal to a weak derivative for all points except for a finite number, also is a weak derivative. For boundary value problems, an equivalent to the weak derivative is the weak formulation, which has to be expressed for each boundary condition and special types of differential equation separately. Many boundary value problems do not have any classical solution. thus, the weak formulation is introduced to relax the concept of the solution to a boundary value problem.

Definition 3 (Uniformly elliptic differential operator)**.** *A differential operator*

$$Lu = -\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}(a_{ij}(x)\frac{\partial u}{\partial x_j}) + c(x)u \tag{5.4}$$

is called uniformly elliptic in Ω*, if it satisfies*

$$\exists \lambda > 0 : \forall \xi \in \mathbb{R}^d, \forall x \in \Omega \setminus N, |N| < \infty : \sum_{i,j=1}^{d} a_{ij}(x)\xi_i\xi_j \geq \lambda|\xi|^2.$$

Definition 4 (Boundary value problem with trivial Dirichlet boundary value condition)**.** *A boundary value problem with trivial Dirichlet boundary value condition is defined as follows:*

- *The Domain* $\Omega \subset \mathbb{R}^d$ *is bounded and open with boundary* $\partial\Omega \in C^{0,1}$.
- *The partial differential equation is given by* $Lu = f$ in Ω*, where* L *is an uniformly elliptic operator. Additionally the following restriction have to be satisfied by the operator* L *and the function* f*:* $\quad a_{ij}, c \in L^\infty(\Omega), \quad c \geq 0 \text{ in } \Omega, \quad g \in H^1(\Omega), f \in L^2(\Omega)$.
- *The boundary condition is a trivial function:* $u \equiv 0 \text{ on } \partial\Omega$.

If a Dirichlet boundary value problem satisfies the conditions presented in Definition 4, a weak formulation of the problem can be derived for

$$Lu = -\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}(a_{ij}(x)\frac{\partial u}{\partial x_j}) + c(x)u = f.$$

Motivated by the weak derivative, the differential equation is multiplied by a function $v \in \mathcal{C}_0^\infty$ and integrated over the domain Ω:

$$-\sum_{i,j=1}^{d} \int_\Omega \frac{\partial}{\partial x_i}(a_{ij}(x)\frac{\partial u}{\partial x_j})v dx + \int_\Omega c(x)uv dx = \int_\Omega fv dx$$

Now the divergence theorem can be applied.

$$\begin{aligned} -\int_\Omega \frac{\partial}{\partial x_i}(a_{ij}(x)\frac{\partial u}{\partial x_j})v dx &= \int_\Omega a_{ij}(x)\frac{\partial u}{\partial x_j}\frac{\partial v}{\partial x_i} dx - \int_{\partial\Omega} a_{ij}(x)\frac{\partial u}{\partial x_j} v dx \\ &= \int_\Omega a_{ij}(x)\frac{\partial u}{\partial x_j}\frac{\partial v}{\partial x_i} dx \end{aligned} \tag{5.5}$$

Since $v \in \mathcal{C}_0^\infty(\Omega)$ implies $v(x) \equiv 0$ for all $x \in \partial\Omega$, the boundary integral is equal to zero. The differential equation can therefore be written in weak form as follows:

$$\begin{aligned} a(u, v) &= F(v) \\ a(u, v) &:= \sum_{i,j=1}^{d} \int_\Omega a_{ij}(x)\frac{\partial u}{\partial x_j}\frac{\partial v}{\partial x_i} dx + \int_\Omega c(x)uv dx \\ F(v) &:= \int_\Omega fv dx \end{aligned} \tag{5.6}$$

A function $u \in H_0^1(\Omega)$ is called a weak solution to the differential equation if it satisfies (5.6) for all $v \in H_0^1(\Omega)$. The lemma of Lax-Milgram guarantees the existence of a unique solution under the assumptions presented in Definition 4.

Lemma 1 (Lax-Milgram)**.** *Let V be a real Hilbert space, $a : V \times V \to \mathbb{R}$ a bilinear form and $F \in V'$ an element of the dual space of V. The bilinear form a satisfies:*

1. *a is bounded:* $\exists K > 0 : \forall u, v \in V : |a(u, v)| \leq K\|u\|_V\|u\|_V$
2. *a is coercive:* $\exists \lambda > 0 : \forall u \in V : a(u, u) \geq \lambda\|u\|_V^2$

Under these assumptions there exists a unique element $u \in V$ *such that*

$$a(u, v) = F(u) \qquad \forall v \in V \tag{5.7}$$

The lemma of Lax-Milgram can be proven by using Riesz' representation theorem, a fundamental theorem in functional analysis. The interested reader is referred to [13] for a proof.

Lemma 2 (Existence and Uniqueness of a weak solution of a boundary value problem with Dirichlet boundary condition)**.** *Under assumptions of Definition 4 there exists a unique weak solution* $u \in H^1(\Omega)$ *to the weak formulation* (5.6).

Proof. The conditions in Lemma 1 have to be checked for $V = H_0^1(\Omega)$:

1. a is bounded: Using the Cauchy-Schwarz inequality (see Appendix A) the bilinear form can be bounded:

$$|a(u,v)| \leq \sum_{i,j=1}^{d} \int_\Omega |a_{ij}| \left|\frac{\partial u}{\partial x_j}\right| \left|\frac{\partial v}{\partial x_i}\right| dx + \int_\Omega |c|\,|u|\,|v| dx \tag{5.8}$$

$$\leq \max_{i,j=1,\dots,d} \{\|a_{ij}\|_{L^\infty}, \|c\|_{L^\infty}\} (\sum_{i,j=1}^{d} \left\|\frac{\partial u}{\partial x_j}\right\|_{L^2} \left\|\frac{\partial v}{\partial x_i}\right\|_{L^2} + \|u\|_{L^2}\|v\|_{L^2}) \tag{5.9}$$

$$= K(\|\nabla u\|_{L^2}\|\nabla v\|_{L^2} + \|u\|_{L^2}\|v\|_{L^2}) \tag{5.10}$$

$$\leq K\|u\|_{H^1}\|v\|_{H^1}, \tag{5.11}$$

where $K = \max_{i,j=1,\dots,d}\{\|a_{ij}\|_{L^\infty}, \|c\|_{L^\infty}\}$.

2. a is coercive:

$$a(u,u) = \sum_{i,j=1}^{d} \int_\Omega a_{ij}(x) \frac{\partial u}{\partial x_j}\frac{\partial u}{\partial x_i} dx + \underbrace{\int_\Omega c(x)u^2 dx}_{\geq 0} \tag{5.12}$$

$$\geq \sum_{i,j=1}^{d} \int_\Omega a_{ij}(x) \frac{\partial u}{\partial x_j}\frac{\partial u}{\partial x_i} dx \tag{5.13}$$

$$\geq \lambda \sum_{i=1}^{d} \int_\Omega \left|\frac{\partial u}{\partial x_i}\right|^2 dx \tag{5.14}$$

$$= \lambda \|\nabla u\|_{L^2}^2 \tag{5.15}$$

$$\geq \lambda_0 \|u\|_{H^1}^2, \tag{5.16}$$

where $\lambda_0 = \lambda/(C_p^2+1)$ and $C_p > 0$ is the constant resulting from the Poincare inequality (see Appendix A).

3. $F \in V' = H^{-1}(\Omega)$: Let $v \in V$. Using the Cauchy-Schwarz inequality the following estimation can be obtained:

$$\begin{aligned} |F(v)| &\leq \|f\|_{L^2}\|v\|_{L^2} \\ &\leq \|f\|_{L^2}\|v\|_{H^1} \end{aligned} \tag{5.17}$$

   and hence $F \in H^{-1}$.

With Lemma 1 there exists a unique $u \in V$ that satisfies (5.21). □

The boundary value problem presented in Definition 4 is restricted to $u(x) \equiv 0, x \in \partial\Omega$. This kind of Dirichlet boundary condition is too restrictive, since in general, $u(x) = g(x) \neq 0, x \in \partial\Omega$. To achieve this, the previously presented boundary value problem can be transformed. Instead of calculating a function u with the boundary condition $u(x) = g(x), x \in \partial\Omega$, a function $w = u - g \in H_0^1(\Omega)$ with the boundary condition $w(x) = 0, x \in \partial\Omega$ is calculated. The new boundary value problem becomes:

$$a(w,v) = G(v) \tag{5.18}$$

$$a(u,v) = \sum_{i,j=1}^{d} \int_\Omega a_{ij}(x) \frac{\partial u}{\partial x_j}\frac{\partial v}{\partial x_i} dx + \int_\Omega c(x)uvdx \tag{5.19}$$

$$G(v) = \int_\Omega fvdx - \sum_{i,j=1}^{d} \int_\Omega a_{ij}(x) \frac{\partial g}{\partial x_j}\frac{\partial v}{\partial x_i} dx - \int_\Omega c(x)uvdx \tag{5.20}$$

The solution $u = w + g$ is the weak solution to the transformed problem (5.18). The lemma of Lax-Milgram also ensures a unique solution to this transformed problem, the proof has to be adapted only slightly in the third step.

5.2 The Ritz-Galerkin Method and Finite Elements

Finding a solution to the weak formulation (5.6) is practically impossible for an infinite-dimensional Hilbert space V. Instead of calculating the weak solution $u \in V$, an approximation $u_h \in V_h$ is calculated, where V_h is a finite-dimensional linear subspace of V. The weak formulation of the boundary value problem changes to

$$a(u_h, v) = F(v) \qquad \forall v \in V_h \tag{5.21}$$

u_h is called Ritz-Galerkin approximation. Since a and F are linear, the weak formulation (5.21) is equivalent to

$$a(u_h, \phi_i) = F(\phi_i) \qquad i = 1, \ldots, N, \tag{5.22}$$

where $\{\phi_1, \ldots, \phi_N\}$ is a linear basis of V_h. The solution u_h can be represented as a linear combination of basis functions:

$$u_h = \sum_{j=1}^{N} \hat{u}_j \phi_j \tag{5.23}$$

Using this representation and the fact that a is bilinear, the following linear equation system can be formulated:

$$a(u_h, \phi_i) = a\left(\sum_{j=1}^{N} \hat{u}_j \phi_j, \phi_i\right) = \sum_{j=1}^{N} a(\phi_j, \phi_i)\hat{u}_j = F(\phi_i) \qquad i = 1, \ldots, N \tag{5.24}$$

Where $\hat{u}_1, \ldots, \hat{u}_N$ are unknown. With $A = (A_{ij}) = (a(\phi_j, \phi_i)), \underline{b} = (b_i) = (F(\phi_i))$ and $\underline{u} = (\hat{u}_i)$ on obtains the linear system.

$$A\underline{u} = \underline{b}. \tag{5.25}$$

Lemma 3 (Solvability of the Ritz-Galkerin approximation). *If a is coercive, then the Ritz-Galkerin approximation* (5.21) *is uniquely solvable in V_h.*

Proof. Since a is coercive in V, it also is coercive in V_h:

$$\underline{x}^T A \underline{x} = \sum_{i,j=1}^{N} a(\phi_j, \phi_i)\underline{x}_i \underline{x}_j = a(\underline{x}, \underline{x}) \geq \lambda \|\underline{x}\|_V^2, \qquad \underline{x} = \sum_{i=1}^{N} \underline{x}_i \phi_i. \tag{5.26}$$

If $A\underline{x} = 0$ then $\underline{x} = 0$. Therefore, A is injective and the linear equation system is uniquely solvable. □

To calculate the Ritz-Galerkin approximation to a weak solution of a boundary value problem, the matrix A and the vector $\underline{b}$ have to be calculated and the system of linear equations has to be solved. It is preferable that the calculation of the entries of the matrix A and the vector $\underline{b}$ is cheap. It is also preferable that the matrix A is very sparse and can be inverted easily. The finite element method (FEM) is a Ritz-Galerkin approximation, which achieves these requirements.

5.2.1 Finite Elements

The finite element method is split up into four parts:

1. The domain Ω is tessellated into small sets, mostly d-Simplices.
2. Basis functions, with smallest possible support, are defined.
3. The matrix A and the vector b in (5.25) are calculated.
4. The linear equation system (5.25) is solved.

Since $A_{ij} := a(\phi_j, \phi_i) = 0$ for $(\operatorname{supp}\phi_i)^O \cup (\operatorname{supp}\phi_j)^O = \emptyset$, basis functions with small support are desirable. In order to use the finite element method efficiently, some restrictions are imposed to the tessellation of the domain Ω.

Definition 5 (d-Simplex). *Let $\underline{a}_1, \dots, \underline{a}_{d+1} \in \mathbb{R}^d$ be some points which are not included in any hyperplane of $\mathbb{R}^d$. A set $\tau \subseteq \mathbb{R}^d$ is called d-Simplex if it is equal to the convex hull of the points $\underline{a}_1, \dots, \underline{a}_{d+1} \in \mathbb{R}^d$:*

$$\tau = \left\{ x \in \mathbb{R}^d : x = \sum_{i=1}^{d+1} \lambda_i \underline{a}_i, \quad 0 \le \lambda_1, \dots, \lambda_{d+1} \le 1, \quad \sum_{i=1}^{d+1} \lambda_i = 1 \right\} \tag{5.27}$$

A 2-Simplex is a triangle, a 3-simplex is a tetrahedron. d-Simplexes are simple subsets of a finite dimensional vector space and suit very well for the segmentation of the domain Ω.

Definition 6 (Triangulation). *Let $\Omega \subset \mathbb{R}^d$ be a bounded subset of the vector space $\mathbb{R}^d$. A segmentation $T_h \subset \mathcal{P}(\Omega)$ of Ω is called triangulation if it satisfies the following:*

1. *All elements $\tau \in T_h$ are closed and the interior $\tau^0 \neq \emptyset$ is connected. The boundary $\partial\tau \in C^{0,1}$.*
2. *The closure of the domain Ω is the union of all elements τ: $\overline{\Omega} = \bigcup_{\tau \in T_h} \tau$.*
3. *The intersection of the interior of two elements τ_1 and τ_2 is empty: $\tau_1^O \cap \tau_2^O = \emptyset \quad \forall \tau_1, \tau_2 \in T_h, \tau_1 \neq \tau_2$.*

Definition 7 (Valid Triangulation)**.** *A triangulation T_h of Ω is called valid if all edges of all $\tau_1 \in T_h$ are either an edge of another $\tau_2 \in T_h$ or a part of $\partial\Omega$.*

Although finite elements can also be defined on arbitrary polytopes, a valid triangulation is a desired tessellation of the domain Ω. A finite element can now be defined.

Definition 8 (Finite Element)**.** *A finite element in the vector space $\mathbb{R}^d$ is a triplet $(\tau, P_\tau, \Sigma_\tau)$ with the following properties:*

1. *$\tau \subset \mathbb{R}^d$ is a closed d-Simplex*
2. *P_τ is a finite-dimensional space of functions $\tau \to \mathbb{R}$, $n := \dim P_\tau$*
3. *Σ_τ is a set of linearly independent continuous functionals $B_1, \ldots, B_n : \mathcal{C}^s(\tau) \to \mathbb{R}, s \in \mathbb{N} : \forall \alpha_1, \ldots, \alpha_n \in \mathbb{R} : \exists p \in P_\tau : B_i(p) = \alpha_i, i = 1, \ldots, n.$*

For each finite element there exists a set of functions $p_1, \ldots, p_n \in P_\tau$ such that $B_i(p) = \delta_{ij}, i, j = 1, \ldots, n$. The functionals B_i are called degrees of freedom and the functions p_i are called the basis functions of this finite element. Typically, basis functions and degrees of freedom are defined for a reference finite element. The basis functions and degrees of freedom of every other finite element are generated by transformation of the basis functions and the degrees of freedom defined on the reference element.

Example 1 (Linear Lagrangian finite elements in two dimensions)**.** *A finite element is called of Lagrangian type, if all degrees of freedom define the value of the basis functions at a vertex of the d-Simplex. Linear Lagrangian finite elements use affine functions P_τ:*

$$P_\tau = P_{1,\tau} = \{p : \tau \to \mathbb{R} : \text{ p is affine in } \tau\} \tag{5.28}$$

Let τ be the d-Simplex of the reference finite element with the vertices $\underline{a}_1, \ldots, \underline{a}_{d+1} \in \mathbb{R}^d$. Then the degrees of freedom of the finite element of Lagrangian type are defines as follows:

$$B_i(p) = p(a_i), \qquad i = 1, \ldots, d+1, p \in P_\tau \tag{5.29}$$

The basis functions $p_i \in P_\sigma$ are uniquely defined by $p_j(\underline{a}_i) = \delta_{ij}, \quad i, j = 1, \ldots, d+1$. For a two dimensional vector space, the d-Simplex τ is a triangle and the reference element can be chosen with the vertices $\underline{a}_1 = (0,0)^T, \underline{a}_2 = (1,0)^T, \underline{a}_3 = (0,1)^T$, see Figure 5.1. The basis functions of this reference finite element are:

$$p_1(x,y) = 1 - x - y, \quad p_2(x,y) = x, \quad p_3(x,y) = y, \quad (x,y)^T \in \tau. \tag{5.30}$$

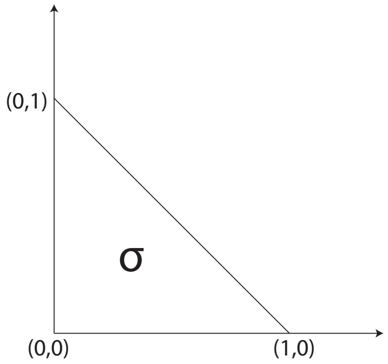


Figure 5.1: The reference triangle for the basis functions.

Definition 9 (Finite Element Space)**.** *The finite element space* X_h *is defined as*

$$X_h = \left\{ v = (v_\tau)_{\tau \in T_h} \in \prod_{\tau \in T_h} P_\tau : \forall b \in N_h : \forall \tau_1, \tau_2 \in T_h(b) : B_{b,\tau_1}(v_{\tau_1}) = B_{b,\tau_1}(v_{\tau_2}) \right\}, \quad (5.31)$$

where T_h *is a valid triangulation of the domain* Ω*,* N_h *is the set of all vertices of all finite elements and* $T_h(b)$ *and* $b \in N_h$ *is the set of all finite elements where* b *is a vertex of this element.*

Lemma 4 (Inclusion of the finite element function and the finite element space in a Sobolev space)**.** *If the function of a finite element are included in a Sobolev space, then the finite element space is a subset of the same Sobolev space. Let* X_h *be the finite element space and* $k = 0, 1$*.*

$$P_\tau \subset H^{k+1}(\tau), \forall \tau \in T_h \wedge X_h \subset \mathcal{C}^k(\overline{\Omega}) \Rightarrow X_h \subset H^{k+1}(\Omega) \quad (5.32)$$

A proof can be found in [14].

If all finite elements and the finite element space are defined, the matrix A can be calculated. A basis of the finite element space is chosen and all matrix entries are calculated: $A_{ij} = a(\phi_j, \phi_i), i, j = 1, \ldots, N$, where $\phi_1, \ldots, \phi_N$ is a basis of the finite element space. As mentioned before, this basis should contain basis elements with small support to obtain a sparse matrix. In addition, the vector b has to be calculated: $b_i = F(\phi_i), i = 1, \ldots, N$. The linear equation system now needs to be solved. The dimension of the matrix A is equal to N, the number of basis functions of the finite element space. To achieve accurate solutions to the weak formulation of a boundary value problem, system matrices are usually large. For such large linear systems of equations with sparse system matrix, iterative solvers are

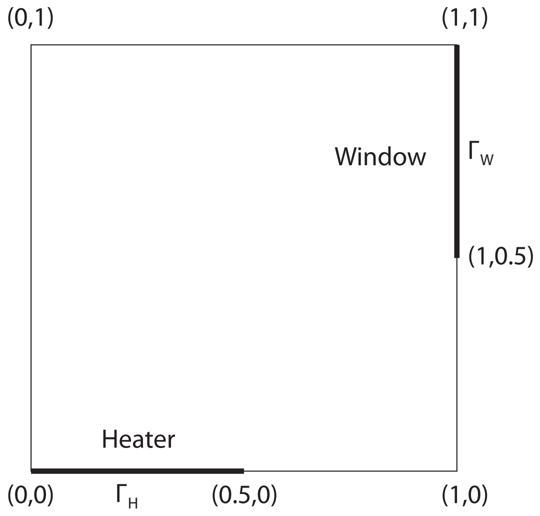


Figure 5.2: Heat distribution example

preferable. Direct solvers are not suitable because they typically destroy sparsity. A good choice for an iterative solver is the conjugate gradient method presented in Appendix B.

5.3 The Poisson Equation

In this chapter an example of a boundary value problem with Dirichlet and Neumann boundary condition is presented: the heat distribution in a quadratic room. Figure 5.2 illustrates the example. The room Ω is represented by the rectangle $(0,1)^2$. There is a heater Γ_H at the bottom left side of the room with a length of 0.5 and a window Γ_W at the top right side of the room with a length of 0.5. The room is assumed to be isolated, so $\frac{\partial u}{\partial \underline{n}} \equiv 0$ for all $x \in \Gamma_N := \partial\Omega \setminus \{\Gamma_H \cup \Gamma_W\}$. Due to the fact that Γ_H and Γ_W are heat sources, $\frac{\partial u}{\partial \underline{n}}$ is in general not equal to zero at Γ_H and Γ_W. At Γ_H the temperature of the room is fixed to 50 °C, at Γ_W the temperature of the room is constant 10 °C. The temperature at the rest of the boundary Γ_N is not defined.

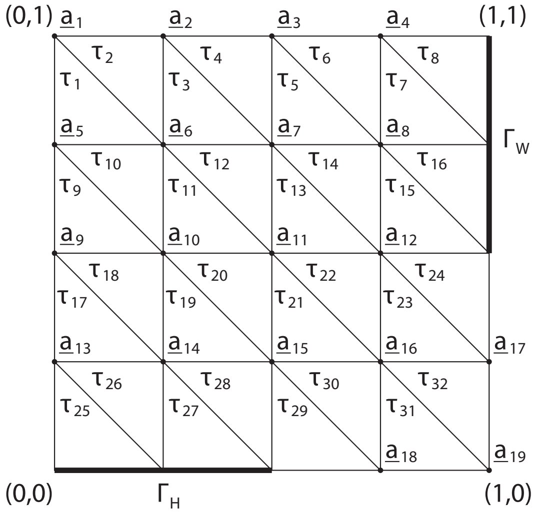


Figure 5.3: Triangulation of the domain $\Omega = (0,1)^2$

Definition 10 (Heat Distribution Example)**.** *The boundary value problem for the heat distribution is given as follows:*

$$
\begin{aligned}
-\Delta u(x) &\equiv 0, \qquad \forall x \in \Omega = (0,1)^2 \\
\frac{\partial u}{\partial \underline{n}} &\equiv 0, \qquad \forall x \in \Gamma_N \\
u(x) &= 50, \qquad \forall x \in \Gamma_H \\
u(x) &= 10, \qquad \forall x \in \Gamma_W
\end{aligned}
$$

The triangulation of the domain $\Omega = (0,1)^2$ is presented in Figure 5.3 with $h = 0.25$. Only the vertices representing degrees of freedom are labeled, for all other vertices the solution is equal to the Dirichlet boundary condition.

Due to the fact that this boundary value problem is just partially of Dirichlet type, the weak formulation of this problem has to be derived individually. First the problem is transformed to a similar problem where the Dirichlet boundary conditions are equal to zero. A function g is introduced which is linear on every element in the tessellation and defined as follows:

$$
g(x) = \begin{cases} 0, & x = \underline{a}_1, \ldots, \underline{a}_{19} \\ 50, & x \in \Gamma_H \\ 10, & x \in \Gamma_W \end{cases} \tag{5.33}
$$

The function u is a solution to the boundary value problem as stated in Definition 10, if

$w = u - g$ is a solution to the following boundary value problem.

$$\begin{aligned} Lw = f - Lg, &\qquad x \in \Omega \\ w(x) \equiv 0, &\qquad \forall x \in \partial\Gamma_H \cup \Gamma_W \\ \frac{\partial w}{\partial \underline{n}} \equiv 0, &\qquad \forall x \in \partial\Omega \end{aligned}$$

Since the values of the solution at Γ_N are not necessarily equal to zero, the solution is not in the Sobolev space $H_0^1(\Omega)$. Therefore, the domain Ω and the Sobolev space of the solution functions is extended. Instead of using the domain Ω, a new domain $\widetilde{\Omega} := \Omega \cup \Gamma_N$ is considered. The Sobolev space of the solution functions is then defined as

$$\begin{aligned} H_0^1(\widetilde{\Omega}) &:= \text{ completion of } C_0^\infty(\widetilde{\Omega}) \text{ in } \|\cdot\|_{H^1} \\ C_0^\infty(\widetilde{\Omega}) &= \Big\{ u \in C^\infty(\overline{\Omega}) : \operatorname{supp} u \subset \widetilde{\Omega} \Big\} \end{aligned}$$

The derivation of the weak formulation is similar to the one presented in Section 5.1.1. Since the boundary integral in (5.5) is zero, the Neumann boundary condition is satisfied. Thus, the weak formulation to this problem is equal to the weak formulation presented in (5.18):

$$a(u,v) = \int_\Omega \nabla u \nabla v dx = -\int_\Omega \nabla g \nabla v dx = G(v). \tag{5.34}$$

Linear finite elements are chosen:

$$X_h = \Big\{ u : \overline{\Omega} \to \mathbb{R} : u \text{ is continuous and piecewise linear} \Big\} \subset H_0^1(\Omega) \tag{5.35}$$

To match the boundary condition $u(x) = 0$ for all $x \in \Gamma_H \cup \Gamma_W$ the following finite element space is defined:

$$V_h = \{ u \in X_h : u(x) = 0, \forall x \in \Gamma_H \cup \Gamma_W \} \subset X_h \tag{5.36}$$

V_h is a linear subspace of X_h of the dimension $N = 19$. A basis $B_h = \{\phi_1, \ldots, \phi_{16}\} \subset V_h$ with small support is chosen:

$$B_h := \Big\{ \phi_i \in V_h : \phi_i(\underline{a}_j) = \delta_{ij}, i, j = 1, \ldots, 19 \Big\} \tag{5.37}$$

The basis function on the reference triangle $\hat{\tau}$ presented in Figure 5.1 with the vertices

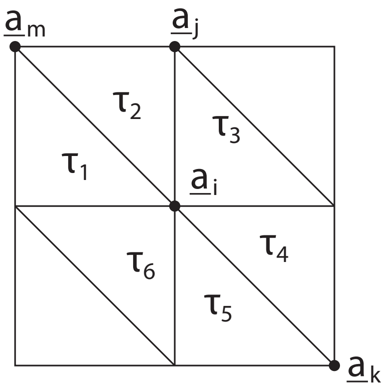


Figure 5.4: Calculation of $a(\cdot,\cdot)$

$a_1 = (0,0)^T, a_2 = (1,0)^T a_3 = (0,1)^T$ are as follows:

$$\hat{p_1}(x,y) = 1 - x - y, \quad \hat{p_2}(x,y) = x, \quad \hat{p_3}(x,y) = y, \quad (x,y)^T \in \hat{\tau} \tag{5.38}$$

These basis functions can be transferred to any triangle τ_i in the segmentation by

$$p_1(x,y) = 1 - \frac{x - x_1}{h} - \frac{y - y_1}{h}, \quad p_2(x,y) = \frac{x - x_1}{h}, \quad p_3(x,y) = \frac{y - y_1}{h} \tag{5.39}$$

where $\underline{a}_1 = (x_1, y_1)^T$.

The matrix entries $A_{ij} = a(\phi_j, \phi_i)$ can now be calculated:

$$a(\phi_j, \phi_i) = \int_\Omega \nabla\phi_j \nabla\phi_i dx \tag{5.40}$$

$$= \begin{cases} \text{const} & : \ (\operatorname{supp}\phi_i)^O \cap (\operatorname{supp}\phi_j)^O \neq \emptyset \\ 0 & : \ \text{otherwise} \end{cases} \tag{5.41}$$

$$= \begin{cases} \text{const} & : \ a_i \text{ and } a_j \text{ share the same triangle} \\ 0 & : \ \text{otherwise} \end{cases} \tag{5.42}$$

Figure 5.4 shows the local finite elements which will be affected in the calculation of $a(\phi_i, \phi_j)$.

$$
\begin{aligned}
a(\phi_i,\phi_i) &= \sum_{i=1}^{6}\int_{\tau_i}|\nabla\phi_i|^2\,dx\\
&= 2\int_{\tau_3}|\nabla p_1|^2\,dx + 4\int_{\tau_1}|\nabla p_2|^2\,dx\\
&= 2\frac{2}{h^2}\,\text{meas}(\tau_3) + 4\frac{1}{h^2}\,\text{meas}(\tau_1)\\
&= 4\\
a(\phi_j,\phi_j) &= \int_{\tau_2}|\nabla p_1|^2\,dx + 2\int_{\tau_3}|\nabla p_3|^2\,dx\\
&= \frac{2}{h^2}\,\text{meas}(\tau_3) + 2\frac{1}{h^2}\,\text{meas}(\tau_3)\\
&= 2\\
a(\phi_k,\phi_k) &= 2\int_{\tau_4}|\nabla p_2|^2\,dx\\
&= 2\frac{1}{h^2}\,\text{meas}(\tau_3)\\
&= 1\\
a(\phi_i,\phi_j) &= \int_{\tau_2}\nabla p_3\nabla p_1 dx + \int_{\tau_3}\nabla p_1\nabla p_3 dx\\
&= -\frac{1}{h^2}\,\text{meas}(\tau_2) - \frac{1}{h^2}\,\text{meas}(\tau_3)\\
&= -1\\
a(\phi_m,\phi_j) &= \int_{\tau_2}\nabla\phi_1\nabla\phi_3 dx\\
&= -\frac{1}{h^2}\,\text{meas}(\tau_2)\\
&= -\frac{1}{2}\\
a(\phi_i,\phi_k) &= \int_{\tau_4}\nabla p_3\nabla p_2 dx + \int_{\tau_5}\nabla p_2\nabla p_3 dx\\
&= 0
\end{aligned}
$$

where $\text{meas}(\tau_i) = \frac{h^2}{2}$.

The matrix A looks like:

$$A = \begin{pmatrix}
1 & -\frac{1}{2} & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-\frac{1}{2} & 2 & -\frac{1}{2} & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & 2 & -\frac{1}{2} & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -\frac{1}{2} & 2 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-\frac{1}{2} & 0 & 0 & 0 & 2 & -1 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 2 & -1 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 0 & -\frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 2 & -\frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & -\frac{1}{2} & 1
\end{pmatrix}$$

The vector $\underline{b} = (b_i) = (F(\phi_i))$ is calculated similarly:

$$\begin{aligned}
b_i &= F(\phi_i) \\
&= \int_\Omega \nabla g \nabla \phi_i dx \\
&= \begin{cases} \text{const} & : \ (\operatorname{supp} g)^O \cap (\operatorname{supp} \phi_j)^O \neq \emptyset \\ 0 & : \ \text{otherwise} \end{cases} \\
&= \begin{cases} \text{const} & : \ i = 4, 8, 12, 13, 14, 15, 17, 18 \\ 0 & : \ \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
F(\phi_4) &= -\sum_{i=7,8}\int_{\tau_i}\nabla g\nabla\phi_4 dx\\
&= -10\left(\int_{\tau_8}\nabla(p_1+p_2)\nabla p_3 dx + \int_{\tau_7}\nabla p_2\nabla p_3 dx\right)\\
&= -10\left(-\frac{1}{2}+0\right) = 5\\
F(\phi_8) &= -\sum_{i=7,15,16}\int_{\tau_i}\nabla g\nabla\phi_8 dx\\
&= -10\left(\int_{\tau_{15}}\nabla p_2\nabla p_3 dx + \int_{\tau_{16}}\nabla(p_1+p_2)\nabla p_3 dx + \int_{\tau_7}\nabla p_2\nabla p_1 dx\right)\\
&= -10\left(0-\frac{1}{2}-\frac{1}{2}\right) = 10\\
F(\phi_{12}) &= -\sum_{i=15,24}\int_{\tau_i}\nabla g\nabla\phi_{12} dx\\
&= -10\left(\int_{\tau_{24}}\nabla p_1\nabla p_3 dx + \int_{\tau_{15}}\nabla p_2\nabla p_1 dx\right)\\
&= -10\left(-\frac{1}{2}-\frac{1}{2}\right) = 10\\
F(\phi_{17}) &= -\sum_{i=24}\int_{\tau_i}\nabla g\nabla\phi_{17} dx\\
&= -10\left(\int_{\tau_{24}}\nabla p_1\nabla p_2 dx\right)\\
&= -10\left(-\frac{1}{2}\right) = 5\\
F(\phi_{13}) &= -\sum_{i=25,26}\int_{\tau_i}\nabla g\nabla\phi_{13} dx\\
&= -50\left(\int_{\tau_{25}}\nabla(p_1+p_2)\nabla p_3 dx + \int_{\tau_{26}}\nabla p_2\nabla p_3 dx\right)\\
&= -50\left(-\frac{1}{2}+0\right) = 25\\
F(\phi_{14}) &= -\sum_{i=26,27,28}\int_{\tau_i}\nabla g\nabla\phi_{14} dx\\
&= -50\left(\int_{\tau_{26}}\nabla p_2\nabla p_1 dx + \int_{\tau_{27}}\nabla(p_1+p_2)\nabla p_3 dx + \int_{\tau_{28}}\nabla p_2\nabla p_3 dx\right)\\
&= -50\left(-\frac{1}{2}-\frac{1}{2}+0\right) = 50
\end{aligned}$$

$$\begin{aligned}
F(\phi_{15}) &= -\sum_{i=28,29}\int_{\tau_i}\nabla g\nabla\phi_{15}dx \\
&= -50\left(\int_{\tau_{28}}\nabla p_2\nabla p_1 dx + \int_{\tau_{29}}\nabla p_3\nabla p_1 dx\right) \\
&= -50\left(-\frac{1}{2}-\frac{1}{2}\right) = 50 \\
F(\phi_{18}) &= -\sum_{i=29}\int_{\tau_i}\nabla g\nabla\phi_{18}dx \\
&= -50\left(\int_{\tau_{29}}\nabla p_1\nabla p_2 dx\right) \\
&= -50\left(-\frac{1}{2}\right) = 25
\end{aligned}$$

The vector $\underline{b}$ is thus given by

$$\underline{b} = \begin{pmatrix} 0 & 0 & 0 & 5 & 0 & 0 & 0 & 10 & 0 & 0 & 0 & 10 & 25 & 50 & 50 & 0 & 5 & 25 & 0 \end{pmatrix}^T .$$

Now the linear equation system $A\underline{x} = \underline{b}$ can be solved and $\underline{x}$ is the solution to the Ritz-Galerkin approximation of the weak formulation of the boundary value problem (5.34):

$$\begin{aligned}
\underline{x} = (30 \quad & 28.6765 \quad 24.7059 \quad 18.2353 \quad 31.3235 \quad 30 \quad 25.9559 \quad 19.1176 \quad 35.2941 \quad 34.0441 \\
& 30 \quad 22.2794 \quad 41.7647 \quad 40.8824 \quad 37.7206 \quad 30 \quad 25 \quad 35 \quad 30)^T
\end{aligned}$$

Figure 5.5 shows the heat distribution within the room.

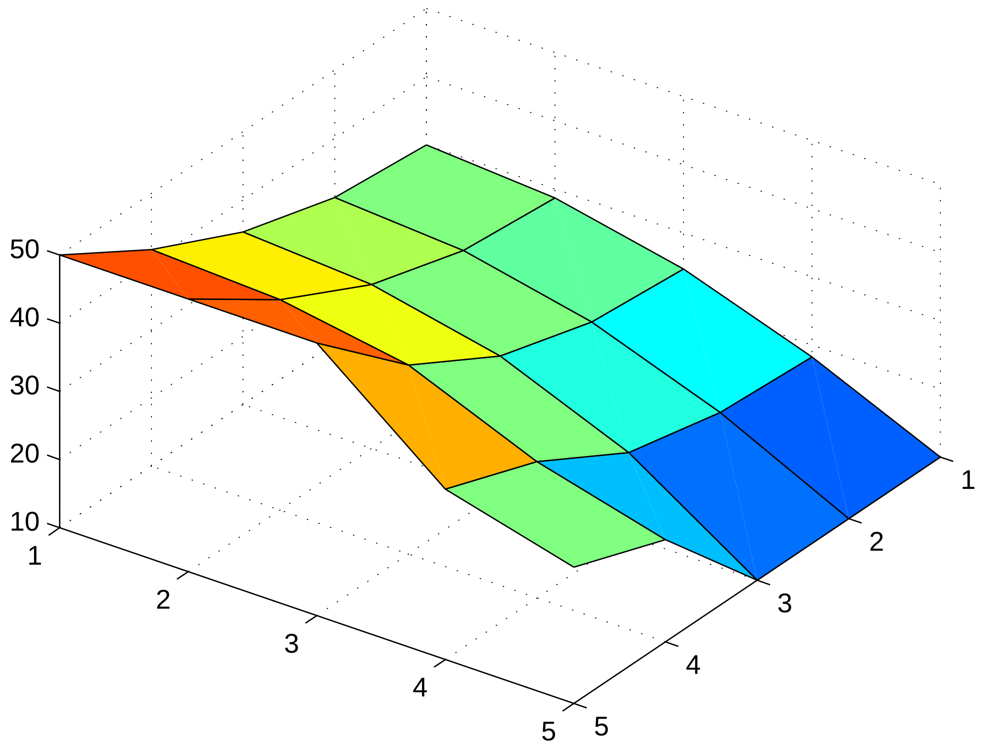


Figure 5.5: Solution of the heat distribution boundary value problem

Chapter 6

The Finite Element Method using OpenCL

One of the first analyzation of finite element method algorithms on GPUs were done by D. Göddeke, R. Strzodka and S. Turek in 2005 [24]. They used OpenGL for a solution to a boundary value problem. More reasearch work on this topic was done in [23].

In this chapter some implementations on solving a boundary value problem with mixed Dirichlet and Neumann boundary conditions using linear finite elements are discussed. In Section 6.1 the problem and the input as well as the output data are defined. Section 6.2 presents algorithms for the matrix setup. Implementations of a finite element solver for boundary value problems are discussed in Section 6.3. The presented implementations are compared and benchmarked in Section 6.4, where more results of the heat distribution boundary value problem of Chapter 5 are presented.

6.1 Definition of the Problem, Input and Output

The underlying boundary value problem is similar to the one defined in Section 5.3. The Poisson equation with mixed Dirichlet and Neumann boundary value condition is to be solved on a two-dimensional domain Ω. In contrast to the heat distribution example, any connected and bounded domain $\Omega \subset \mathbb{R}^2$ and any mixed Dirichlet and homogeneous Neumann boundary condition can be used as input data.

The implementations presented in this chapter cover the setup and solving of a boundary value problem where a triangulation is already given. Thus, the topological data structure of the system has to be defined. First an array of vertices including the type of the vertex is required. A vertex can be of type degree of freedom or of type Dirichlet boundary vertex. Then, the topological information of all triangles connected to a vertex and all vertices of a triangle are needed. At last, the values of the solution at the Dirichlet boundary vertices are required. Linear finite elements are used, hence no additional topological information such as edge data is required. The inputs are defined in Listing 6.1.

Listing 6.1: Definition of the input to the finite element method implementations

```
cl_double2 * mesh_coordinates;            // all vertices of the triangulation
cl_int * mesh_vertex_to_dof;              // maps a vertex to the index of the degree of
                                          // freedom
cl_uint4 * mesh_cells;                    // all cells of the triangulation
                                          // including the vertex indices of the triangle
cl_uint * mesh_vertex_to_cells_id;        // all cell indices,
                                          // which are connected to a vertex
cl_uint * mesh_vertex_to_cells_jumper;    // start index of the cell indices,
                                          // which are connected to a vertex
cl_uint * boundary_vertex_neighbor_ids; // all degree of freedom indices,
                                          // which are neighboring a Dirichlet boundary
                                          // vertex
cl_double * boundary_vertex_values;       // the values of the solution at the
                                          // Dirichlet boundary vertex
```

Since numerical accuracy and OpenCL compatibility is important, OpenCL vector and base types with double precision floating point types are used. *mesh_coordinates* defines an array with all vertices, degrees of freedom and boundary vertices, of the triangulation. *mesh_vertex_to_dof* is an array which maps a vertex index to the index of the degree of freedom. If the vertex is a Dirichlet boundary vertex, the array maps this vertex to the value -1. *mesh_cells* defines an array of all triangles. Each triangle is specified by 3 vertex indices. The data type *cl_double4* is used because of alignment and the last entry of the 4 dimensional type is ignored. *mesh_vertex_to_cells_id* and *mesh_vertex_to_cells_jumper* define a mapping from a vertex to all triangles connected to that vertex. Similar to the compressed matrix scheme, a jumper array is used to specify the start and stop index in the *mesh_vertex_to_cells_id* array. *boundary_vertex_neighbor_ids* defines a list of all degree of freedom indices which are neighboring a Dirichlet boundary vertex. *boundary_vertex_values* is a mapping from all vertices to the Dirichlet boundary value of that vertex. This array is only defined for vertex indices which are Dirichlet boundary vertices.

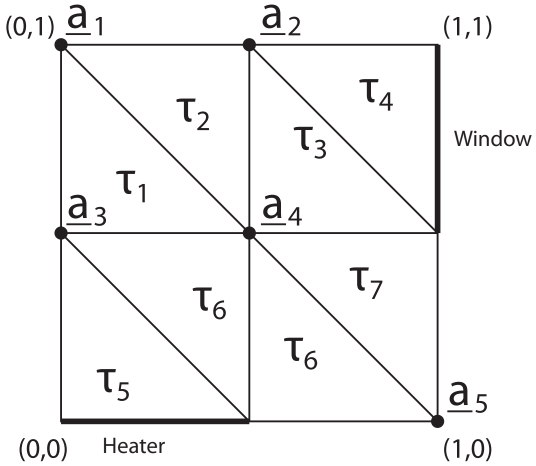


Figure 6.1: Trivial triangulation of the Poisson equation boundary value problem

Figure 6.1 shows a trivial triangulation for the problem presented in Section 5.3. For this triangulation, the arrays defined in Listing 6.1 are:

Listing 6.2: The input arrays for the trivial triangulation

```
cl_double2 * mesh_coordinates = { {0,1}, {0.5,1}, {1,1}, {0,0.5}, {0.5,0.5}, {1,0.5},
    {0,0}, {0.5,0}, {1,0} };
cl_int * mesh_vertex_to_dof = { 0, 1, -1, 2, 3, -1, -1, -1, 4 };
cl_uint4 * mesh_cells = { {0,3,4,0}, {0,4,1,0}, {1,4,5,0}, {1,5,2,0}, {4,6,7,0,
    {3,7,4,0}, {4,7,8,0}, {4,8,5,0} };
cl_uint * mesh_vertex_to_cells_id = { 0, 1, 1, 2, 3, 3, 0, 4, 5, 0, 1, 2, 5, 6, 7, 2, 3,
    7, 4, 4, 5, 6, 6, 7 };
cl_uint * mesh_vertex_to_cells_jumper = { 0, 2, 5, 6, 9, 15, 18, 19, 22, 24 };
cl_uint * boundary_vertex_neighbor_ids = { 1, 3, 4, 8 };
cl_double * boundary_vertex_values = { NAN, NAN, 10, NAN, NAN, 10, 50, 50, NAN };
```

The output of the finite element algorithms presented in this chapter is simply an array of *cl_double* types containing the values of the solution at all degree of freedom vertices. The array *mesh_vertex_to_dof* can be used to map the output array back to the vertices.

6.2 Matrix and Right Hand Side Setup

Before the matrix can be set up, some preparations are necessary. First of all, the integrals of the basis functions in the reference triangle have to be calculated. Since the triangulation is not necessarily based on a regular grid, as it has been the case for the example presented in Section 5.3, the local integrals of the basis function on an arbitrary triangle with vertices $\underline{v}_1, \underline{v}_2, \underline{v}_3$ have to be transformed from a reference triangle. Figure 6.2 shows the reference

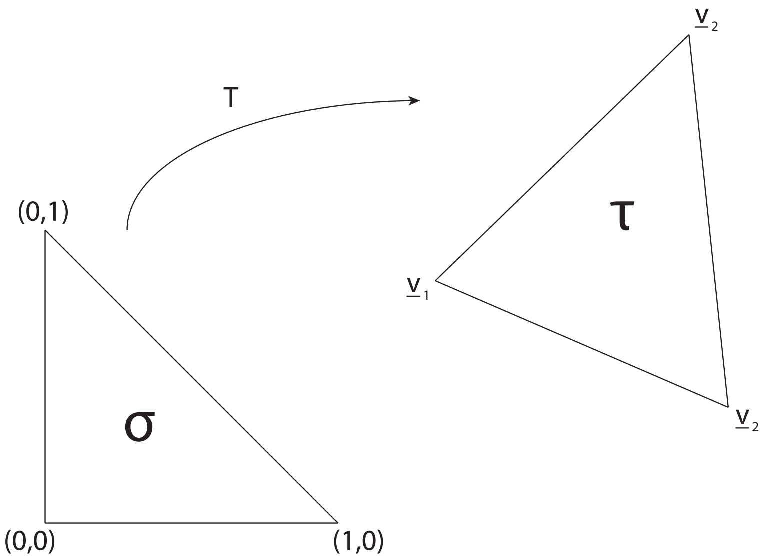


Figure 6.2: Transformation of the reference triangle

triangle and an exemplary triangle on the plane. The basis functions p_1, p_2, p_3 in the reference triangle σ are defined as presented in (5.30). Similar to the basis function in the reference triangle, the basis function q_1, q_2, q_3 in any arbitrary triangle have to be affine functions and should satisfy:

$$q_i(\underline{v}_i) = \delta_{ij}$$

Instead of defining the basis functions for each triangle τ, the basis functions for each triangle can be obtained from the basis functions in the reference triangle. A transformation T is introduced, which achieve

$$q_i(\underline{v}) = p_i(T^{-1}(\underline{v})).$$

T maps the triangle σ to the triangle τ and is affine per definition. Using the transformation T, the basis function q_i in any arbitrary triangle τ can be transformed back to the basis function $p_i(\underline{v}) := \alpha_i \underline{v}_x + \beta_i \underline{v}_y + \gamma_i$ in the reference triangle. The operator T also trivially satisfies:

$$T\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \underline{v}_1 \quad T\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \underline{v}_2 \quad T\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \underline{v}_3$$

For later calculation, the operator T is specified here in detail. The vectors $\underline{d}_2$ and $\underline{d}_3$

define the vector from $\underline{v}_1$ to $\underline{v}_2$ respectively $\underline{v}_3$.

$$\begin{aligned}
\underline{d}_2 :&= \underline{v}_2 - \underline{v}_1 = \begin{pmatrix} \underline{d}_{2,x} \\ \underline{d}_{2,y} \end{pmatrix} \\
\underline{d}_3 :&= \underline{v}_3 - \underline{v}_1 = \begin{pmatrix} \underline{d}_{3,x} \\ \underline{d}_{3,y} \end{pmatrix}
\end{aligned}$$

A matrix A is introduced which represents the linear part of the affine operator T. The inverse matrix A^{-1} is used for the inverse operator T^{-1}.

$$\begin{aligned}
A :&= \begin{pmatrix} \underline{d}_2 & \underline{d}_3 \end{pmatrix} \\
A^{-1} &= \frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}} \begin{pmatrix} \underline{d}_{3,y} & -\underline{d}_{3,x} \\ -\underline{d}_{2,y} & \underline{d}_{2,x} \end{pmatrix}
\end{aligned}$$

The explicit representations of the transformation T and its inverse T^{-1}

$$\begin{aligned}
T\left(\underline{v}\right) &= A\underline{v} + \underline{v}_1, \\
\underline{t} :&= \begin{pmatrix} \underline{t}_x \\ \underline{t}_y \end{pmatrix} = A^{-1}\underline{v}_1, \\
T^{-1}\left(\underline{v}\right) &= A^{-1}\underline{v} - \underline{t}.
\end{aligned}$$

To calculate the local integrals of the basis functions in the triangle τ, the basis function p_i have to be calculated.

$$\begin{aligned}
q_i(\underline{v}) = p_i\left(T^{-1}(\underline{v})\right) =& p_i\left(T^{-1}\begin{pmatrix} x \\ y \end{pmatrix}\right) = p_i\left(A^{-1}\underline{v}A^{-1}\underline{v_1}\right) \\
=& p_i\left(\frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}} \begin{pmatrix} \underline{d}_{3,y}x - \underline{d}_{3,x}y + \underline{t}_x \\ -\underline{d}_{2,y}x + \underline{d}_{2,x}y + \underline{t}_y \end{pmatrix}\right) \\
=& \alpha_i \frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}(\underline{d}_{3,y}x - \underline{d}_{3,x}y + \underline{t}_x) + \\
& \beta_i \frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}(-\underline{d}_{2,y}x + \underline{d}_{2,x}y + \underline{t}_y) + \gamma_i
\end{aligned}$$

Now the partial derivatives of $p_i(T^{-1})$ have to be calculated.

$$\begin{aligned}
\left(p_i\left(T^{-1}\left(\underline{v}\right)\right)\right)_x =& \frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}(\alpha_i\underline{d}_{3,y} - \beta_i\underline{d}_{2,y}) \\
\left(p_i\left(T^{-1}\left(\underline{v}\right)\right)\right)_y =& \frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}(-\alpha_i\underline{d}_{3,x} + \beta_i\underline{d}_{2,x})
\end{aligned}$$

The formula for the local integrals is derived as follows:

$$\begin{aligned}
\int_\tau \nabla q_i(\underline{v})\nabla q_j(\underline{v})d\underline{v} =& \int_\tau \nabla\left(p_i\left(T^{-1}(\underline{v})\right)\right)\nabla\left(p_j\left(T^{-1}(\underline{v})\right)\right)d\underline{v} \\
=& \int_\tau \left(p_i\left(T^{-1}(\underline{v})\right)\right)_x\left(p_j\left(T^{-1}(\underline{v})\right)\right)_x + \\
& \left(p_i\left(T^{-1}(\underline{v})\right)\right)_y\left(p_j\left(T^{-1}(\underline{v})\right)\right)_y d\underline{v} \\
=& \left(\frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}\right)^2 \int_\tau (\alpha_i\underline{d}_{3,y} - \beta_i\underline{d}_{2,y})(\alpha_j\underline{d}_{3,y} - \beta_j\underline{d}_{2,y}) + \\
& (-\alpha_i\underline{d}_{3,x} + \beta_i\underline{d}_{2,x})(-\alpha_j\underline{d}_{3,x} + \beta_j\underline{d}_{2,x})d\underline{v} \\
=& \left(\frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}\right)^2 \int_\tau \alpha_i\alpha_i(\underline{d}_{3,y}^2 + \underline{d}_{3,x}^2) + \beta_i\beta_j(\underline{d}_{2,y}^2 + \underline{d}_{2,x}^2) - \\
& \alpha_i\beta_j + \alpha_j\beta_i)(\underline{d}_{2,y}\underline{d}_{3,y} + \underline{d}_{2,x}\underline{d}_{3,x})d\underline{v} \\
=& \left(\frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}\right)^2 \int_\tau \alpha_i\alpha_j\left\langle\underline{v}_3, \underline{v}_3\right\rangle + \beta_i\beta_j\left\langle\underline{v}_2, \underline{v}_2\right\rangle - \\
& (\alpha_i\beta_j + \alpha_j\beta_i)\left\langle\underline{v}_2, \underline{v}_3\right\rangle d\underline{v} \\
=& \left(\frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}\right)^2 (\alpha_i\alpha_j\left\langle\underline{v}_3, \underline{v}_3\right\rangle + \beta_i\beta_j\left\langle\underline{v}_2, \underline{v}_2\right\rangle - \\
& (\alpha_i\beta_j + \alpha_j\beta_i)\left\langle\underline{v}_2, \underline{v}_3\right\rangle)\,\mathrm{meas}(\tau) \\
=& \frac{1}{2}\left(\frac{1}{\underline{d}_{2,x}\underline{d}_{3,y} - \underline{d}_{2,y}\underline{d}_{3,x}}\right)(\alpha_i\alpha_j\left\langle\underline{v}_3, \underline{v}_3\right\rangle + \beta_i\beta_j\left\langle\underline{v}_2, \underline{v}_2\right\rangle - \\
& (\alpha_i\beta_j + \alpha_j\beta_i)\left\langle\underline{v}_2, \underline{v}_3\right\rangle)
\end{aligned}$$

With this preparation, the local basis integrals can be calculated for $i, j = 1, 2, 3$:

$$\begin{aligned}
\psi_{i,j} &:= \int_\tau \frac{\partial \phi_i}{\partial x} \times \frac{\partial \phi_j}{\partial x} d\underline{x} = \frac{\alpha_i \alpha_j}{2} \\
\xi_{i,j} &:= \int_\tau \frac{\partial \phi_i}{\partial x} \times \frac{\partial \phi_j}{\partial y} + \frac{\partial \phi_i}{\partial y} \times \frac{\partial \phi_j}{\partial x} d\underline{x} = \frac{\alpha_i \beta_j + \alpha_j \beta_i}{2} \\
\zeta_{i,j} &:= \int_\tau \frac{\partial \phi_i}{\partial y} \times \frac{\partial \phi_j}{\partial y} d\underline{x} = \frac{\beta_i \beta_j}{2}
\end{aligned}$$

With these elements, the basis integrals on an arbitrary triangle τ with the vertices $\underline{v}_1, \underline{v}_2, \underline{v}_3$ can be calculated as follows:

$$\begin{aligned}
\theta_{\tau,i,j} &: = \int_\tau \nabla \phi_i \nabla \phi_j dx \\
&= \frac{\langle \underline{d}_2, \underline{d}_2 \rangle \times \psi_{i,j} - \langle \underline{d}_2, \underline{d}_3 \rangle \times \xi_{i,j} + \langle \underline{d}_3, \underline{d}_3 \rangle \times \zeta_{i,j}}{\det \begin{pmatrix} \underline{d}_2 & \underline{d}_3 \end{pmatrix}}
\end{aligned}$$

The matrix entries are then given by

$$\begin{aligned}
A_{ij} &= a(\phi_i, \phi_j) \\
&= \sum_{\tau \in T_h(\underline{v}_i) \cap T_h(\underline{v}_j)} \int_\tau \nabla \phi_i \nabla \phi_j dx \\
&= \sum_{\tau \in T_h(\underline{v}_i) \cap T_h(\underline{v}_j)} \theta_{\tau,i,j}
\end{aligned}$$

An array *cell_integrals* including all $\theta_{\tau,i,j}, \tau \in T_h, i, j = 1, 2, 3$ is generated before the matrix is actually set up. With this array the algorithm presented in Listing 6.3 generates the matrix A. Figure 6.3 gives a graphical overview of this algorithm.

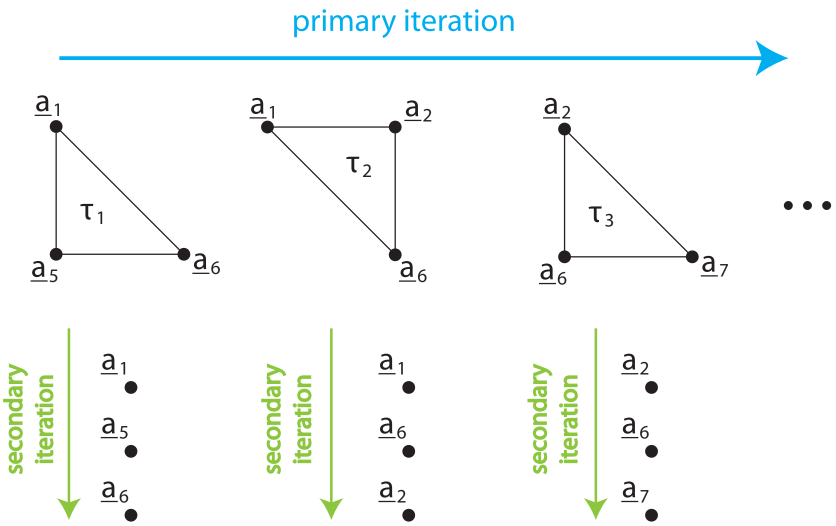


Figure 6.3: Assembly algorithm with primary iteration over all cells

Listing 6.3: Matrix setup pseudo code

```
A = 0

for each triangle T in triangulation
    if (T.index_1 != -1)
        A[T.vertex_1, T.vertex_1] += cell_integrals[T,1,1]
        if (T.index_2 != -1) A[T.vertex_1, T.vertex_2] += cell_integrals[T,1,2]
        if (T.index_3 != -1) A[T.vertex_1, T.vertex_3] += cell_integrals[T,1,3]
    end

    if (T.index_2 != -1)
        if (T.index_1 != -1) A[T.vertex_2, T.vertex_1] += cell_integrals[T,2,1]
        A[T.vertex_2, T.vertex_2] += cell_integrals[T,2,2]
        if (T.index_3 != -1) A[T.vertex_2, T.vertex_3] += cell_integrals[T,2,3]
    end

    if (T.index_3 != -1)
        if (T.index_1 != -1) A[T.vertex_3, T.vertex_1] += cell_integrals[T,3,1]
        if (T.index_2 != -1) A[T.vertex_3, T.vertex_2] += cell_integrals[T,3,2]
        A[T.vertex_3, T.vertex_3] += cell_integrals[T,3,3]
    end
end
```

Due to the fact, that different outer loop iterations may access the same element in the matrix data structure A, the algorithm presented in Listing 6.3 is difficult to parallelize. A race condition with the vertices $\underline{a}_1$ and $\underline{a}_2$ can be seen in Figure 6.3. An alternative algorithm which does not have this disadvantage is presented in Listing 6.4. This implementation iterates over all degrees of freedom vertices instead of iterating over all triangles as shown in Figure 6.4. Since each degree of freedom vertex represents one row in the matrix, the outer loop can be executed in parallel.

Listing 6.4: Matrix setup pseudo code with Iteration over vertices

```
A = 0

for each degree of freedom vertex V in triangulation

    for each triangle T connected to V
        if (T.index_1 == V)
            if (T.index_1 != -1) A[V, T.vertex_1] += cell_integrals[T,1,1]
            if (T.index_2 != -1) A[V, T.vertex_2] += cell_integrals[T,1,2]
            if (T.index_3 != -1) A[V, T.vertex_3] += cell_integrals[T,1,3]
        end

        if (T.index_2 == V)
            if (T.index_1 != -1) A[V, T.vertex_1] += cell_integrals[T,2,1]
            if (T.index_2 != -1) A[V, T.vertex_2] += cell_integrals[T,2,2]
            if (T.index_3 != -1) A[V, T.vertex_3] += cell_integrals[T,2,3]
        end

        if (T.index_3 == V)
            if (T.index_1 != -1) A[V, T.vertex_1] += cell_integrals[T,3,1]
            if (T.index_2 != -1) A[V, T.vertex_2] += cell_integrals[T,3,2]
            if (T.index_3 != -1) A[V, T.vertex_3] += cell_integrals[T,3,3]
        end
    end

end
```

Since both algorithms make heavy use of random accesses in the matrix, the matrix types presented in Chapter 3.1 are not suitable for these algorithms. A dense matrix has constant random access complexity, but requires a lot of memory. A binary search tree with the location of the matrix entry as index is a good choice for a data type in these algorithms. Nevertheless, the data should be converted to a storage scheme which provides a fast matrix-vector multiplication after the matrix setup is done.

The algorithms for generating the right hand side vector are similar. When using the algorithm prototype presented in Listing 6.4, the outer loop iterates over all degree of freedom vertices neighboring a Dirichlet boundary vertex instead of iterating over all vertices. This version of the algorithm can also be parallelized.

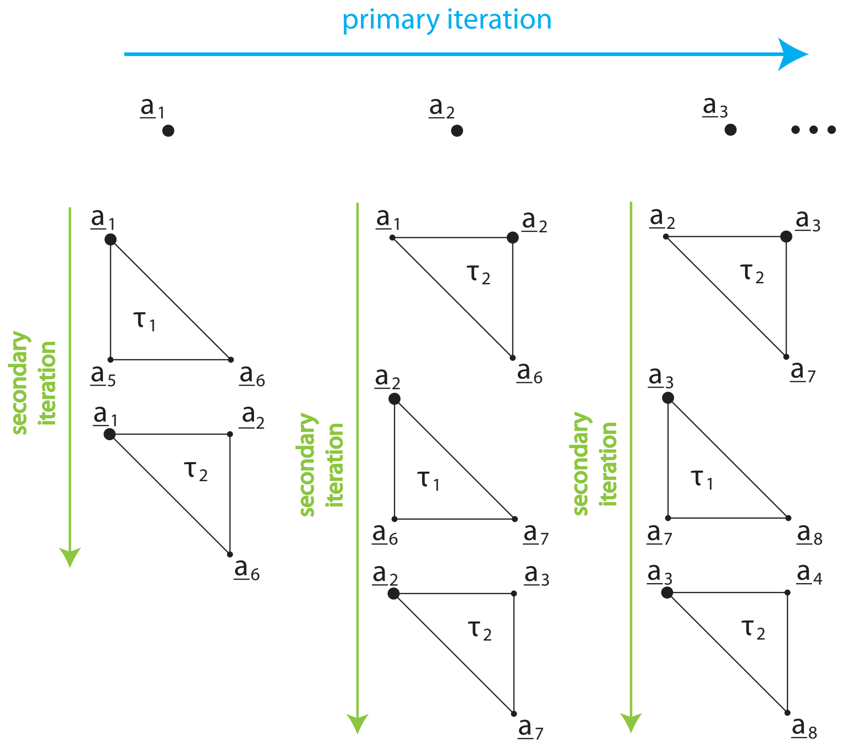


Figure 6.4: Assembly algorithm with primary iteration over all vertices

6.3 Solving the Boundary Value Problem

With the algorithms presented in the previous section an implementation using the CPU can be created. At first the local integrals have to be computed. Then the system matrix and right hand side vector can be set up. The matrix is converted to a storage scheme with fast matrix-vector multiplication and the system is solved using the iterative CG algorithm. The first implementation makes use of the CPU only.

As presented in Chapter 4, ViennaCL provides an implementation of the CG algorithm. To take advantage of that fact, the solution of the system can be accomplished on an OpenCL device. To achieve that, the data has to be converted to ViennaCL types. In this case the solving of the equation system is entirely done on the OpenCL device. The result vector has to be transferred back to the CPU after solving. Although the algorithm presented in Listing 6.4 can be executed in parallel, the matrix setup process cannot be converted easily to OpenCL due to restrictions in data management in OpenCL kernels. The calculation of the right hand side vector does not require complex data management and can therefore be done with OpenCL.

Listing 6.5: Direct FEM operator in OpenCL

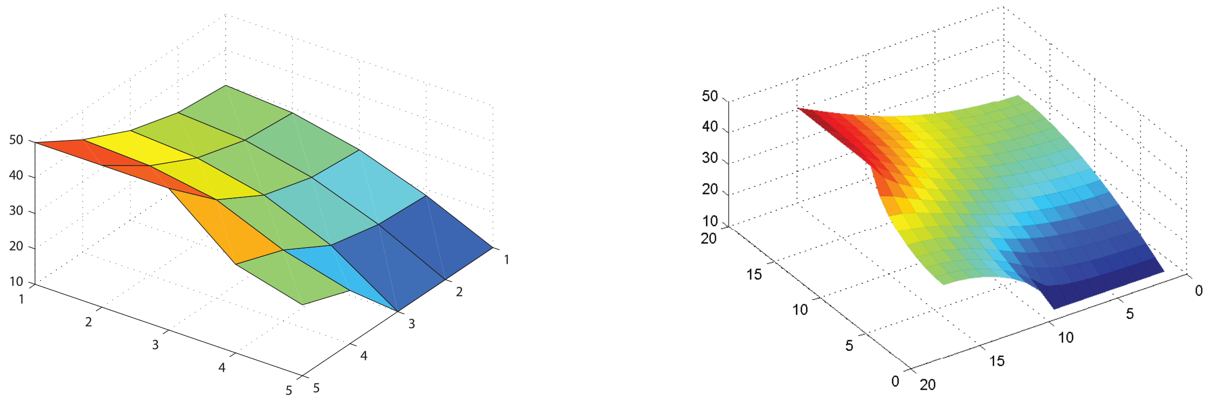
```
__kernel void FEMOperator(
    unsigned int mesh_vertex_num, // the number of vertices
    __global const int * mesh_vertex_to_dof, // degrees of freedom id
                                                   // (-1 for Dirichlet boundary vertex)
    unsigned int mesh_cell_num, // number of cells
    __global const uint4 * mesh_cells, // cells with vertex ids
    __global const double4 * cell_integrals_alpha, // the local integrals
    __global const double4 * cell_integrals_beta,
    __global const double4 * cell_integrals_gamma,
    __global const unsigned int * mesh_vertex_to_cells_id,  // vertex to cells mapping
    __global const unsigned int * mesh_vertex_to_cells_jumper,
    __global double * result, // the result vector vector
    __global const double * rhs) // the right hand side vector
{
    // iterate over all vertices of the mesh
    for (unsigned int vertex_id = get_global_id(0);
        vertex_id < mesh_vertex_num;
        vertex_id += get_global_size(0))
    {
        int dof_id = mesh_vertex_to_dof[vertex_id];
        if (dof_id < 0) continue; // non-degree of freedom vertices are ignored
        double tmp = 0;
        // iterate over all cells, which are connected to the current vertex
        unsigned int index_stop = mesh_vertex_to_cells_jumper[vertex_id+1];
        for (unsigned int index = mesh_vertex_to_cells_jumper[vertex_id];
            index < index_stop;
            ++index)
        {
            unsigned int cell_id = mesh_vertex_to_cells_id[index];
            uint4 cell_vertices = mesh_cells[cell_id];
            // distinction of cases: which of the three cell vertices is the current
                vertex
            if (cell_vertices.s0 == vertex_id)
            {
                int local_vertex_dof_1 = mesh_vertex_to_dof[cell_vertices.s1];
                int local_vertex_dof_2 = mesh_vertex_to_dof[cell_vertices.s2];
                double4 cell_integrals = cell_integrals_alpha[cell_id];
                // perform the operator-vector multiplication for all degree of freedom
                    neighbor vertices on the current triangle
                tmp += rhs[dof_id] * cell_integrals.s0;
                if (local_vertex_dof_1 >= 0)
                    tmp += rhs[local_vertex_dof_1] * cell_integrals.s1;
                if (local_vertex_dof_2 >= 0)
                    tmp += rhs[local_vertex_dof_2] * cell_integrals.s2;
            }
            // the cases cell_vertices.s1 == vertex_id and cell_vertices.s2 == vertex_id
                are similar
        }
        result[dof_id] = tmp;
    }
}
```

The CG algorithm does not necessarily require a matrix, it only needs an operator which maps a vector to another vector based on the linear system. Therefore, it is not required to set up the system matrix explicitly. Such an operator can be provided by using the algorithm presented in Listing 6.4. Instead of setting up the matrix explicitly, a linear multiplication with a vector is performed. This can be achieved by adding up all local cell integral values multiplied by the corresponding value in the right hand side vector. Since this algorithm is derived from the algorithm presented in Listing 6.4, it can be executed in parallel and is therefore appropriate for OpenCL. Although the complexity of this algorithm is the same as the complexity of a matrix-vector multiplication, it will be slightly slower because the direct finite element operator requires more operations. An OpenCL implementation of this algorithm is presented in Listing 6.5. The *cell_integrals_** pointers define the precomputed local cell integral values. *cl_double4* is used because of alignment. When using this operator, the local cell integrals can also be computed using OpenCL.

6.4 Results, Benchmarks and Comparison

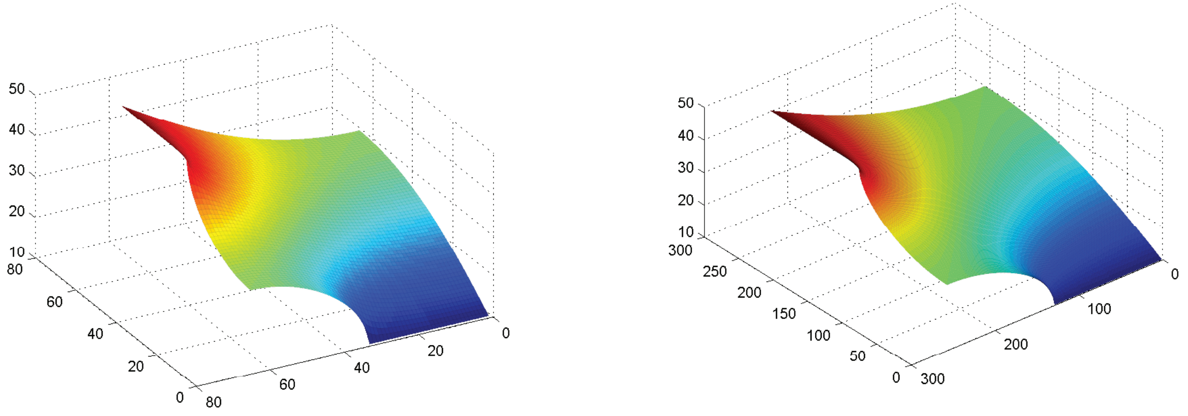
The accuracy of solution to the boundary value problem presented in Section 5.3 can simply be increased by increasing the number of degree of freedom vertices. All results given in this section are calculated by using a triangulation based on a regular grid consisting of squares. Some solutions to the boundary value problem with different discretization levels are presented in Figure 6.5.

The benchmark setup in this section is the same as in Section 3.3. The only difference is that only the NVIDIA GeForce GTX 470 device is used for benchmarking all OpenCL implementations. The parts stated in Table 6.1 were all benchmarked separately. The benchmark results are given in Table 6.2 for the setup routines, in Table 6.4 for solving the linear system and in Table 6.5 for the whole process. The linear dependency of the setup computation times can be observed for large numbers of cells. An increase by a factor of four in the number of cells leads to a factor of about four in the computation times for all setup routines. This behavior is not present for the solving algorithms. The larger the linear system is, the more iterations are needed by the CG algorithm for achieving the same accuracy. Additionally, the matrix-vector multiplication complexity and linear FEM operator multiplication complexity is linearly bounded in the number of cells. This results in a solver complexity which is above linear. Table 6.3 presents the benchmark results for one single matrix-vector



Solution with 5x5 Vertices

Solution with 17x17 Vertices



Solution with 65x65 Vertices

Solution with 257x257 Vertices

Figure 6.5: Solution of the heat distribution boundary value problem with different discretization levels

Shortcut	Algorithm-part
setup_li	Computing the local cell basis integrals using the CPU.
setup_mv	Computing the matrix values in a binary search tree data structure using the CPU.
convert_cm	Converting the binary search tree data structure to a compressed matrix data structure using the CPU.
vcl_convert_cm	Converting the binary search tree data structure to a *viennacl::compressed_matrix* with alignment of 4 on the OpenCL device.
setup_femop	Creating the direct linear FEM operator, including data transfers to the OpenCL device.
setup_rhs	Computing the entries for the right hand side vector using the CPU.
vcl_convert_rhs	Converting the right hand side vector to a *viennacl::vector* on the OpenCL device.
solve_cm	Solving the system of linear equations with the compressed matrix data structure on the CPU.
solve_vcl_cm	Solving the system of linear equations with the *viennacl::compressed_matrix* using the OpenCL device.
solve_femop	Solving the system using the direct linear FEM operator on the OpenCL device.
rdb	Reading back the result vector from the OpenCL device to the CPU.

Table 6.1: Parts of the algorithms which were benchmarked

multiplication with a compressed matrix data structure, *viennacl::compressed_matrix* and one linear FEM operator multiplication. The FEM operator is about 2 times slower than *viennacl::compressed_matrix* for large problem sizes. A comparison of the benchmark results of one matrix-vector multiplication versus the whole CG algorithm can be seen in Figure 6.6 and Figure 6.7.

It can easily be seen, that the setup computation time is relatively small compared to the solution of the linear system. Therefore, the performance of the whole FEM algorithm is highly dependent on the performance of the CG implementation as can be seen in Figure 6.7 and Figure 6.8. The best solver times are achieved by ViennaCL linear algebra types. With an asymptotic factor of two, the direct FEM operator is the second best solving implementation. The asymptotic factor of two is a result of the relative difference of two when comparing the matrix-vector multiplication with *viennacl::compressed_matrix* with the direct FEM operator multiplication. For small systems the ViennaCL overhead is relatively large compared to the actual computation time. Since the linear operator multiplication is

Number of cells	setup_li	setup_mv	convert_cm	vcl_convert_cm	setup_femop	setup_rhs	vcl_convert_rhs
2048	0.16	0.655	0.089	1.303	0.137	0.011	0.155
8192	0.44	2.454	0.521	2.117	0.442	0.028	0.193
32768	2.586	9.885	2.272	5.708	1.927	0.078	0.286
131072	14.308	39.222	7.425	21.21	8.743	0.252	0.573
524288	52.132	153.05	33.04	83.18	35.067	0.851	1.645
2097152	210.94	606.62	132.53	329.95	141.64	3.673	6.729

Table 6.2: Benchmark result in seconds for the setup algorithm parts

Number of cells	Compressed matrix	viennacl::compressed_matrix	direct FEM operator
2048	0.000010	0.000021	0.000033
8192	0.000022	0.000030	0.000046
32768	0.000061	0.000070	0.000141
131072	0.000339	0.000236	0.000471
524288	0.002397	0.000923	0.001875
2097152	0.009523	0.003802	0.007916

Table 6.3: Benchmark result in seconds for one matrix-vector multiplication

the main part of the CG algorithm from the performance point of view and there is some overhead using ViennaCL, the relative difference in performance of the whole finite element algorithm is smaller for a lower number of cells. The implementation on the CPU has the worst performance. Due to implicit OpenCL calls, the benchmark times are unstable for small systems. Therefore, the direct OpenCL operator might be faster than the method using *viennacl::compressed_matrix*.

Since the direct FEM operator algorithm for linear finite elements does not require much more operations than a simple sparse matrix-vector multiplication, the algorithm is memory bandwidth limited. Compared to a compressed matrix-vector multiplication, the direct FEM operator needs to read more memory. Therefore, the direct FEM operator multiplication is slower than the *viennacl::compressed_matrix* matrix-vector multiplication. A different result may be achieved when using a higher-order finite element basis. With higher-order finite elements, the total memory transferred may be smaller when using a direct FEM operator instead of using *viennacl::compressed_matrix*. In this case, the local integrals have to be computed within the OpenCL kernel. Since the direct FEM operator is memory bandwidth limited, the additional operations may not affect the performance.

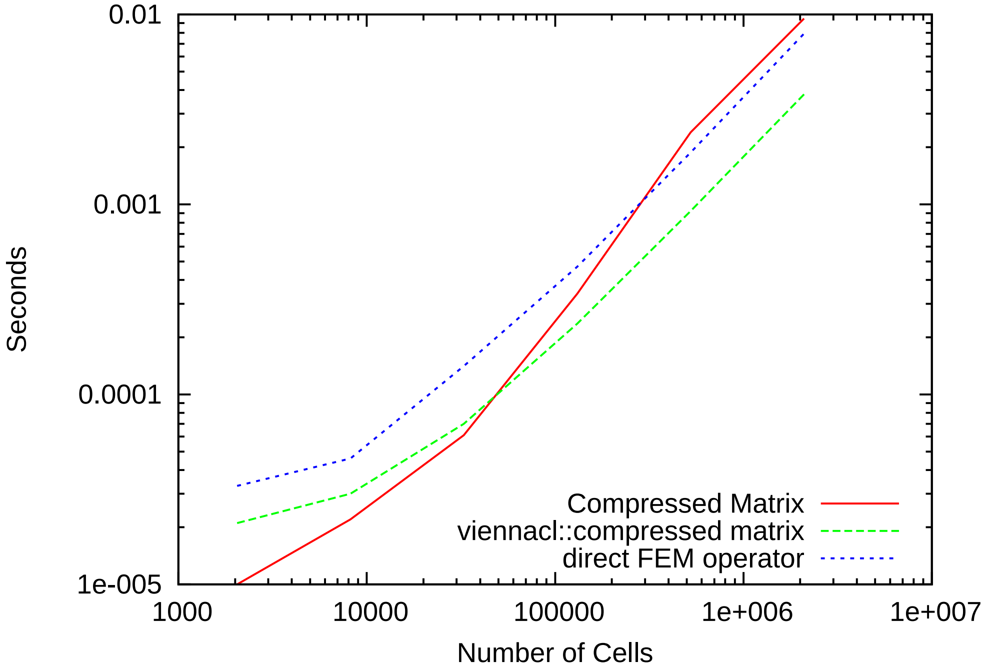


Figure 6.6: Benchmark result in seconds for one matrix-vector multiplication

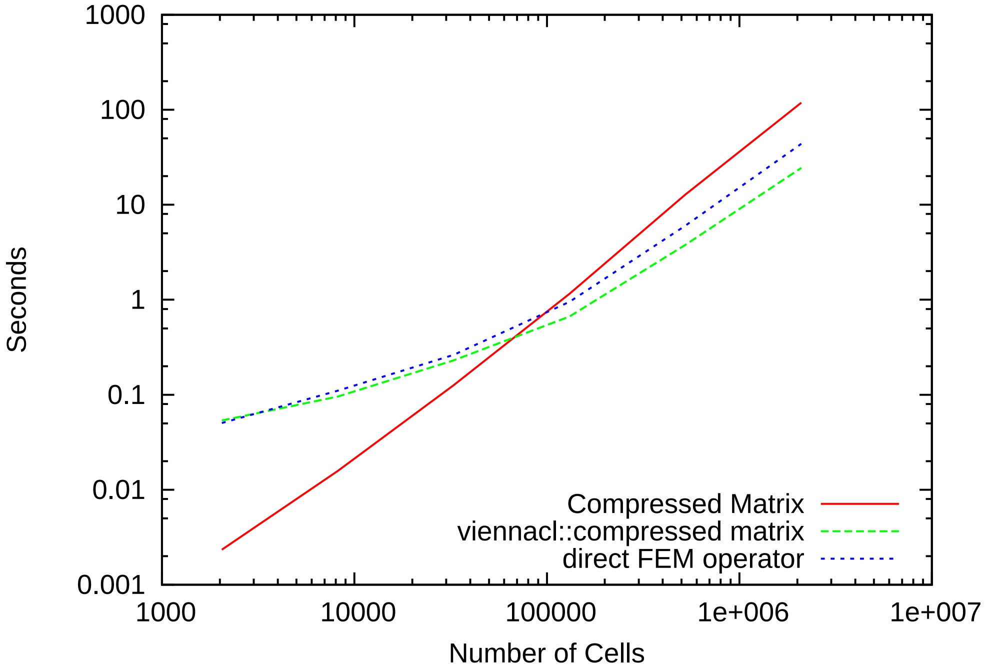


Figure 6.7: Benchmark result in seconds the CG algorithm

Number of cells	solve_cm	solve_vcl_cm	solve_femop	rdb
2048	0.00234	0.05360	0.05042	0.000126
8192	0.01578	0.09583	0.11026	0.000118
32768	0.12575	0.23065	0.26229	0.000129
131072	1.15548	0.66637	0.95301	0.000473
524288	12.83190	3.77916	6.03665	0.001641
2097152	118.72100	24.48980	43.78280	0.007695

Table 6.4: Benchmark result in seconds for the CG solving algorithm parts

Number of cells	Setup and solution is done on the CPU	Setup on the CPU, solution on the GPU using ViennaCL linear algebra	Preparation on the CPU, solution on the GPU using the direct FEM operator
2048	0.0033	0.0560	0.0510
8192	0.0192	0.1012	0.1115
32768	0.1406	0.2493	0.2673
131072	1.2167	0.7424	0.9774
524288	13.0710	4.0717	6.1280
2097152	119.6748	25.6554	44.1535

Table 6.5: Benchmark result in seconds for whole finite element algorithm

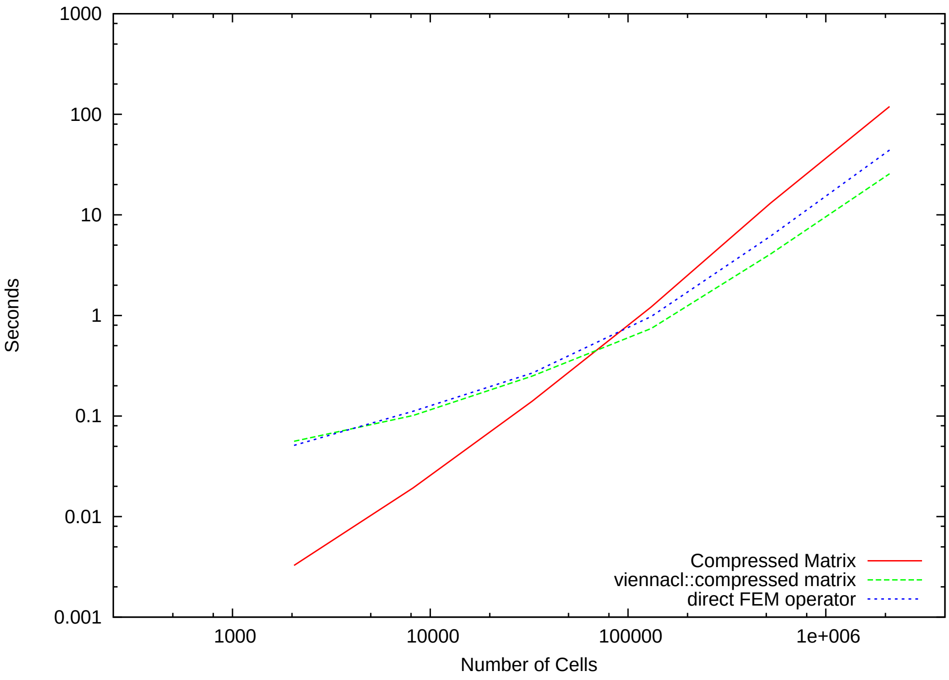


Figure 6.8: Benchmark result in seconds for whole finite element algorithm

Chapter 7

Conclusion

The results presented in Section 3.3 imply a great potential of graphics adapters for parallel algorithms and large amounts of data. Without optimization, the matrix-vector multiplication on NVIDIA GeForce GTX 470 is about 35% faster than the implementation on the CPU for 67108864 non-zero elements. With more effort in optimization, especially hardware-dependent optimization, graphics adapters might be better than CPUs in these cases.

As discussed in Chapter 6, graphics adapters are also a good choice for the whole finite element algorithm. The solution of the linear system using ViennaCL is by a factor of 4.6 and the presented direct FEM operator is by a factor of 2.7 faster than the algorithm using the CPU. The price of both hardware used for benchmarking, the Intel Core i7-960 and the NVIDIA GeForce GTX 470, is comparable, hence the graphics adapter is the preferable platform.

7.1 Outlook

Finite element algorithms using graphics adapters have already been investigated for example by D. Göddeke in 2005, yet the topic is relatively new and more research work has to be done [23] [24]. Some possible future topics are presented here.

For matrix data structures, other schemes could be used for the finite element method. NVIDIA presented specialized matrix data types for GPU computing to gain more performance in matrix-vector multiplication [22].

The presented finite element algorithms with OpenCL can be optimized by transferring some steps of the setup process to OpenCL. The calculation of the local integrals and the setup of the right hand side vector can be parallelized easily and, therefore, be computed using OpenCL. Due to data management restriction with OpenCL, it is not possible to port the classical algorithm for assembling the mesh data to OpenCL. A sparse matrix data structure can be defined as a modification of the coordinate or compressed scheme, where each entry in the matrix is represented by the sum of all entries with the same row and column indices in the matrix data structure. Instead of summing up all local element integral values to one element in the system matrix, each local element integral contribution is stored on its own. With this modification to the matrix setup, two threads do not make write accesses to the same element in the system matrix at one time and the algorithm can, therefore, be executed in parallel.

As mentioned in Section 6.4, bandwidth limitation are less severe when using a direct FEM operator with higher-order finite elements. With higher-order finite elements and less pre-processing, the amount of floating point operations increases, which might lead to better performance on GPUs.

ViennaCL provides a generic implementation of the conjugate gradient algorithm. This implementation works with uBLAS and ViennaCL matrix and vector types. But this flexibility generates unnecessary overhead due to data transfer between CPU and GPU and superfluous API calls. A customized conjugate gradient implementation with custimized OpenCL kernels might get better performance results for small systems where kernel launch overheads are an issue.

Preconditioners using parallel computing architectures for the banded matrix scheme are discussed for example by D. Göddeke in [23]. Most preconditioners are hard to parallelize and often the matrix structure is destroyed in the process of creating the preconditioner [27] [28] [29]. Using topological mesh information, some preconditioner algorithms can be modified to a multi-threaded version. Additionally, preconditioner algorithms can be adapted for direct FEM operators.

For large systems, the computations could be split up into work packages which are processed by multiple devices. For example, the CPU can perform some calculations while the OpenCL device processes some other work package. This requires tricky synchronization and load balancing but might also lead to some performance improvements.

In this work, the tessellation of the domain Ω of a boundary value problem is not covered. Porting this step to OpenCL might increase the performance of the whole algorithm due to a reduction of memory transfer. If this step is accomplished on an OpenCL device, additional topological data is available in device memory, which can be used for multi-grid solvers or related techniques.

Appendix A

Function Spaces

The function spaces used for the finite element method are presented in the following [15].

Definition 11 (Space of coninuously differentiable functions)**.**

$$\mathcal{C}^k(\Omega) := \{f : \Omega \to \mathbb{R}, f \text{ is k-times continuously differentiable}\}, \quad k \geq 0 \tag{A.1}$$

Definition 12 (Space of piecewise differentiable line segments)**.**

$$\mathcal{C}^{0,1} := \{f : \mathbb{R} \to \mathbb{R}^k, f \text{ is continuous and piecewise differentiable}\} \subset \mathcal{P}(\mathbb{R}^k), \quad k \geq 1 \tag{A.2}$$

Definition 13 (Essential supremum)**.**

$$\operatorname{esssup}_\Omega u := \inf\{K \in \mathbb{R} : u(x) \leq K \text{ for mostly all } x \in \Omega\} \tag{A.3}$$

A function u is called essentially bounded if $\operatorname{esssup} |x| \leq \infty$*.*

Definition 14 (Lebesgue space)**.** *Let* $\Omega \subset \mathbb{R}^d, d \geq 1$*. The Lebesgue spaces are defined as follows:*

$$L^p(\Omega) = \{u : \Omega \to \mathbb{R} \text{ is measurable}, |u|^p \text{ is Lebesgue-integrable}\}, \quad 1 \leq p < \infty \tag{A.4}$$

$$L^\infty(\Omega) = \{u : \Omega \to \mathbb{R} \text{ is measurable}, u \text{ is essential bounded}\} \tag{A.5}$$

Definition 15 (Lebesgue norm)**.**

$$\|u\|_{L^p} := \left(\int_\Omega |u(x)|^p dx\right)^{\frac{1}{p}}, \quad 1 \leq p < \infty \tag{A.6}$$

$$\|u\|_{L^\infty} := \operatorname{esssup}_\Omega |u| \tag{A.7}$$

The Lebesgue space $L^p(\Omega)$ together with the Lebesgue norm is a Banach space. The space $L^2(\Omega)$ with the scalar product $(u, v)_{L^2} := \int_\Omega u(x)v(x)dx$ is a Hilbert-space.

Definition 16 (Support of a function, test function)**.** *The support of a function is defined as follows:*

$$\operatorname{supp} u := \overline{\{x \in \Omega : u(x) \neq 0\}} \tag{A.8}$$

A test function is a infinitly differentiable function with compact support:

$$C_0^\infty := \{u \in \mathcal{C}^\infty(\Omega) : \operatorname{supp} u \text{ is compact}\} \tag{A.9}$$

Definition 17 (Sobolev space)**.** *Let $m \in \mathbb{N}, 1 \leq p \leq \infty$. The space*

$$W^{m,p}(\Omega) = \{u \in L^p(\Omega) : D^\alpha u \in L^p(\Omega) \forall |\alpha| \leq m\} \tag{A.10}$$

is called Sobolev space.

Lemma 5.

1. *The Sobolev space $W^{m,p}(\Omega)$ with norm*

$$\|u\|_{W^{m,p}} := \left(\sum_{|\alpha| \leq m} \|D^\alpha u\|_{L^p}^p\right)^{\frac{1}{p}}, \quad 1 \leq p < \infty \tag{A.11}$$

$$\|u\|_{W^{m,\infty}} := \sum_{|\alpha| \leq m} \|D^\alpha u\|_{L^\infty} \tag{A.12}$$

   is a Banach space.

2. *The Sobolev space $H^m(\Omega) := W^{m,2}(\Omega)$ with the scalar product*

$$(u, v)_{H^m} := \sum_{|\alpha| \leq m} (D^\alpha u, D^\alpha v)_{L^2} \tag{A.13}$$

   is a Hilbert space.

A proof is given in [15].

Definition 18.

$$W_0^{m,p}(\Omega) \text{ is the completion of } C_0^\infty \text{ using the norm } \|\cdot\|_{W^{m,p}} \tag{A.14}$$

$$H_0^m(\Omega) := W_0^{m,2}(\Omega) \tag{A.15}$$

Lemma 6. *The space* $W_0^{m,p}(\Omega)$ *with norm* $\|\cdot\|_{W^{m,p}}$ *is a Banach space and* $H_0^m(\Omega)$ *with the scalar product* $(\cdot,\cdot)_{H^m}$ *is a Hilbert space.*

Lemma 7. *Let* $\Omega \subset \mathbb{R}^d$ *be an open domain.*

1. Ω is bounded $\wedge\, \partial\Omega \in \mathcal{C}^{0,1} \Rightarrow \mathcal{C}^\infty(\overline{\Omega})$ is dense in $W^{m,p}(\Omega)$
2. $C^\infty(\Omega) \cap W^{m,p}(\Omega)$ is dense in $W^{m,p}(\Omega)$

A proof is provided in [16].

Definition 19. *Let* $\Omega \subset \mathbb{R}^d$ *be an open domain.*

$$H^{-k}(\Omega) := \{F : H_0^k(\Omega) \to \mathbb{R} : F \text{ is linear and continuous}\} \tag{A.16}$$

$$\|F\|_{H^{-k}} := \sup_{\|u\|_{H_0^k(\Omega)}=1} |F(u)| \tag{A.17}$$

H^{-k} *is the algebraic dual space of* $H_0^k(\Omega)$ *and* $\|\cdot\|_{H^{-k}}$ *is a norm. The dual space with this norm is a Banach space.*

Lemma 8 (Cauchy-Schwarz inequality)**.** *If* H *is a linear space with an inner product* $<\cdot,\cdot>$, *then*

$$|\langle x,y\rangle| \leq \|x\|\|y\|, \qquad \forall x,y \in H. \tag{A.18}$$

Lemma 9 (Poincare inequality)**.** *If* $\Omega \subset \mathbb{R}^d, d \geq 1)$ *be a bounded domain with* $\partial\Omega \in \mathcal{C}^{0,1}$, *then there exists a* $C_p > 0$ *such that*

$$\|u\|_{L^2} \leq C_p\|\nabla u\|_{L^2}, \qquad \forall u \in H_0^1(\Omega). \tag{A.19}$$

The Poincare inequality ensures the equivalence of the norm $\|\cdot\|_{H^1}$ *and* $\|\nabla(\cdot)\|_{L^2}$.

Appendix B

The Conjugate Gradient Method

The conjugate gradient method is an algorithm for solving symmetric and positive definite systems of linear equations $A\underline{x} = \underline{b}$, with $A \in \mathbb{R}^{m \times m}$ and $\underline{x}, \underline{b} \in \mathbb{R}^m$ [13]. Although the method is an iterative algorithm, the precise solution is given at most m iterations. The conjugate gradient method transforms the system of linear equations to the quadratic form $E(x) := \frac{1}{2} \langle A\underline{x}, \underline{x} \rangle - \langle \underline{b}, \underline{x} \rangle$ and minimizes this form.

Definition 20 (The Conjugate Gradient Method)**.** *Choose an arbitrary* $\underline{x}_0 \in \mathbb{R}^m$. *The initial residuum and search direction are calculated by*

$$\underline{r}_0 := \underline{b} - A\underline{x}_0, \tag{B.1}$$

$$\underline{d}_0 := \underline{r}_0. \tag{B.2}$$

In each iteration step, the quadratic form E *is minimized using the approximation* x_k *and the search direction* d_k *of the previous iteration:*

$$\alpha_k = \frac{\underline{d}_k^T \underline{r}_k}{\underline{d}_k^T A \underline{d}_k} \tag{B.3}$$

$$\underline{x}_{k+1} := \underline{x}_k + \alpha_k \underline{d}_k \tag{B.4}$$

$$\underline{r}_{k+1} := \underline{r}_k - \alpha_k A \underline{d}_k \tag{B.5}$$

Finally, the search direction d_k *is updated:*

$$\beta_k := \frac{\underline{r}_{k+1}^T \underline{r}_{k+1}}{\underline{r}_k^T \underline{r}_k} \tag{B.6}$$

$$\underline{d}_{k+1} := \underline{r}_{k+1} + \beta_k \underline{d}_k \tag{B.7}$$

The iteration is stopped if the norm of the residuum $\|r_{k+1}\|$ *is smaller than a specified tolerance.*

Bibliography

[1] *Cramming more components onto integrated circuits*, G. E. Moore, Electronics Magazine p. 4. Retrieved 2006-11-11, 1965

[2] *GPU Architectures: Implications & Trends*, D. Luebke, NVIDIA Research, SIGGRAPH 2008, `http//s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf`

[3] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, NVIDIA Cooperation 2009, `ftp://download.intel.com/museum/Moores_Law/Articles-press_Releases/Gordon_Moore_1965_Article.pdf`

[4] *Computer Vision Signal Processing on Graphics Processing Units*, J. Fung, S. Mann, ICASSP 2004

[5] *OpenCL Programming Guide for the CUDA Architecture*, NVIDIA Cooperation, Version 4.1, 2012, `http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf`

[6] *The OpenCL Specification*, Khronos OpenCL Working Group, Version 1.2, Nov 2011, `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`

[7] *OpenCL Optimization Webinar*, NVIDIA Cooperation, 2009, `http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf`

[8] *TESLA C2050 / C2070 GPU Computing Processor*, NVIDIA Cooperation, 2010, `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`

[9] *NVIDIAs Next Generation CUDA Compute Architecture: Fermi*, NVIDIA Cooperation, Version 1.1, 2009, `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`

[10] *NVIDIA GeForce 470 specification*, NVIDIA Cooperation, `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-470/specifications`

[11] *Intel Core i7-900 Desktop Processor Extreme Edition Series and Intel Core i7-900 Desktop Processor Series*, Intel Cooperation, 2010, `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/core-i7-900-ee-and-desktop-processor-series-datasheet-vol-1.pdf`, `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/core-i7-900-ee-and-desktop-processor-series-datasheet-vol-2.pdf`

[12] *Intel Core i7-960 Processor specification*, Intel Cooperation, 2010, `http://ark.intel.com/products/37151`

[13] *Finite Element Solution of Boundary Value Problems*, O. Axelsson and V. A. Barker, Society for Industrial Mathematics, 2001

[14] *Das kleine Finite-Elemente-Skript*, A. Jüngel, Technical University of Vienna, 2001, `http://www.asc.tuwien.ac.at/~juengel/scripts/femscript.pdf`

[15] *Partielle Differentialgleichungen. Sobolevräume und Randwertaufgaben*, J. Wloka, Teubner, 1982

[16] *An Introduction to Partial Differential Equations*, M. Renardy and R. Rogers, Springer, 1993

[17] *ViennaCL 1.2.1 User Manual*, K. Rupp, Technical University of Vienna, 2011 `http://viennacl.sourceforge.net/viennacl-manual-current.pdf`

[18] *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems*, H. A. Van der Vorst, SIAM Journal on Scientific and Statistical Computing, Vol. 13, No. 2, pp. 631-644, 1992

[19] *GMRES: A Generalied Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, Y. Saad and M. H. Schultz, SIAM Journal on Scientific and Statistical Computing, Vol. 7, pp. 856-869 1986

[20] *An Automatic OpenCL Compute Kernel Generator for Basic Linear Algebra Operations*, P. Tillet, K. Rupp, S. Selberherr, Spring Simulation Multiconference (SpringSim 12), Florida; 2012-03-26 - 2012-03-29; in: Proceedings of the Spring Simulation Multiconference 2012

[21] *ViennaCL compilation and transfer times*, K. Rupp, Intel Developer Forums, 2011, `http://software.intel.com/en-us/forums/showthread.php?t=81682&o=a&s=lr`

[22] *Efficient Sparse Matrix-Vector Multiplication on CUDA*, N. Bell and M. Garlandy, NVIDIA Technical Report NVR-2008-004, 2008

[23] *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*, D. Göddeke, PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, Logos Verlag, Berlin, May 2010

[24] *Accelerating Double Precision FEM Simulations with GPUs*, D. Göddeke, R. Strzodka and S. Turek, 18th Symposium Simulation Technique (ASIM'05), Erlangen, Germany, Sep. 2005

[25] *Eigen C++ library*, `http://eigen.tuxfamily.org/index.php?title=Main_Page`

[26] *Matrix Template Library 4*, `http://www.simunova.com/de/node/65`

[27] *Implementing the Chebyshev Polynomial Preconditioner for the Iterative Solution of Linear Systems on Massively Parallel Graphics Processors*, A. Asgasri and J. E. Tate, Proc. CIGRE Conf. Power Systems, 2009

[28] *A Parallel Algebraic Multigrid Solver on Graphics Processing Units*, G. Haase et al., High Performance Computing and Applications (Lecture Notes in Computer Science), vol. 5938, p. 38-47, 2010

[29] *Parallel Preconditioning with Sparse Approximate Inverses*, M. J. Grote and T. Huckle, SIAM J. Sci. Comp., vol. 18, no. 3, p. 838-853, 1997